Security and Privacy of Smart Devices

Dissertation Proposal

Yinhao Xiao

B.S. in Information and Computing Science,
May 2012, Guangdong University of Technology
M.A. in Mathematics,
May 2014, The George Washington University
M.S. in Computer Science,
Dec 2015, The George Washington University

A Dissertation Proposal submitted to
The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Philosophy

December 11, 2018

Dissertation directed by

Xiuzhen Cheng
Professor of Computer Science

Security and Privacy of Smart Devices

Dissertation Proposal

Yinhao Xiao

Dissertation Proposal Committee:

Xiuzhen Cheng, Professor of Computer Science, Dissertation Director

Hyeong-Ah Choi, Professor of Computer Engineering, Committee Member

Arkady Yerukhimovich, Assistant Professor of Computer Science, Committee Member

Xiang Chen, Assistant Professor of Computer Engineering, Committee Member

Abstract

The advent of smart devices, i.e. smartphones and smart home devices, have greatly revolutionized and modernize people's daily lives in every aspect. Yet, the security condition of the devices and their corresponding systems is concerning since traditional security measures fail to cope with them due to limitations of computation power and hardware/firmware heterogeneity. In this dissertation proposal, we present our initial research on studying the security and privacy of smartphones on OS-Level and side-channel level.

Firstly, we study the OS security of Android devices. In order to facilitate apps to collaborate to finish complex jobs, Android allows isolated apps to communicate through explicit interfaces. However, the communication mechanisms often give additional privilege to apps, which can be exploited by attackers. The Android Task Structure is a widely-used mechanism to facilitate apps' collaboration. Recent research has identified attacks to the mechanism, allowing attackers to spoof UIs in Android. In this work, we present an analysis of the security of the Android task structure. In particular, we analyze the system/app conditions that can cause the task mechanism to leak privilege. Furthermore, we identify new end-to-end attacks that enable attackers to *actively* interfere with victim apps to steal sensitive information. Based on our findings, we also develop a task interference checking app for exploits to the Android task structure.

Secondly, we study how the side-channel information publicly available in Android devices can result in severe privacy leakage on social networks. Owing to the various features provided by mobile devices, a user's online social activities are tightly tied to his phone, and are conveniently, sometimes unnecessarily, available to social networks. In this work, we propose a novel attack architecture to show that attackers can infer a user's social network identities behind a mobile device through new dimensions. Specifically, we first developed a correlation between a user's device system states and the social network events, which leverage multiple mechanisms such as learning-based memory regression model, to infer the possible accounts of the user in the social network app. Then we exploited the social network to social network correlation, via which we correlated information across different social networks, to identify the accounts of the target user. We implemented and evaluated these attacks on three popular social networks, and the results corroborate the effectiveness of our design.

Lastly, we plan to extend our research on studying the security and privacy of smart home devices and their corresponding smart home systems by proposing novel vulnerability-discovery measures, excavating new attack vectors and designing effective defense mechanisms.

Table of Contents

List of Figures

List of Tables

Introduction

Smart devices refer to the electronic devices that are connected to the cloud or edge servers or with each other through different wireless protocols (e.g., Wi-Fi, 4/5G, Zigbee, Bluetooth, Software-Defined Radio, RFID, etc) in order to complete ubiquitous “smart” tasks such as sensing, controlling, actuating, messaging, or even decision making. Smart devices can mainly be taxonomized into two categories: smart mobile devices (i.e., smartphones and tablets), and smart home devices (e.g., smart thermostats, smart light, smart switch, and smart speakers). According to market research conducted by Statista, the total revenue of the smart devices in the US has reached 79.8 billion dollars by the year 2018 [63] [64]. It is also projected that the average number of networked smart devices per person can reach 13 by the year of 2021 [49].

With the presence of the prevailing performance with respect to market sharing, as shown above, one may raise a question: If security vulnerabilities of smart devices and their corresponding systems are discovered by vicious personnel who later exploits these vulnerabilities to conduct malicious activities, are the consequences more serious than the ones of traditional computer or Internet infrastructure? Unfortunately, the answer is positive for this question due to the following two reasons:

- Smart devices tend to reflect more personal information than the tradition computers do. For example, using smartphones to take pictures has become the main way of photo-taking [50]. Hence, the smartphone is the major source to store a user’s personal photos. It is not difficult to envision the devastating consequence if an attacker can access the resources in a smartphone without authorization through security breaches.

- Smart devices are interconnected tightly for better functioning and automation (e.g., a smart home system). The security drawback of the design allows an attacker to take down the whole system more quickly and effectively by compromising a single device, to begin with. A notable example is the Mirai virus which managed to infect 200,000 - 300,000 IoT smart devices in the first 20 hours [2].

Even though defending smart devices against malicious exploits and attacks is urgent and critical, securing them is more challenging than securing traditional computers not only because they are computationally limited to empower high-end firewalls, but are highly heterogeneous both on the hardware level and firmware level. Having observed the situation, we mainly focus on identifying vulnerabilities, inventing attacks, and developing corresponding defensive mechanisms for smart devices. In this proposal, we specifically target the security of smartphones and their corresponding systems. In the following contexts, we focus on two main directions of studying the security and privacy of smartphones, i.e., from the perspective of system design flaw and from the perspective of side-channel information exploit. For the system design flaw, we mainly present an OS-Level logic flaw existing in Android Task Mechanism that results in severe privilege leakage, through which we implemented four proof-of-concept attacks. Lastly, we developed an efficient scanner to help users to avoid the attacks. For the side-channel information exploit, we mainly present a novel attack vector which effectively associates the user's identity in social networks with the smartphone device using Android side-channel information.

1.1 Exploiting Android Task Mechanism

The Android system's security is based on several layers of security mechanisms. In particular, each app is assigned a set of permissions and is only allowed to access system resources and services within the permissions given. In addition, to prevent apps from accessing information of others, each app is confined into its own partition. It is enforced using the process isolation and user-based protection mechanisms provided by Linux, where each app is assigned to a unique Linux user ID.

Strong isolation increases the bar for attackers to carry out malicious activities, but it also hinders benign apps from communicating and collaborating with one another. To facilitate apps collaboration in a complex task, Android allows isolated apps to communicate through explicit interfaces, such as the Intent mechanism. For example, Instagram uses intents to access the Single Sign-On (SSO) service of Facebook to authenticate users. Furthermore, Android provides the Android Task Structure mechanism to allow activities from different apps to be seamlessly integrated into a task, giving them the convenience when accessing common information. For example, when Instagram uses the Facebook API for the authentication service provided by Facebook, users can navigate through activities Instagram app and Facebook app as if they are the same app.

Though the mechanisms are designed for facilitating app communication and collaboration, relaxing the isolation provided by the Android system often causes over-permissive privilege to apps. As the task mechanism of Android is developed to facilitate inter-app collaboration, apps in a task may get additional privilege beyond what is allowed by the isolation-based Android security mechanism. Demonstrated by recent exploits [56], a malicious app can hijack the task mechanism for attacks such as spoofing and phishing. The privilege obtained by apps in the same task is well beyond that for collaboration, effectively making the Android task mechanism a form of authorization.

In this work, we conduct an analysis of the security of the Android task mechanism. First, we analyze possible ways that an app can join a task and the privilege "leaked" to other apps in the same task. Specifically, to explore the ways Android controls tasks, we dynamically probe possible combinations of the flags and system states that can affect an app's task status. We also analyze the additional privilege that can be obtained by an app when it joins a task. Second, built on the understanding of the task control mechanism and task privileges, we identify end-to-end attacks that steal information from other apps. In particular, we identified four proof-of-concept attacks based on exploiting the task mechanisms. The attacks include UI phishing, screenshot based password stealing, activity-in-the-middle activity, and gallery stealing. All of them only require common permissions, e.g., `INTERNET` and `READ_EXTERNAL_STROAGE`. Compared to the attacks by existing exploits, we have identified new attack mechanisms that can *actively* interfere with benign apps. The short video demos can be found in [72]. Finally, to prevent attackers from misusing the task mechanism, we develop an efficient scanner that can help users to identify the risks related to Android tasks.

The details of this work is presented in Section 2.2.3.

1.2 Device-Identity Association Attack Using Android Side-Channel Information

Online social networks such as Twitter and Flickr revolutionize the ways people interact with each other. More and more users begin to access social networks through mobile devices, making it completely possible to de-anonymize a target individual based on the side-channel information from his mobile device due to the rich features generated by the user's device while operating on social network apps. We call this kind of attack *device-identity association*. Though it is relatively difficult for an attacker to trick users into installing a malware with strong permissions, recent research shows that public information obtained from the Android system without permission can be used to link to a user's identity [78]. Nevertheless, the attack method proposed in [78] by observing the fixed TCP payload sequence pattern is no longer effective due to the recent updates of the social network apps. We manually monitored the Twitter app's TCP sequence while tweeting and found that the sequence was irregular and noisy. Then we used Frida, an injection framework [14], to trace the function calls of the Twitter app, and found that the root cause lies in that the Twitter app no longer handles each tweeting event in a separate TLS session (separate calls to the `url.openConnection()` function), but rather combining them into one TLS session (only one call to the `url.openConnection()` function).

We also observed that the attack based solely on the mobile device side-channels faces challenges due to limited information from one social network app. Nevertheless, in practice, the majority of social network users access their accounts very infrequently, according to our study on 500,008 Twitter accounts); and a user typically has accounts in different social networks such as Twitter, Flickr, and Instagram; furthermore, the social network accounts of the same user are often highly similar/correlated in their user profiles, e.g., user name, picture, and locations. Such facts reveal a high potential of device-identity association based on account correlations.

The objective of this work is to investigate efficient and effective techniques to accurately identify a target user (more precisely, the social network accounts of the user) in social networks even though the information obtained by an attacker is limited. In order to accomplish this goal, we proposed a novel attack architecture with two attack vectors, correlation from device system states to a social network (DS-SN) and correlation for cross social networks (SN-SN). For the DS-SN attack, we studied the association between a user's identity in a social network and the user's device system states, e.g., memory and network data. In our threat model, an attacker can get information from the system level of a user's smartphone through installed apps without any permission or the user's consent. Leveraging these states, the attacker can infer the system events, e.g., activity transitions and keyboard status, which can be used to further infer the user's social network events, e.g., sending tweets and posting Instagram photos at certain timestamps. The attacker then collects and aggregates these social network events to identify the target user's identity in the social network. However, DS-SN is sometimes not enough to identify a user's identity due to limited social network events the user leaves in a single social network; therefore, we proposed the idea of SN-SN attack, which exploits the cross social network similarity to allow the attacker to efficiently and accurately figure out the identities of a user through the system and network state left by the user's activities and account profiles on the social networks.

The details of this work is presented in Section 3.6.

Preliminaries

2.1 Android Security Basis and Android Task Mechanism

Components in Android include *Activity*, *Service*, *Content Provider*, and *Broadcast Receiver*. An activity, representing a single screen with user interface, is the most basic elements in Android OS. A service in Android is a UI-less component running in the background. A content provider supplies data from one application to another through methods of the `ContentResolver` class with the ways of storing data in databases, in files, or over the network. A broadcast receiver responds to broadcast messages from other applications.

Android adopts several layers of isolation and sandboxing mechanism as its basic mechanism of security. In particular, it defines a set of permissions to control the access of apps. Apps can access specific resources only if they are granted with required permission. In addition, Android uses user-based protection of Linux to isolate apps. It allocates a unique Linux user ID to each app, which naturally isolates the app from others using the process-based isolation mechanism provided by the Linux kernel.

The Android task mechanism is designed for facilitate inter-app communication and for better support app collaborating under same tasks. It allows activities from different apps can reside in the same task to perform communications more conveniently [34]. As an example, when the user clicks a "feedback" button from an activity of a game app, Android starts the composer activity of an email app, and puts it onto the game app's activity. After the user finishes sending email, the composer activity is put off and the game app returns to the top. In such a way, two activities are organized to finish a task, while they are actually from different apps.

The Android task mechanism is affected by several flags of apps. The following are the key attributes affecting how apps are grouped.

launchMode: This is the attribute which decides how an activity will be launched. It has four values, i.e., *standard*, *singleTop*, *singleTask* and *singleInstance*. Activities with *standard* or *singleTop* can be instantiated multiple times while activities with *singleTask* or *singleInstance* can only begin a task and be the root of the task. Moreover, *singleInstance* does not permit other activities to be part of its task. An activity without *launchMode* specifically set is assumed to be *standard* by default.

taskAffinity: Activities with the same *taskAffinity*, normally the name of the package, conceptually are in the same task, but this is not always the case. We refer to Section 3.2 and Table 2 for more details. An activity without this attribute set is assumed to have the same *taskAffinity* as its own package name.

allowTaskReparenting: This is a boolean attribute indicating whether an activity can be moved to the task which has the same *taskAffinity* from the original task it is started. An activity without this attribute specifically set is assumed to be *allowTaskReparenting=false*.

For better demonstration, consider the example where an app has the functionalities of viewing contacts as well as sending emails to contacts. The app has two activities for these two functionalities, *SendEmailActivity* and *ViewContactActivity*. For better IPC and logic concerns, designers of the app set the *taskAffinity* of *SendEmailActivity* to be the same as the system email app, the *taskAffinity* of *ViewContactActivity* to be the same as system contact app and *allowTaskReparenting* to be true for both activities.

2.2 Publicly Available System States in Android

In this subsection, we present the publicly available side-channel state information in Android that are exploited by our proposed inference attacks detailed in Section 3.6. We focus on three common channels of public information, namely memory, CPU, and network.

2.2.1 Memory

The Android system leverages Android Runtime (ART) or its predecessor, Dalvik Virtual Machine (DVM), as the runtime environment to execute binaries in the Dalvik Executable

(DEX) format. Both ART and DVM use the paging and memory-mapping mechanisms [18] to manage memory allocation of apps. Particularly, Android maintains four types of memory data for each process running on the system: `Virtual Set Size` (*VSS*), `Resident Set Size` (*RSS*), `Proportional Set Size` (*PSS*), and `Unique Set Size` (*USS*), which are listed in Table 1.

Table 1: Memory Size Information

Type of Memory Size	**Description**
Virtual Set Size (VSS)	Total virtual memory of a process
Resident Set Size (RSS)	Total physical memory of a process
Proportional Set Size (PSS)	Memory shared between a process and other processes
Unique Set Size (USS)	The set of pages unique to a process

Android does not treat memory information as sensitive system data. Consequently, an app can obtain other apps' memory information without requesting any permission. There are four methods to retrieve the memory data of an app, which are listed below:

- The Android API has a standard class `Debug.MemoryInfo`, which provides complete interfaces to query the information for a process [24], including private dirty pages, shared dirty pages, and the PSS for Dalvik.

- The `/proc` file system has `/proc/pid/statm` (`pid` is the id of the process) that lists all four types of memory information: *VSS*, *USS*, *RSS*, and *PSS*.

- The commands "`top`" and "`ps`" in Android Toolbox [10] yield *VSS* and *RSS* for any given process. These commands also use information from the `/proc` file system.

- In some devices, `/system/xbin/procrank` is provided. The command `procrank` yields all four types of memory information for all processes in real time.

2.2.2 CPU Usage

Android maintains three types of CPU usage data for a process, which are publicly available:

- CPU usage rate: the total percentage of time a CPU operates on a running process, with 100% indicating that for a given period, the CPU spends all its available cycles running the specific process.
- User time (`utime`): the CPU time spent in the user code of a process, measured in clock ticks.
- System time (`stime`): the CPU time spent in the system code (which is the kernel) of a process, measured in clock ticks.

A zero-permission app can retrieve the CPU usage data of another app by either (1) accessing `/proc/pid/stat` which lists *user time*, *system time*, and other time-related information; or (2) using commands "`top`" and "`ps`" with Android Toolbox to return the user time, system time, and CPU usage percentage of any given process.

2.2.3 Network

Android does not store the contents of TCP packets of any process; but it maintains a record of the number of bytes sent and received through TCP connections by a process. This information is available to any zero-permission app, and can be obtained by the following ways:

- The `TrafficStats` from Android API has the method `getUidRxBytes(int uid)`, which returns the received bytes for a given user ID `uid`.
- The `/proc` system files `/proc/uid_stat/pid/tcp_snd` and `/proc/uid_stat/pid/tcp_rcv` respectively maintain the bytes sent and received for an app.

Privilege Leakage and Information Stealing through the Android Task Mechanism

3.1 Threat Model and Approach Overview

In this subsection, we present the threat model and the overview of our attack approach.

3.1.1 Threat Model

To study the security of the Android task mechanism, we consider a scenario as follows. There are two apps, i.e., AppB and AppM, installed in the same Android device, where AppB is benign and AppM is developed by attackers. We assume AppM does not have to require any permission to manipulate tasks. However, we assume AppM can be granted permissions for following-up behaviors, such as sending the retrieved information out or accessing local storage.

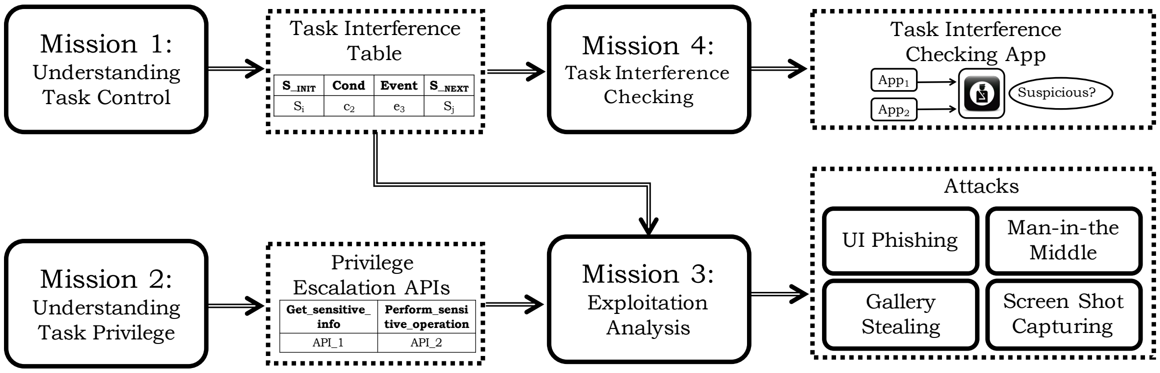
3.1.2 Approach Overview

The goal of our work is to comprehensively analyze the Android task mechanism to identify attacks, as well as creating solutions to prevent such attacks.

As shown in Figure 1, our research consists of four components: *Understanding Task Control*, *Understanding Task Privilege*, *Exploitation Analysis* and *Task Interference Checking*.

The first component is to analyze the Android task mechanism, identifying control conditions that can be leveraged by attackers. We aim to find out the interference of tasks between two apps, and identify the dominating factors deciding the apps' property. Secondly, we focus on studying the additional privilege apps obtained when two apps are in the same task. We delicately test sensitive system APIs and compare the difference of the results before and after apps in the same task. Based on the understanding from the previous stages, to demonstrate the achievability and severity of the privilege escalation against Android task mechanism, we develop four light-weight real-world attacks that can steal the

Figure 1: Approach Overview


Mission 1: Understanding Task Control
Task Interference Table
S_INIT Cond Event S_NEXT
Si c2 e3 Sj
Mission 4: Task Interference Checking
Task Interference Checking App
App1
App2
Suspicious?
Attacks
UI Phishing
Man-in-the Middle
Gallery Stealing
Screen Shot Capturing
Mission 2: Understanding Task Privilege
Privilege Escalation APIs
Get_sensitive_info Perform_sensitive_operation
API_1 API_2
Mission 3: Exploitation Analysis

sensitive information successfully, most of which only requires `INTERNET` permission. At last, we design a task interference checking app to detect the task interference between users' important apps and other installed apps.

3.2 Security Analysis of Android Task Structure

In this subsection, we introduce our approach. We focus on two aspects of the Android task mechanism. First, We analyze the conditions that affect Android task control to identify different ways that can include an app into a task. Second, we explore the privilege an app can get when it is included into a task. These are two necessary components to identify new attacks.

3.2.1 Understanding Android Task Control Conditions

We explore the conditions and actions of Android task control through dynamic testing. To do this, we examine the Android documentation [34] to create test cases to drive the exploration. Our goal is to check the influence of the flags introduced in Section 1.2 on the task mechanism.

Testing Methodology. We implemented the two template apps introduced in Section 3.1, AppB and AppM, as the inputs to drive the testing process. For each combination of the task-control-related flags, such as *launchMode* and *taskAffinity*, we set the corresponding value in app templates, create a pair of AppB and AppM, and test them with different sequences of launching events, e.g., using the Android Launcher to start AppB (denoted as Launcher→AppB) or using AppB to launch AppM (denoted as AppB→AppM). During our test, AppB's *taskAffinity* is set to "TaskB". We only test the conditions where AppM's *allowTaskReparenting* is set to "true", as a "false" value in this flag will not result in task interference.

The results are summarized in Table 2. We are interested in cases with potential task interference, i.e., AppM ends up running as part of TaskB. We mark the cases for task interference, i.e., AppM running as part of TaskB, with an asterisk "*". The cases without an asterisk attached are considered to be safe. We list four identified dangerous cases below.

- **Case 2**. Under the conditions of this case, AppM is launched first by the Android Launcher, followed by AppB. Only AppM runs at the foreground, while AppB cannot be executed.

  In this case, AppM blocks AppB from execution, which is a case of denial-of-use to AppB.

- **Case 4**. Under the conditions of this case, AppM is launched by the Android Launcher, followed by AppB. AppB runs in the foreground, and AppM runs in the background, both in TaskB.

- **Case 9**. Under the conditions of this case, the Android Launcher starts AppM. AppM then starts AppB. AppB runs in the foreground, and AppM runs in the background, both in TaskB.

- **Case 10**. Under the conditions of this case, the Android Launcher starts AppB. AppB then starts AppM. AppM runs in the foreground, and AppB runs in the background, both in TaskB.

Whether two Apps are in the same task can be determined by viewing the *Recents* screen which renders all processes that were opened since last clearance [19]. *Recents*

Table 2: Task Interference Table

Case #	Initial Conditions				Events	Resulting State		
	AppB	AppM				AppB	AppM	
	LaunchMode	LaunchMode	TaskAffinity	Reparenting		Status	Status	Task
1	standard or singleTop or flag(SINGLE_TOP)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppB; Launcher→AppM	F	X	-
2 *	standard or singleTop or flag(SINGLE_TOP)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppM; Launcher→AppB	X	F	TaskB
3	singleTask or flag(NEW_TASK)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppB; Launcher→AppM	F	X	-
4 *	singleTask or flag(NEW_TASK)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppM; Launcher→AppB	F	B	TaskB
5	singleTask or flag(NEW_TASK)	singleTask or flag(NEW_TASK)	TaskB	True	Launcher→AppB; Launcher→ AppM	B	F	TaskM
6	singleTask or flag(NEW_TASK)	singleTask or flag(NEW_TASK)	TaskB	True	Launcher→ AppM ; Launcher→AppB	F	B	TaskM
7	singleInstance or	any or	TaskB	True	Launcher→AppB; Launcher→AppM	F	X	-
8	singleInstance or	any or	TaskB	True	Launcher→AppM; Launcher→AppB	F	B	TaskM
9 *	standard or singleTop or flag(SINGLE_TOP)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppM; AppM→AppB	F	B	TaskB
10 *	standard or singleTop or flag(SINGLE_TOP)	standard or singleTop or flag(SINGLE_TOP)	TaskB	True	Launcher→AppB; AppB→AppM	B	F	TaskB

We assume AppB is running with the task "TaskB." In the events, the operation A→B stands for A launches B. In the resulting state's status, F stands for execution in foreground; B stands for execution in background; X stands for not-running.

screen is rendered when a user presses the *Recents* button which is located at the third from left to right at the button bar followed by *Back* and *Home* buttons. If two Apps are in the same task, *Recents* screen will only show one process other than two when they do not reside in the same task. Note that according to our experiment, flags such as `FLAG_ACTIVITY_CLEAR_TOP` and `FLAG_ACTIVITY_REORDER_TO_FRONT` do not pose a difference than the `SIGLE_TOP` in our case. Therefore, we only include the `SINGLE_TOP` flag in our Table 2.

3.2.2 Understanding Privilege Obtained in the Same Task

From the results in Table 2, we can see that AppM has several ways to be included into the task of AppB, often without involving actions from other apps or the system. Next, we explore the privilege obtained through the Task mechanism, including privilege for retrieving other apps' information and privilege for changing other apps' states. Therefore, if the privilege given to apps in the same task allows them to carry out dangerous actions, it can be potentially misused by the malicious app.

Retrieving Information of Other Apps

Figuring out the execution state of a victim app, such as whether it is running and which activity is in foreground, is often used as the first step in several attacks, such as UI hijacking [8]. Therefore, Android by default disallows one app from directly querying another app's runtime information from through the Android sandbox policy.

In older versions of Android, there were APIs allowing inter-app runtime information checking. For devices that are prior to *Android Lollipop (v5.0)*, directly calling the API `getRunningTasks(int maxNum)` will return the information of as many as `maxNum` running activities [20]. However, this function is deprecated after *Lollipop* since allowing third-party apps to invoke the function directly will cause information leakage in important apps.

For devices prior to *Android MarshMallow (v6.0)*, directly calling `getRunningAppProcesses()` returns a list of application processes that are running on the device [20]. This function returns a *RunningAppProcessInfo* object, which includes a member variable called *importance* that represents the importance level that the system places on the process [33]. It has one of these values: `IMPORTANCE_FOREGROUND`, `IMPORTANCE_VISIBLE`, `IMPORTANCE_SERVICE`, `IMPORTANCE_BACKGROUND` and `IMPORTANCE_EMPTY`. If *importance* is `IMPORTANCE_FOREGROUND`, the corresponding process is running in the foreground. This method, however, cannot accurately point out which activity running in the foreground since it operates on a process level and accesses only the package name. This method is also no longer supported for *MarshMallow* devices with API level 23 unless the third-party app who is making a call to this function has the same process ID as the target process.

Getting App Running Information in a Task. Although the Android API `getRunningTasks()` is deprecated for direct usage, we have found that it still works if the calling app and the target app are in the same task. The official documentation of `getRunningTasks()` does not explicitly point it out but only states that if it is called, this function only returns a small subsets of information, e.g., the information of the caller's own task and home task which is considered to be not sensitive [20].

Changing States of Other Apps

UI Injection. Ideally, if an app is running in the foreground, other apps isolated from this app should not perform sensitive operations on it. Chen *et al.* [8] show two UI-injection methods that do not require any permissions: (1) starting an `Activity` by setting `lauchMode=singleInstance`. This is also how most system apps, such as the alarm app, pop up a window on the top of another app [22]; (2) starting an `Activity` from the Android broadcast receiver [21].

In our analysis, we find that if two apps are in the same task, the UI-injection attack becomes easier. The attacker can simply call the function `startActivity()` to achieve the same effect. Through understanding the Android SDK source code, `startActivity()` invokes `startActivityForResult()` with `requestCode=-1`. Later `startActivityForResult()` invokes `execStartActivity()` in the Instrumentation class, a base class for implementing application instrumentation code [28]. `execStartActivity()` then checks the base package of the calling activity by invoking `getBasePackageName()`. If the base package of calling activity matches with that of target activity, the Android system launches the target activity.

Terminating UI. Android does not allow a third-party app to terminate a running app unless they are in the same process. Although an app is not in the foreground, the Android system still allows any third-party apps to make calls to `killBackgroundProcesses()` [29] to terminate any specific background app. There is no direct way to terminate a foreground-running activity.

Unlike `startActivity()`, even if two apps are in the same task, calling `finish()`

or `finishActivity()` will not terminate the foreground activity, but will terminate the activity that makes the call. However, we found that as long as two apps are in the same task, calling the API `finishAndRemoveTask()` results in terminating the whole Task regardless of whether an activity is running in the foreground [26] or not. This provides the malicious app an interface to terminate other apps which are within the same task as it.

3.3 Information Stealing Attacks

Based on analysis from Section 3.2, we develop four proof-of-concept attacks: *UI Phishing*, *Activity-in-the-middle Attack*, *Gallery Stealing*, and *Screen Shot Capture*. These four attacks demonstrate the severity of the security problems we identified. In this subsection, the attacks are illustrated using two most popular social apps, Instagram and Facebook. Except Gallery Stealing, which requires `READ_EXTERNAL_STORAGE`, all other three attacks only require the `INTERNET` permission (in order to send out the information stolen).

3.3.1 UI Phishing

UI Phishing is a popular type of attacks to spoof users. The difficulty of phishing attack is to decide the timing when the spoofing interface should be prompted, in order to prevent the victim from noticing it. Ren *et al.* [56] introduced "Back Hijacking", which directs users to a spoofed bank *LoginActivity*. The key difference is that the attack method identified in our approach *actively* interact with the victim app using the privilege obtained through the approaches we discussed in Section 3.2.2.

Our proof-of-concept UI Phishing attack is implemented against the scenario when a user logs in to Instagram using his/her Facebook account. In order to ease understanding the phishing attack against this scenario, we brief the Facebook SSO service. According to the Facebook Android developer documentation [25], Facebook SDK takes three ways for apps who require Facebook Login as part of functionality:

- **Native App Login.** If the Android device already has the Facebook app installed, pressing the Facebook login button directly opens the Facebook app where user can log in his/her Facebook account and grants permission to the third-party app to access his/her Facebook personal information.

- **Chrome Custom Tab Login.** If the Android device does not have the Facebook app installed, a third-party app can ask the Chrome browser to open Facebook login page by registering a scheme in the `Manifest` with the format `fb+facebook_app_id`.
- **WebView Login.** Finally if neither the Android device has Facebook app installed, nor does the third-party app register for Chrome Custom Tab Login, an embedded WebView will be launched dynamically rendering the content of Facebook login page.

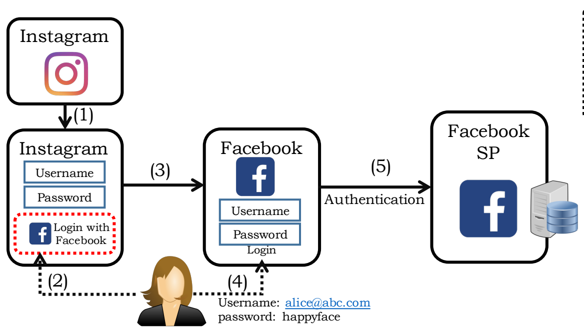
Our attack works when a user presses the "Log in with Facebook" button in Instagram. We assume that a user already has Facebook App installed in the device. Instagram includes Facebook Login as a part of its functionality. It does not register for Chrome Custom Tab Login. In other words, after pressing the "Log in with Facebook" button in Instagram, it will launch Facebook App directly. The package name of Facebook app is `"com.facebook.katana"`, the package name of self-build WebView is `"com.android.webview"` and the package name of Instagram is `"com.instagram.android"`. Shown in Figure 2, it works in the following steps.

(1) Attacking app declares the same task as Instagram. This step can be achieved by setting the attributes of *taskAffinity* and *allowTaskReparenting* as follows.

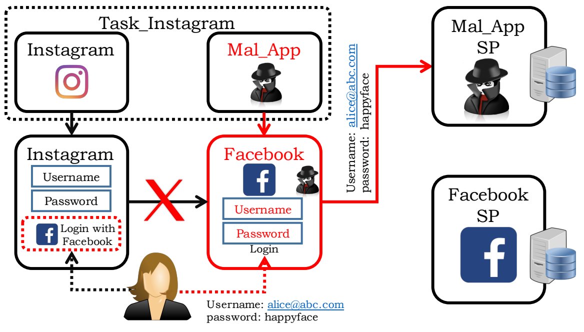
```
<activity
            android:name=".UIPhishingActivity"
            android:allowTaskReparenting="true"
            android:taskAffinity="com.instagram.android">
```

(2) The launcher first launches the attacking app, which launches Instagram immediately through invoking the task parent since Instagram is the parent of the current task.

Figure 2: UI Phishing



(a) Single-Sign-On Architecture



(b) UI Phishing Attack

(a) When the user selects ”login with Facebook” in Instagram, Instagram will launch the Facebook app. After the user inputs username, password and finishes the authentication with the Facebook service provider, Instagram gets the permission to access the user’s Facebook resources. (b) The malicious app Mal_App declares same taskAffinity as Instagram. Mal_App creates a background thread and monitors the running states of Instagram. When the user selects Facebook login button and Instagram launches the Facebook activity, Mal_App will override Facebook with a fake login UI. The username/password input by the user will be sent to Mal_App’s service provider.

Before Instagram’s `setContentView(R.layout.main)` function can be called, the attacking app launches Launcher (Home), meaning that Instagram cannot be displayed in order to achieve the stealthiness. The process can be completed in a fairly short time and will not be noticed by the victim with bare eyes. After that, our attacking app waits for victim to launch Instagram herself.

(3) The attacking app then creates a background thread which runs in a cycle of, we devise, every 100 milliseconds. In the meantime, it checks whether there is a change in the foreground package name, i.e., in our case, the package name changes from `"com.android.webview"` to `"com.facebook.katana"`. This can be achieved by simply making API call to `getRunningTasks()`. Once the foreground package has been changed to Facebook, our attack pops up our counterfeit Facebook *LoginActivity* overriding the real one. The spoofed UI collects the user’s login information and send it to adversary’s back-end server.

3.3.2 Activity-in-the-middle Attack

The Facebook SSO process is based on OAuth 2.0 [6]. The main steps of this mechanism are: (1) When the user opens the relying party (RP) app, in our case, Instagram, it passes its `Facebook_app_id` and directed URL to the Android System. (2) The Android system then redirects user to the Service Provider (SP), in our case, Facebook, and passes it the `Facebook_app_id`. (3) If the user requests to grant the permission to the RP, SP will issue an access token to the Android system. (4) The Android system passes the access token to RP, by which RP is able to access the user's protected resource on SP.

Facebook has two types of access tokens, short-term and long-term [11]. Short-term access token lasts several hours and long-term lasts for 60 days. Unless specifically required, Facebook usually issues short-term access token. RPs use graph API provided by Facebook to retrieve protected resources hosted on Facebook server [12]. Graph API is an HTTP-based API, which is implemented as the following URL:

```
https://graph.facebook.com/me?fields=xxx&access_token=xxx
```

In other words, any party who has the access token can access the user's protected resources hosted on Facebook servers.

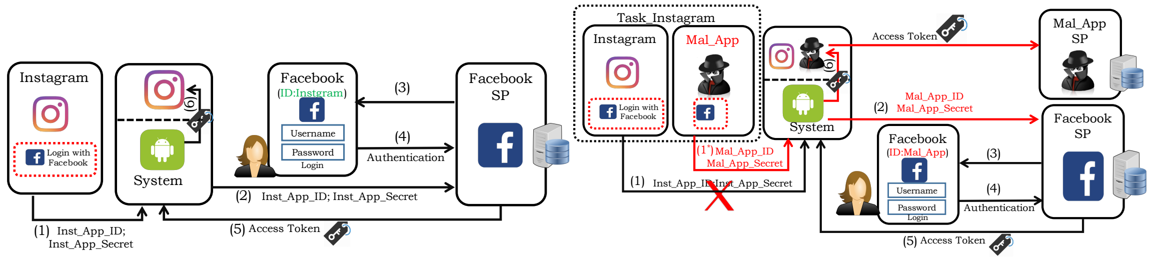
In Android, redirecting to the Facebook app is done by intent transition based on `startActivityForResult()` and `onActivityResult()`. We implement an MITM attacking app whose model runs in the following steps, shown in Figure 3.

(1) It follows the first three steps in the UI Phishing attack model.

(2) Instead of popping up a phishing login page like UI Phishing, the MITM attack pops up a transparent activity, which blocks the traffic that is supposed to be relayed to Facebook server from user's device system and passes our own traffic to Facebook. This step can be easily realized by creating an invisible Facebook Login Button and sending a button pressed event itself by invoking `mFacebookLoginBtn.performClick()`.

(3) After the user grants the permission, which he/she intends for Instagram, the MITM app retrieves the access token. Since Instagram runs in the background, once the foreground

Figure 3: Activity-in-the Middle Attack on SSO



(a) Single-Sign-On Permission Granting Architecture

(b) Activity-in-the-Middle Attack

(a) When a user wants to login with Facebook in the Instagram, Instagram sends system its application ID with application secret, Inst_App_ID and Inst_App_Secret. Android system forwards request with Inst_App_ID and Inst_App_Secret to Facebook service provider and retrieves Facebook Login Activity bound with Inst_App_ID. The user inputs her/his username/password for authentication. Facebook service provider returns the access token to Android system if the authentication passed. Android system sends the access token to Instagram according to the application ID that system received. (b) The malicious app Mal_App overrides the ownership of the Instagram task in the same way as UI-phishing. It monitors the Instagram running status and sends system its Facebook login request (including Mal_App_ID and Mal_App_Secret) before the Instagram's request arrives Android system. Android system sends Mal_App's request to Facebook service provider and drops the same login request arrived later. Facebook return Login Activity bound with Mal_App_ID. The user inputs her/his username/password for authentication. Facebook service provider returns the access token to Android system if the authentication passed. Android system sends the access token to Mal_App according to the application ID that system received.

finishes, Instagram will be invoked.

(4) To finalize the process, the adversary needs to verify `APP_ID` and *App Secret* with Facebook. An adversary can either register its attacking app in Facebook Developer Website and use its own `APP_ID` and *App Secret* or steal other RP apps' ID and Secret. Facebook RP Apps will post an HTTP message when user system launches native Facebook App.

3.3.3 Gallery Stealing

Starting from Android 6.0 (API level 23), in order to access gallery, an app has to request for `READ_EXTERNAL_STORAGE` permission at runtime rather than at the installation time, which is classified as one of the dangerous permissions [32]. This mechanism provides more secure and flexible protection to user's photo gallery. However, this new security mechanism can be bypassed by exploits to the Android task mechanism, as shown in this subsection.

Timing. For Android devices before *marshmallow* (API level lower than 23), permissions are requested at the time when an app is being installed. Apps with suspicious permissions will easily trigger the user's attention and likely be denied access. However, with the new requesting-permissions-at-run-time mechanism, our attacking app can avoid requesting permission when being installed since it is requested at the run time. To figure out the suitable timing, it continuously monitors Instagram. Once the victim clicks the "Camera" button on Instagram which allows Instagram to access the gallery and camera, our attacking app instantly kills Instagram and pops up our requesting dialog for access photos in gallery. User mistakenly believes that he/she is granting the permission to Instagram. After the permission is granted, our attacking app retrieves all the photos from user's gallery and send them to the back-end server. In order to be stealthier, the attacking app pops up a dialog shows "System encounters errors" and finally kills itself.

Permission Dialog. Even though timing improves the naturalness of the attack, Android permission dialog shows the name of the app in bold who makes the request. We propose two ways to circumvent this problem.

(1) By employing the idea of social engineering, the attacker can name the attacking app using a name similar to the target app. In our case, e.g., "Instgram" or "Instagam". However, naming the app in this way can hardly pass the review of Google Play. Even if it does, careful users may still notice the difference. Therefore, in our attack, we do not use this method.

(2) Employing the tapjacking. The idea of tapjacking is putting message the attacker wishes to display on the top of the real system message by setting a window layout flag `TYPE_SYSTEM_OVERLAY`. Android realizes the potential threat and adopts the mechanism `MotionEvent.FLAG_WINDOW_IS_OBSCURED` which alerts the real dialog is being overlaid [27]. Unfortunately, Banaś [39] found that

`MotionEvent.FLAG_WINDOW_IS_OBSCURED` is not triggered if the covered text does not cover the touch points, which are the buttons in the dialog. Android does not give a patch to the issue. Instead, it adopts an intent transition scheme to notify the user that the content is being overlaid starting from Android 6.0 (API level 23). We managed to circumvent the issue by implementing our main attacking app with `targetSdkVersion=23`, and in the target app we tricked the user to install a helper package, which is an activity-less service. Only app which has the `targetSdkVersion=22` and has the overlaying functionality implemented. It seems tricking users to install additional package is not applicable, but it is in fact a very common situation for many apps such as those who require Android SQLite Manager.

The basic steps of the attack model are listed as follows.

(1) The attack follows the first three steps of UI Phishing attack.

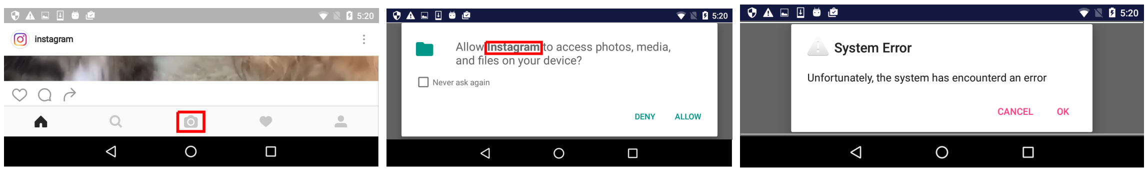
(2) Once a user presses the camera button of Instagram as shown in Figure 4(a), the foreground `Activity` will change to `"com.instagram.android.creation.activity.-MediaCaptureActivity"`. Therefore, Instead of detecting whether the foreground package has changed to another app, we zoom in the design to focus on changing of the foreground `Activity`.

(3) Instead of popping up a counterfeit page like UI Phishing does, it pops up an `Activity` which immediately asks a user for `READ_EXTERNAL_STORAGE` so that the user thinks he/she is granting the permission to Instagram.

(4) Once the attack app gets the permission, it traverses all the pictures and transmits the image buffers to server.

(5) To achieve better stealthiness, we introduce the Tapjacking to assist our attack. Its basic idea is to cover the real texts with some fake texts. Then we are able to change the text of the permission dialog to “Allow Instagram to access ...?” instead of the real text which is “Allow GalleryStealing to access ...?”, which is shown in Figure 4(b). Once user grants the permission, the app quickly sends all images as buffer to the server. Meanwhile, it fools the user by showing a “system warning dialog” telling the user that the system encounters

Figure 4: Gallery Stealing Attack



Allow Instagram to access photos, media, and files on your device?

System Error

Unfortunately, the system has encounterd an error

First the user clicks the camera button highlighted in the first picture. Then the attacking app kills the real Instagram and pops up its `Activity` asking for user permission, the text of permission dialog is overlaid by the fake text as shown in the second picture. Once user grants the permission, the app quickly sends all images as buffer to the server, meanwhile, it fools the user by showing a "system" warning dialog telling the user the system encounter an error.

an error (shown in Figure 4(c)).

3.3.4 Screen Shot Capturing

With the ability of knowing which `Activity` is currently running in the foreground, we implement this screen shot capturing attack which starts taking screen shots while a user is entering username and password in the Facebook `LoginActivity` which is `com.facebook.katana.LoginActivity` in full. For password typing, every time a character is entered, the character will be shown for a short time before it turns into a star sign. Therefore, taking screen shot every 0.1 second should capture everything a user types in password box.

With respect to taking a screen shot programmatically, we summarize four possible ways:

- **Using READ_FRAME_BUFFER Permission.** Declaring `READ_FRAME_BUFFER` permission in the `Manifest` allows an application to take screen shots by making calls to `ISurfaceComposer` [31]. However, this permission is not available to third-party application unless it has the same signature as the system does.

- **Using fb*.** Some Linux systems store frame buffers in `/dev/graphics/fb*` or `/dev/fb*`. `fb0` represents the first frame buffer, `fb1` represents the second frame buffer and so on. Using native C/C++ code to get access to these files and copy the buffer as a `GGLSurface` structure is theoretically possible. But there are two unsolvable obstacles of this method:

  – This method requires root permission.
  – It is likely that `fb*` does not even exist.

- **Using Backup Channel over USB.** Android system uses Android Debug Bridge (ADB) to listen to the debugging connections over USB [17]. ADB has slightly more privileges than normal apps. Bai *et al.* [4] manage to exploit Backup Channel through ADB to steal access tokens from other apps. Combining ADB with Dalvik Debug Monitor Server (DDMS) tool enables an app to get the screen shot from the device without any permission [23].

- **Using MediaProjection.** For devices beginning in Android 5.0 (API level 21), a class called `MediaProjection` was added to Android SDK which enables a third-party app to capture screen shot and record system audio [30]. While recording system audio requires `RECORD_AUDIO` permission, capturing screen shots does not.

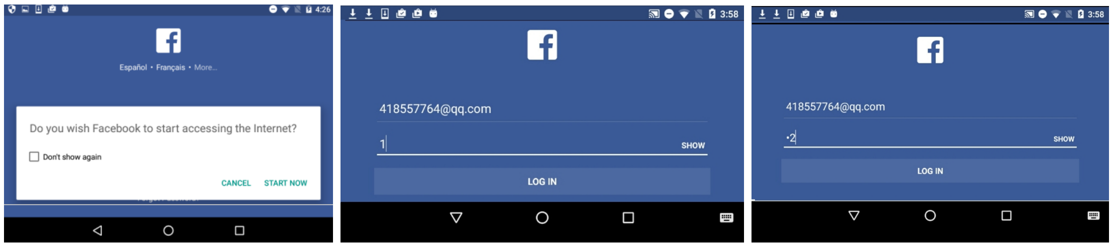
We employed the fourth method to capture screen shots. The basic steps of the attack are listed as follows.

(1) It follows the first three steps of UI Phishing. What's different is we devise this attack to focus on Facebook App since we are hoping to steal user's Facebook username and password. Hence, we declared this attack to reside in the same task as Facebook.

(2) Once a user launches the Facebook app, a transparent Activity is popped up start taking screenshots. Although it does not require permission for taking screen shots, it uses intent transition to let the user decide whether or not an app can capture screen shots in a permission-like dialog. Again we employ the tapjacking in Gallery Stealing attack to cover the text to be "Allow Facebook start accessing Internet?"

(3) The attack starts capturing screens by calling `startActivityForResult()` which

Figure 5: Screenshot Capturing Attack



(a) (b) (c)

First the user launches Facebook, then the attacking app launches a transparent `Activity` which quickly asks for taking screen shot, the notification dialog is again overlaid by fake text as shown in the first picture. Once the user clicks start now, the attacking app begins to take screen shots as well as sending the screen shots to back end server. (b) and (c) are two of the screen shots received by server which expose the password plaintext.

passes a screenshot as an intent from which we can extract an object of `MediaProjection` later, and finally use this `MediaProjection` object to pass the image to an object of `ImageReader` through its member function `createVirtualDisplay`. Since the screenshots are taken in the `RGBA_8888` format while bitmap takes `ARGB_8888`, we still need to do matrix transformations to get the image.

3.4 Attack Performance

In order to study the feasibility of the four attacks, we implement them and evaluate them in two aspects, *time cost* and *memory usage*.

3.4.1 Time Cost

We conduct 6 - 8 rounds of tests for every attack to calculate the average running time cost. The testing results are summarized in Table 3. As shown in Table 3, our attacks are efficient since the time cost of most attacks is less than 8 seconds: the UI Phishing attack costs 7.1 seconds, which includes user interactions such as launching Instagram, entering username and passwords; The Activity-in-the-middle attack costs time 5.8 seconds; and the average time cost of the gallery-stealing attack is 4.6 seconds.

The Screenshot Capturing attack is an exception, taking 38.1 seconds. This attack uses much longer time because screenshots in Android are passed in the format of RGBA, while we need to change to ARGB in order to convert it to common format such as JPEG and PNG. We perform the matrix transformation required for the conversion on the mobile device. But since the process is undertaken in the background, it will not trigger the suspicions of the user. In the real world attack, an adversary can leave the job of matrices transformation to the server.

Table 3: Time Cost of Proof-of-Concept Attacks

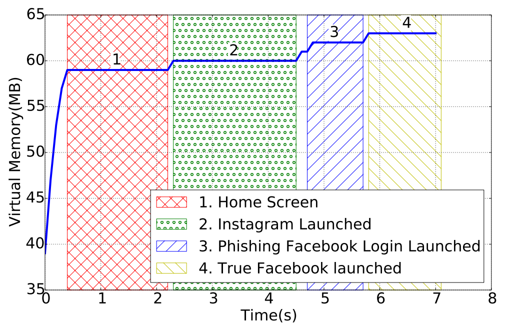
Attack	**Time Cost (s)**
UI Phishing	7.1
Activity-in-the-Middle	5.8
Gallery Stealing	4.6
Screenshot Capturing	38.1

3.4.2 Memory Usage

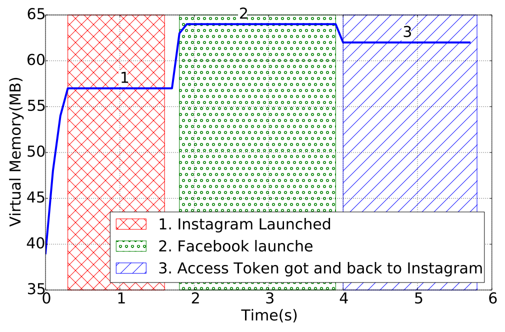
We conduct 3-round experiments for each attacks to get the memory distractions of attacks and evaluate their average memory usage. The testing results are illustrated in Figure 6. It shows that the maximum memory usage of most attacks is less than $70MB$ except Gallery Stealing attack, whose memory usage is around $100MB$. Our testing results also disclose the memory usage distribution on different period, in which the major memory usage of each attack is to launch victim app (normal app), e.g., Instagram and Facebook. The memory usage differences caused by stealthy behaviors/operations are negligible.

In addition, we also conduct experiment to monitor the battery consuming status and evaluate the battery usage of our attacks. The results show that our attacks may not cause influence on battery aspect (the battery usage rates of most attacks are 0%). Above all, the experiment results demonstrate that our proof-of-concept attacks are light-weight with limited permission requirements.

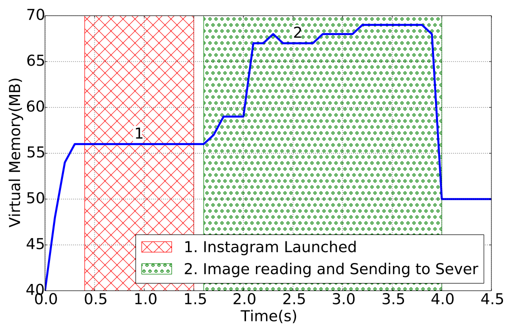
Figure 6: Memory Distribution Curves



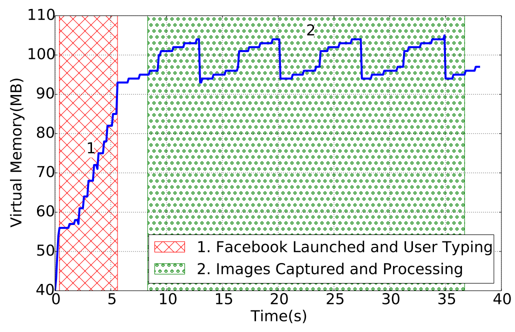
(a) UI Phishing



(b) Activity-in-the-middle



(c) Gallery Stealing



(d) Screenshot Capturing

3.5 Solutions to Eliminate Task Interference

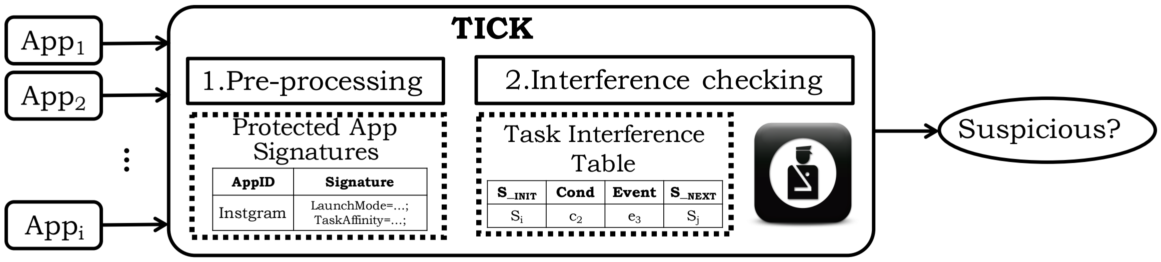
In this subsection, we discuss solutions on mitigating Android task interference. Based on our study, we design and implement a task interference checking tool to detect the potential risk of task interference among apps. It protects apps specified by users from being manipulated by untrustworthy apps. In addition, we also propose some suggestions to limit the additional privilege achieved by the apps in the same task.

3.5.1 Task Interference Checking

In this subsection, we present the design and implementation of our *Task Interference Checking* app named *TICK*.

Based on our understanding of the task interference problem, which includes the necessary conditions, events and their dependency, we design a method to check the task inter-

Figure 7: Task Interference Checking Architecture



ference status among Android apps. Table 2 provides us with the basis of evaluating task interference checking.

Design. As shown in Figure 7, *TICK* consists of two basic modules, *Pre-processing* and *Interference Checking*, and two supporting databases, *Protected App Signature* and *Task Interference Table.*

- *Protected App Signature.* For the apps users who want to protect against attacks to Android tasks, this database includes the abstracted conditions of such attacks of each app, which is stored as signatures in this database.

- *Task Interference Table.* This database is the output from our research in Section 3.2, which includes the fundamental rules of task interference.

- *Pre-processing.* The module takes as inputs the manifest files of apps to be checked, abstracts and outputs their task interference features for checking.

- *Interference Checking.* The interference checking module takes inputs from *Pre-processing* module and *Protected App Signature* database, and checks the suspiciousness of the testing apps according to the rules specified in the Task Interference Table.

There are two application scenarios for deploying *TICK.*

C-1: Before a user installs an app, she/he can use TICK to detect the potential risk of task-related attacks from the app. TICK will parse the app meta data and check with our table and signature database to see if it interferes with an existing app. If the app is detected to cause interference, TICK will issue a warning to the user and suggest she/he not install the app.

C-2: Our checking app can do chronically scan from time to time, check if new packages are added to the device, audit if they have security concerns, and notify users if and suspicious package were installed.

Getting Meta Data of an Activity. Currently, our checking app only considers static declarations of task related attributes from the app package. Moreover, the key attributes related to Android Task cannot be altered during runtime, e.g., *taskAffinity* and *allowTaskReparenting*.

Android SDK provides standard APIs to get an app's Manifest meta data without requiring any permission. Given a package name and a flag, one can retrieve an object of `PackageInfo` through the `PackageManager`. Here, we set the flag to `GET_ACTIVITIES` and we retrieve a list of `ActivityInfo` by fetching the attribute `PackageInfo.activities`. ActivityInfo contains all the meta data described in the `Manifest` about every activity of a package, including those we care, e.g., `taskAffinity`, `FLAG_ALLOW_TASK_REPARENTING`, `LAUNCH_SINGLE_TASK` and so on. The key APIs of getting `packageManager` and `packageInfo` are respectively `getPackageManager()` and `getPackageInfo()`.

Effectiveness. To verify the effectiveness, we used Instagram and our UI-Phishing app as inputs to TICK. It parsed the meta data of each activity in both apps and found that most activities of Instagram with *taskAffinity* set to "com.instagram.android", while the rest has *taskAffinity* of "com.instagram.android.ShareHandlerActivity". Most of the activities in Instagram have *launchMode* set to "standard". A small portion of the activities have the *launchMode* set to "singleTop", e.g., the `LoginActivity`. Very few activities have *launchMode* set to "singleInstance". The UI-Phishing app's first activity has the following flags set: `taskAffinity = "com.instagram.android"`, `allowTaskReparenting = true` and `launchMode = standard`. This matches

Case 9 and Case 10 in our task interference table. When we use TICK to scan the device, specifying the Instagram app as the one to be protected, TICK successfully warns users about the potential risk from the UI-Phishing app. The overhead of TICK can vary based on a number of factors such as device hardware and number of third-party Apps installed in the device. In our experiment, excluding all system-level Apps, there are totally 65 third-party Apps installed in our device. It rougly took 4-5 seconds to scan all Apps and identify suspicious ones.

3.5.2 Design Suggestions

From security issues we have demonstrated in the above attacks, it is clear that the privilege given to apps in the same task is well beyond what is expected for a mechanism that facilitates app collaboration and interaction. In fact, the task mechanism should be treated as a way of *authorization*, and the security mechanism around the task mechanism also should be designed accordingly.

In particular, when treating a task as a boundary for authorization, we need to be explicit about the ownership of a task and its authenticity. For example, if an app specifies the *taskAffinity* of an existing task, there needs to be a form of *authorization* before the app can be included into the task, and the authorization should be carried out by entities with privilege greater than the privilege given to the task. This is similar to the requirement made by the UNIX group mechanism. In addition, as the name of "task affinity" becomes an identifier for a security object, the system should avoid name conflicts. In case they occur, they need to be resolved with all involving entities to avoid unexpected privilege escalation.

3.6 Related Works

IPC Security: IPC security is one of the top concerns while designing OSes. Some early studies have reported security threats in the Android IPC mechanism. More specifically, Ren et al. [56] have proposed the first study of the security of Android task mechanisms and showed the possibilities of several enabled attacks, such as back-button hijacking and uninstalling-prevention attack. Other popular mobile systems like iOS are also not immune to the risks. Xing et al. [73] registered a counterfeit scheme which hijacks the real Facebook scheme in iOS and successfully stole a Facebook access token that was supposed to be passed onto the relying-party app. Besides problematic designs of IPC of OSes, mis-implementation of certain IPC-based protocols can also lead to security concerns. Chen et al. [6] have conducted an analysis on 149 mobile applications and showed 89 of them (59.7%) incorrectly implemented OAuth and thus are vulnerable to SSO-oriented attacks. Furthermore, IPC vulnerabilities were also documented on other platforms besides mobile OSes. Take browser platform for an example. Wang et al. [67] discovered 8 serious logic flaws among the traffics between high-profile ID provider and relying website through browser platform. Wang et al. [68] discovered logic flaws in several shopping websites and finally purchasing goods without or with little payment.

GUI Security: As for traditional desktop and browser environment, GUI security issues have been studied extensively [13, 57]. Niemietz et al [52] implement a UI redressing attacks on Android devices base on clickingjacking and tapjacking and the attack is feasible to be transferred from desktop to mobile and to browser, enabling the attack to be adapted to multiple platforms and functionalities. As mobile market begins to thrive, GUI security is more concerned in mobile platforms than ever before. Chen et al. [8] managed to impose Hidden Markov Model (HMM) on a public resource `shared_vm` combining a bundle of data to perform UI Inference Attack and successfully stole sensitive information from users such as user names, passwords and check images. Wang et al. [69] implemented a malicious app which circumvented the Apple Code Review system and successfully stole user secrets stealthily.

Defending against Malicious Behaviors: Defending malicious behaviors can be categorized into two branches, detection and prevention. In previous studies, various detection schemes have been introduced to prevent GUI-related attacks. Fu et al. [15] employ the Earth mover's distance (EMD) mechanism to detect possible malicious web page through measuring the similarity between two web pages by first converting web pages to images, and then grabbing and comparing the feature points through training data set. More generally, Chen et al [7] introduce the concept of permission event graph (PEG) with model checking mechanisms to detect abnormal behaviours of Android apps. As for prevention, one idea is to prevent sensitive data from being leaked to the malicious server party. Hornyack et al. [38] develop a system for Android called AppFence, which can block the sensitive data from being transmitted, or substitute the fine-grained data to coarse-grained data if transmission is unpreventable. Ren et al. [55] develop WindowGuard which protect against GUI attacks by enforcing the Android Window Integrity (AWI).

A Novel Attack Architecture for Device-Identity Association based on Android Side-Channels

4.1 Threat Model and Approach Overview

In this subsection, we present the threat model and the overview of our attack approach.

4.1.1 Threat Model

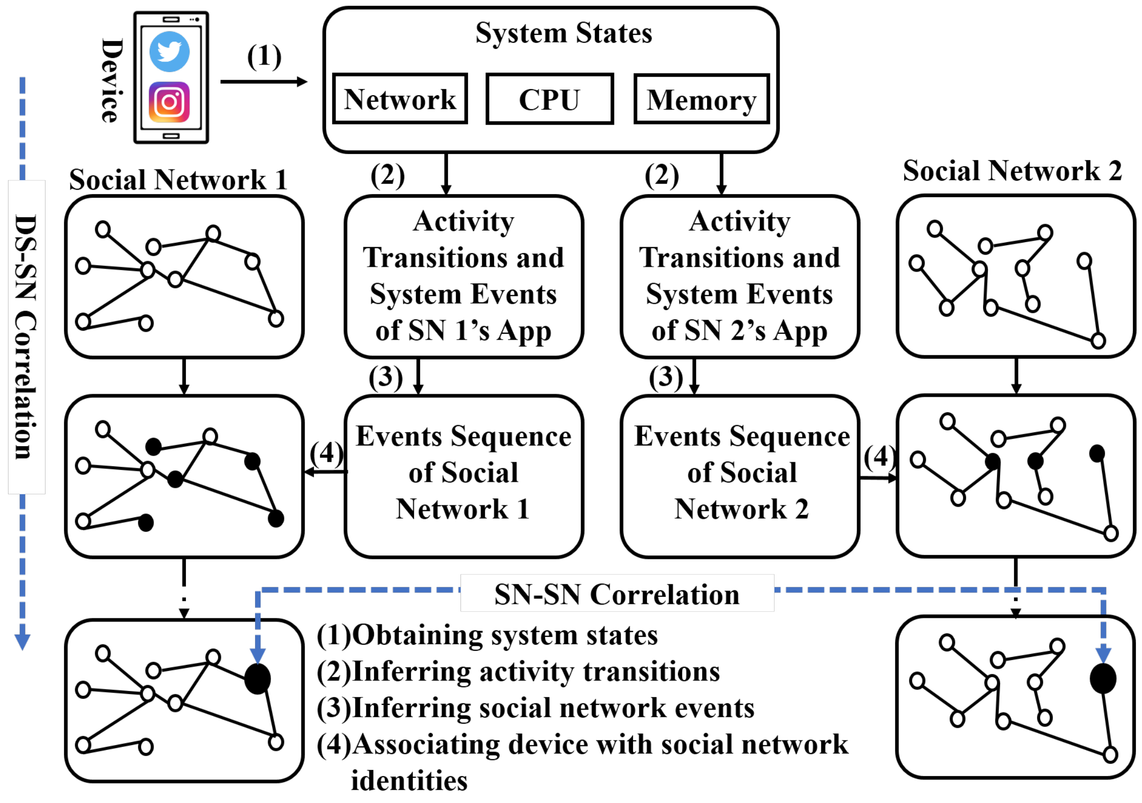
We consider an attacker who can accomplish inference attacks through a malicious app installed on a victim's device with no special permission – this malicious app only requests the `INTERNET` permission from Android to send the collected data to the attacker for analysis, which by default is automatically granted by the Android system without notifying the user. How to install a malicious app in a victim's device is out of the scope of this work. Here are a few viable approaches: an attacker can trick a user to install a malicious app by simply disguising the app as a benign one and putting it on the Google Play Store since current automatic malware detection for Android is still not sufficiently reliable [36]; alternatively, an attacker can secretly install a malicious app on a victim's device by exploiting a remote code execution (RCE) vulnerability such as CVE-2017-0561, which allows RCE on Wi-Fi SoC [54]. In this work, we assume that a malicious app has already been installed on the victim's device. This malicious app can silently collect and send to the attacker certain system states that are publicly available.

4.1.2 Approach Overview

The objective of our design is to associate an Android device (running social network apps to perform social activities such as posting a photo or a message) with the social network accounts (the accounts accessed by the device) based on the following publicly available information: (i) the system state information (see Section 1.2) collected by a malicious app installed in the victim Android device; and (ii) the social network events crawled from the social network databases. We consider three popular social networks: Twitter, Flickr, and

Instagram, and exploit the tweeting events in Twitter and photo-posting events in Flickr and Instagram for our studies on device-identity association.

Figure 8: Attack Architecture


Device
(1)
System States
Network
CPU
Memory
Social Network 1
(2)
(2)
Social Network 2
DS-SN Correlation
Activity Transitions and System Events of SN 1's App
Activity Transitions and System Events of SN 2's App
(3)
(3)
Events Sequence of Social Network 1
Events Sequence of Social Network 2
(4)
(4)
SN-SN Correlation
(1)Obtaining system states
(2)Inferring activity transitions
(3)Inferring social network events
(4)Associating device with social network identities

Our attack architecture is shown in Figure 8. It is composed of two attack vectors: *device-social network correlation (DS-SN)* attack and *social network-social network correlation (SN-SN)* attack. The objective of the DS-SN attack is to identify a list of candidate accounts that might have been accessed via the device for a target social network. We developed an approach to infer a user's social network events from his device system states. More specifically, we leveraged the system states collected by a zero-permission malicious app installed in the user's device to identify the events triggered by the social network app, e.g., the time to tweet and the size of the tweet in Twitter, or the time of posting a photo in Flickr/Instagram. By correlating such information with the public events in the target social network, one can identify a list of candidate accounts in the social network for the user who may have used the device to access his account. To uniquely identify the social network account accessed via the device, one can observe multiple social network events triggered by the device to shorten the candidate account list, which may take a long time if the user does not frequently access his account via the device. If the device is used to access multiple accounts of the same user belonging to different social networks, the attacker can employ the SN-SN attack that examines the profile similarity between two social network accounts from two different social networks to identify the most possible account for the user in each social network. More specifically, the attacker first obtains the candidate account lists for two (or more) social networks via the DS-SN attack, then calculate the profile similarity of two accounts, with one from each account list, and identify the pair of accounts with the highest similarity. Such a SN-SN attack can not only speedup the process of device-identity association attack, but also help identify two or more social network accounts accessed via the same device for the victim.

4.2 Design and Implementation of Our Attacks

In this subsection, we detail our design of the two inference attacks: DS-SN attack and SN-SN attack.

Table 4: Summary of the DS-SN Attack

	Inferred SN Events	Corresponding Activity Transition	Corresponding System States
Twitter	Tweeting Timestamp	`MainActivity` $\Downarrow$ `ComposerActivity`	*VSS* of Twitter `tcp_snd` of Twitter `tcp_rcv` of Twitter
	Number of Characters of a Tweet	N/A	`utime` of Keyboard `stime` of Keyboard
Instagram	Photo-posting Timestamp	`MediaCaptureActivity I` $\Downarrow$ `MediaCaptureActivity II`	*VSS* of Instagram `tcp_snd` of Instagram `tcp_rcv` of Instagram
Flickr	Photo-posting Timestamp	`MainActivity` $\Downarrow$ `FilterUploadActivity`	*VSS* of Flickr *RSS* of Media Process `tcp_snd` of Flickr `tcp_rcv` of Flickr

4.2.1 DS-SN Correlation Attack

Our approach to attacking the DS-SN correlation is composed of 4 steps: 1) obtaining system states from a victim's device via a zero-permission malicious app, 2) inferring activity transitions in the device, 3) inferring the corresponding social network events triggered by the device, and 4) associating the victim's device with his social network account based on the inferred events triggered by the device and the publicly available events crawled from the target social network. We considered three popular social networks: Twitter, Flickr, and Instagram. Our purpose is to infer the tweeting time of and the number of characters in a tweet for Twitter, the posting time of a photo in Instagram, and the posting time of a photo in Flickr. This inference is based solely on the publicly available information: the Android system states collected by a zero-permission app and the tweeting/photo-posting events crawled from the social network databases. The device state information needed by the DS-SN attack as well as the inferred device activity transitions and social network events for the three social networks are presented in Table 4.

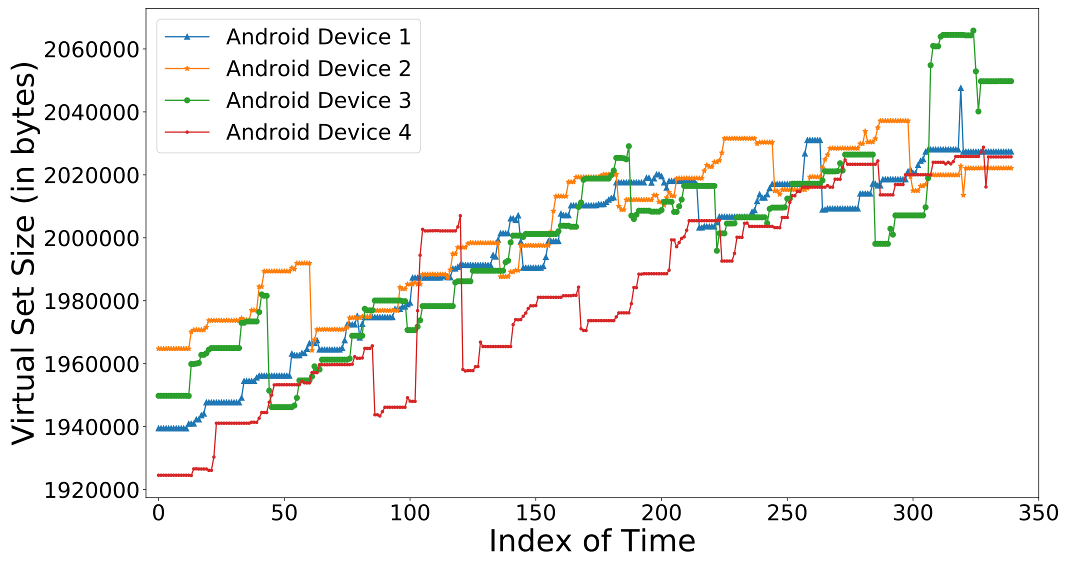
Obtaining System States.

As shown in Table 4, we exploited *VSS*, *RSS*, `utime`, `stime`, `tcp_snd`, and `tcp_rcv` for our DS-SN attack. There are multiple ways to obtain these system data in Android, as elaborated in Section 1.2. The most intuitive way is to make calls to the Android APIs. However, one has to retrieve the three types of data using different APIs, making this method less efficient. To overcome this problem, we first leveraged the "`ps`" command with Android Toolbox [10], which can return the memory and CPU data in one call; then we directly read the system files `/proc/uid_stat/pid/tcp_snd` and `/proc/uid_stat/pid/tcp_rcv` to retrieve the network data.

Inference of Activity Transitions

Activity transition is one of the most critical states to infer private information from an app. Previous research done by Chen *et al.* [9] described an approach that can infer an app's activity transition using a Hidden Markov model (HMM) over the memory data

Figure 9: The Variations of *VSS* of the Twitter App

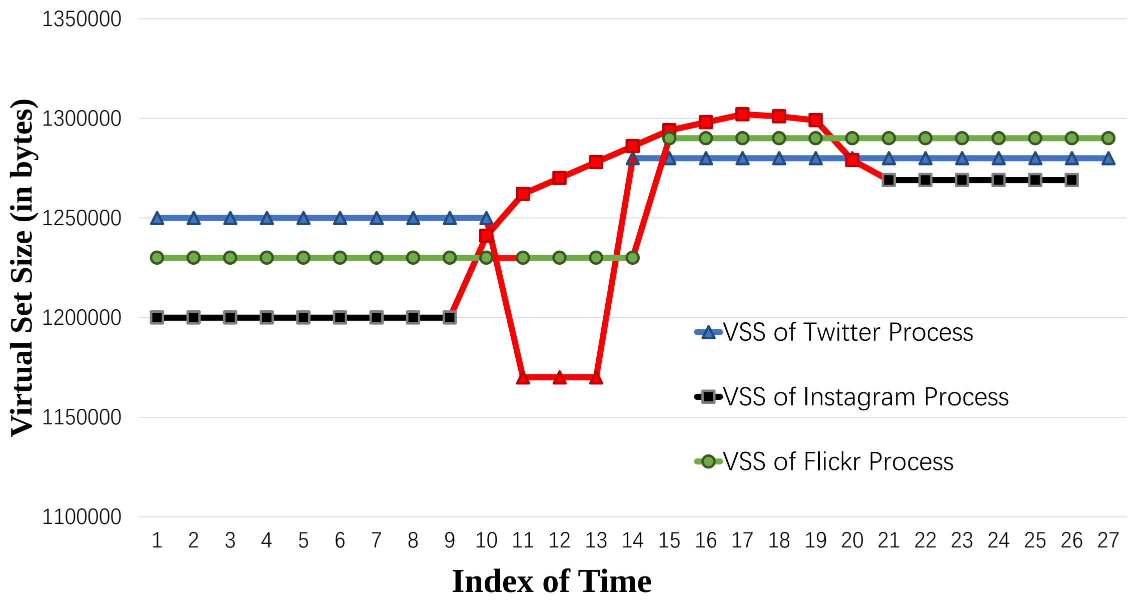


The variations of *VSS* of the Twitter App in four different Android devices within approximately 2 minutes of swiping and tweeting.

and then further infer the current foreground UI/activity the user is landing on. We tested this approach in our Android devices but failed to make it work. A careful study indicates that this approach is quite sensitive to the quality of the memory data but social network apps when running constantly incur significant noises compared with the clean acctivity transitions considered in [9]. Figure 9 illustrates the variations of the *VSS* of the Twitter app in four Android devices with different OSes when we performed swiping and tweeting for approximately two minutes. Note that people usually swipe when tweet; thus these two activities are commonly performed together. From this study one can conclude that social network apps incur noisy memory data in most Android OSes, and the HMM inference method would face challenges when distinguishing tweeting from swiping. Having noticed this fact, we propose our own approach to precisely infer the activity transitions with the noisy memory data.

To proceed, we take a look at the VSS changes of the three social networks (see Figure 10) when only a posting event is performed - no other activities such as swiping and tapping is involved to remove noise. One can see from Figure 10 that the activity transition of tweeting (from `MainActivity` to `ComposerActivity`) takes 4 time intervals[1]; the one for photo-posting in Instagram (from (`MediaCaptureActivity I` to `MediaCaptureActivity II`) takes 12 time intervals; while the one for photo-posting in Flickr (from `MainActivity` to `FilterUploadActivity`) takes only one time interval. Thus obviously, it is hard to design a uniform approach that can work for all the three social networks. In this subsection, we propose a novel *memory pattern regression* model that can infer the activity transitions based on the VSS variations for Twitter and Instagram, and employ the RSS of the system media process[2] to infer the activity changes in Flickr. In the following, we first detail our novel memory pattern regression model.

Figure 10: The Variations of *VSS* of Twitter, Instagram, and Flickr



The variations of *VSS* of Twitter, Instagram, and Flickr when a tweet or a photo is posted. The red parts refer to the activity transitions of tweeting/photo-posting in the corresponding social network.

Memory pattern regression. Our memory pattern regression algorithm takes as input the training set $D = \{\mathbf{y}_j\}_{j=1\cdots d}$, which is the memory data tested d times for a single activity

[1]Here we refer a time interval as the time span needed for the retrieval of one record of the system states, which can differ from device to device but normally it is within 0.5$\sim$0.6 seconds

[2]We employed RSS for Instagram too but our memory pattern regression model performs better.

transition with $\mathbf{y}_j = (y_{j_1}, y_{j_2}, \cdots, y_{j_{N_j}})$ being the *VSS* vector of size N_j obtained in the j-th experiment. We first need to find out the number n, which represents the number of time intervals in which a target activity transition, i.e., an activity transition of a tweeting in Twitter or a photo-posting in Instagram, takes place. In order to determine n, we first construct a sequence $\mathbf{\Delta y_j} = (\Delta y_{j_1}, \cdots, \Delta y_{j_{N_j - 1}})$, where $\Delta y_{j_i} = |y_{j_{i+1}} - y_{j_i}|$; then we set up a threshold δ, which can be determined experimentally, and let n_j be the number of terms in $\mathbf{\Delta y_j}$ whose values are greater than δ; finally we set $n = \lceil \frac{\sum_{j=1}^{d} n_j}{d} \rceil$.

Having determined n, we proceed to extract the *VSS* change pattern of a transition. Instead of directly using the raw data of *VSS*, we use slopes of each pair of adjacent *VSS* points since doing this can not only mitigate the noises of memory but also obtain the shape of the transition (i.e., pattern). In order to do so, we minimize a function $f : \mathbb{R}^{n+1} \longrightarrow \mathbb{R}^{n+1}$, which is represented as

$$f \leftarrow \arg\min_{f} \{ \frac{1}{d} \Sigma_{j=1}^{d} [\frac{1}{n} \Sigma_{i=1}^{n} (\Delta f(j_i) - \Delta y'_{j_i})^2] \} \tag{1}$$

where $\Delta f(j_i) = f(j_{i+1}) - f(j_i)$ is the slope between the point j_{i+1} and j_i since $j_{i+1} - j_i = 1$, and $\Delta y'_{j_i}$ is the ith term in the consecutive interval of $\mathbf{\Delta y_j}$ with size n that covers the maximum number of values greater than δ.

In order to learn the f satisfying (1), we employ a 5-layer feedforward neural network with n neurons at each layer. In this deep neural network, each neuron in the input and output layers is connected to a neuron (one-to-one) in the adjacent hidden layers, but all the hidden layers are fully connected. Moreover, the input and output layers simply use the Hadamard product as the activation function while all other layers use the Sinusoid function for activation.

The neural network adopted in our approach was implemented based on the Google TensorFlow [1] deep learning framework with the Adam stochastic optimization [41] that takes the right part of (1) as a loss function. By minimizing the loss function, we can obtain $\mathbf{v_j} = (\Delta f(j_1), \Delta f(j_2), ..., \Delta f(j_n))$ as the *VSS* pattern, which is related to an activity transition with n pieces of slopes.

Let $\mathbf{y} = (y_1, ..., y_N)$, with $N > n$, be the *VSS* sequence collected from a victim's device. The question here is: how can we tell whether the target activity transition happens in this sequence based on the pattern $\mathbf{v_j}$? Intuitively, directly determining whether the Euclidean distance between them is smaller than a threshold σ, i.e., $\|\mathbf{v_j} - \Delta\mathbf{y}\| < \sigma$ with $\Delta\mathbf{y}$ being n consecutive slops of $\mathbf{y}$, seems reasonable. However, the Euclidean distance is not applicable in our case since if a user constantly uses an app without quitting, the *VSS* of the App would accumulate. Thus clearly, statically setting a threshold can result in a gradually larger error. Therefore, we developed a comparison method which fits our problem well. Instead of directly considering $\Delta\mathbf{y}$, we tolerate a little bit by considering a neighborhood of $\Delta\mathbf{y}$, denoted by $B_r(\Delta\mathbf{y})$, where r is a small integer such that $n < r < N$. Specifically, for a *VSS* value at i, we consider the previous and post r slopes of this value, yielding $B_r(\Delta y_i) = (\Delta y_{i-r}, ..., y_i, ..., \Delta y_{i+r})$ (totally $2r + 1$ terms). After that, given a threshold σ, our comparison algorithm extracts each continuous segment of slopes with size n from $B_r(\Delta\mathbf{y_j})$ and calculates the mean squared errors with $\mathbf{v_j}$, denoted as $mse_1, mse_2, ..., mse_{2r+2-n}$ (totally $2r + 2 - n$ terms). Lastly, if $\max(|mse_1 - mse_2|, |mse_2 - mse_3|, ..., |mse_{2r+2-n} - mse_{2r+2-n}|) < \sigma$, we perceive that the target activity transition occurs in $B_r(\Delta y_i)$.

For the activity transition inference of the photo-posting event in Flickr, we employed a different approach based on the following observation: when posting a photo in Flickr, a user needs to choose a picture from his album or to take a photo; in this case, the *RSS* of the system media process, i.e., `android.process.media`, increases and the *VSS* of Flickr increases. Therefore, using this information, one can infer that an activity transition of a photo-posting event occurs in Flickr. Note that even though Instagram is also a photo-based social network, this method does not apply to it because Instagram allows a user to post comments with a picture. Therefore, if directly using this method in Instagram, one may not be able to tell if a user posts a photo or just sends a comment.

Inference of Social Network Events

To infer the timestamp of a tweeting/photo-posting event, we need to check not only the corresponding activity transitions implied by the memory state changes but also the network states of the device: the existence of an activity transition alone does not mean that the event actually has happened - it is completed only after the tweet message or the photo is sent via the network. Therefore, we need to figure out whether `tcp_snd` and `tcp_rcv` increase after the detection of the activity transition[3]. In other words, an attacker can infer when a tweeting/photo-posting event happens by combining activity transitions with the network `tcp_snd` and `tcp_rcv` information.

In Twitter, we can also infer the number of characters in a tweet message by the keyboard event. Note that after a tweet is posted, keyboard would disappear and `tcp_snd` would increase. However, we cannot use `tcp_snd` to precisely estimate the number of characters sent in a tweet due to the protocol overhead of TCP connections. In our approach, we resort to CPU usage information. We found that the system states of the Android keyboard process (`com.google.android.inputmethod.latin`) are tightly related to a user's typing actions. More specifically, when the keyboard is launched, the *VSS* of the keyboard process increases drastically from a constant state; when a user types a keystroke, the `utime` and the `stime` of the keyboard process roughly increase by 1 clock tick. Thus we took the ceiling of the averaged increased clock ticks of `utime` and `stime` to estimate the number of characters in a tweet message.

Associating Device with Social Network Accounts

After retrieving the targeted social network events based on the Android system states, i.e., the timestamps of the posts and the sizes of the tweets, we can repeatedly match this information with the public information of the target social network to obtain a gradually smaller list of potential social network accounts for the device. For example, from the

[3]One can use only `tcp_snd` for the event inference but our experiments indicate that both `tcp_snd` and `tcp_rcv` increase when an event occurs, which is reasonable as TCP connections involve two-way traffics.

device system states we inferred that our target victim tweeted twice in two different timestamps and estimated the sizes of these two tweets. Then we proceeded to crawl the Twitter database and filter out those who did not tweet in these two timestamps and whose tweet sizes do not match with our inferred tweet sizes. By this way we can get a reasonably small list of potential Twitter accounts that may be associated with the victim with a high probability.

Crawling the Social Network Events

As mentioned earlier, we need the public information of the target social network to realize the DS-SN attack. More specifically, we need to crawl the tweets/posts occurring at a specific time within a social network. However, it is not trivial to retrieve the required data from the three social networks under our consideration. We have to overcome the following challenges.

First, even though Twitter has a streaming API which provides an interface to facilitate the retrieval of the tweets for a given time period, this method does not work since the API tends to drop a large amount of unimportant data and it has a rate limit. Luckily, we found that Twitter offers a webpage called Twitter Advanced Search (TAS) via which we can retrieve a list of tweets that contain the queried keywords, locations, or user accounts, for a given time period. Since user accounts are what we want to retrieve, we cannot use accounts as a filter. Therefore, we query with letters from “a” to “z” as the filtering keywords, and the TAS server replies a list of tweets containing any letter from “a” to “z” with very minimal data loss - the Twitter server fails to return some tweets only when it thinks that these tweets are duplicates or these tweets only contain non-English contents. Hence, we leverage this method by making queries through TAS with 26 letters and set the time period to be 1 minute. We wrote a program that sends a url in the following format:

```
https://twitter.com/i/search/timeline?f=tweets&vertical=default&q= keyword%20since%3
    Astart_date%20until%3Aend_date&src=typd&include_available_features=1&include_entities
    =1&lang=en&max_position=TWEET-old_tweet_id-new_tweet_id-
    BD1UO2FFu9QAAAAAAAAETAAAAAcAAAASAAAA...A&reset_error_state=false
```

where *new_tweet_id* is the tweet id of the very first tweet shown in the result, while *old_tweet_id* is the id of the last tweet received. Then TAS returns a JSON file containing 20 tweets for each query sorted by the posting times. Note that this method requires us to recurrently query until we retrieve all the tweets for our desired time period. We found that in average there are approximately 30,000 - 60,000 new tweets generated globally at every minute.

Flickr offers a similar search engine that does not require the inputs of keywords, locations, and so on. In other words, by merely feeding a time period to the Flickr search engine one can retrieve all the posts of that time period. Based on our observation, Flickr follows the following format to retrieve posts:

```
https://api.flickr.com/services/rest?sort=relevance&parse_tags=1&
    content_type=7&extras=can_comment%2Ccount_comments%2Ccount_faves%2'
    Cdescription%2Cisfavorite%2Clicense%2Cmedia%2Cneeds_interstitial%2
    Cowner_name%2Cpath_alias%2Crealname%2Crotation%2Curl_c%2Curl_l%2
    Curl_m%2Curl_n%2Curl_q%2Curl_s%2Curl_sq%2Curl_t%2Curl_z&per_page
    =100&page=xx&lang=en-US&advanced=1&min_upload_date=start_time&
    max_upload_date=end_time&viewerNSID=xx&method=flickr.photos.search&'
    csrf=xx&api_key=xx6&format=json&hermes=1&'hermesClient=1&reqId=xx&
    nojsoncallback=1'
```

where *start_time* and *end_time* are Unix times. The server returns a JSON file containing 100 posts for each requested page number. Here a page number represents the index of the Flickr post pages, with each page containing 100 posts. For example, the third page (page number three) contains the 301st to the 400th posts following a descending posting time. We wrote a program that keeps on sending the url until all pages have been returned.

In contrast, Instagram does not have any interface for returning posts based on time; instead, it provides an interface that returns posts based on locations, accounts, and hashtags. Therefore, we can only collect as many user accounts as possible for further scrutiny. Due to the nature of Instagram, almost every Instagram user follows at least one verified account (celebrity). Thus we collected all the followers of the official account of

Instagram, Selena Gomez, and Taylor Swift, the top three most-followed accounts on Instagram. As a result, we obtained in total 390,592,786 users while the number of active Instagram users is about 500 million to 600 million [62]. To collect the followers of the three most-followed accounts, we wrote a program that sends a post data to the url https://www.instagram.com/query/ with the format of

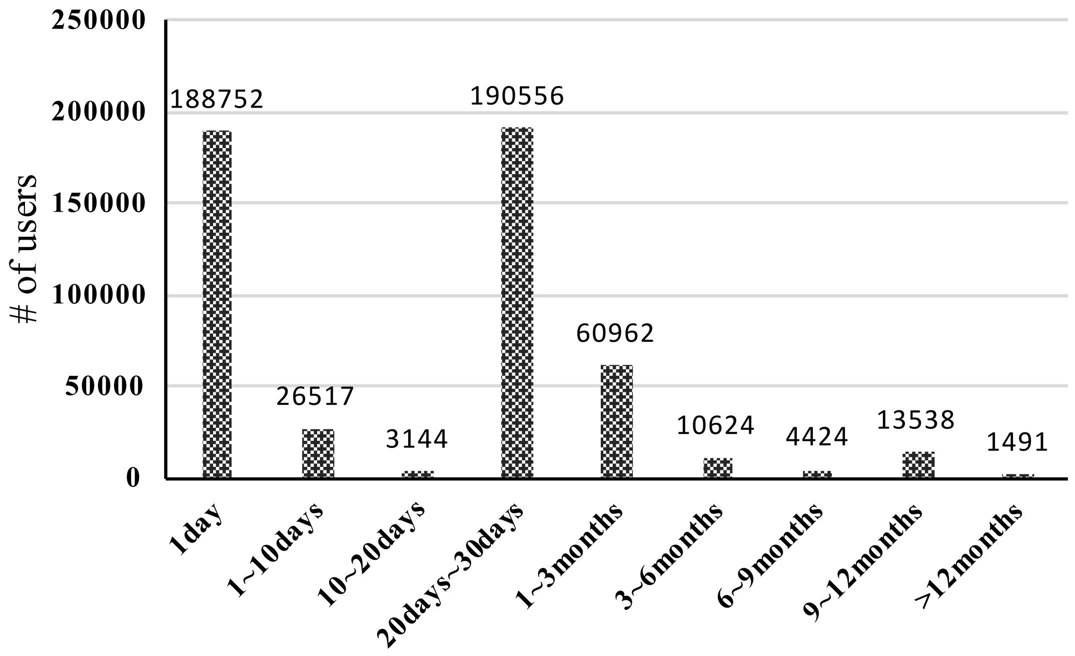
```
"q": "ig_user(xxx)
{ followed_by.after(end_cursor, step)
{count,page_info
{ end_cursor, has_next_page}, nodes {id, is_verified,
followed_by_viewer, requested_by_viewer, full_name, profile_pic_url, username}
}
}"
```

where *end_cursor* indicates the cursor of the last retrieved user and *step* indicates the number of users returned at each round.

4.2.2 SN-SN Correlation Attack

Our SN-SN correlation attack is designed to assist the DS-SN attack since the latter may not always be feasible (e.g., taking too long time) due to the infrequency of successive tweets/posts in a single social network by the target victim. To understand this challenge, we carried out a measurement study over 500,008 users randomly selected from the followers of 10 celebrities in Twitter to collect the time intervals between the most recent two successive tweets. As one can see from Figure 11, over half of these Twitter users tweeted again more than 20 days later after their last tweet. Therefore, if only exploiting the DS-SN correlation, an attacker may have to wait for a long time to collect enough data for a satisfying result. Nevertheless, a user may access different social networks with the same device at different times; thus exploiting the correlations of the accounts in different social networks may help speedup and enhance the accuracy of the device-identity association attack. In this subsection, we investigate the SN-SN correlation to identify the possible accounts in different social networks for the same user, which can help shorten the list of candidate accounts obtained from the DS-SN attack. Assume that an attacker has launched the DS-SN attack on a victim device for two social networks and obtained two lists of candidate accounts. Then the objective of our SN-SN attack is to identify the accounts that can match to the same user with high probability. This is done by the so-called *profile similarity*, which is a weighted average of the similarities of five attributes: *location* (s_1), *personal website link* (s_2), *username* (s_3), *biography* (s_4), and *profile image* (s_5). A novel learning-based model is presented to calculate the weight for each attribute.

Figure 11: Real-World Tweeting Situation



The number of users *vs.* the time difference between two successive tweets. One can see that the majority of the Twitter users tweeted twice between 20 days to 30 days.

In the following we present the definitions of the five attribute similarities. For better elaboration, we use $u_1^{t_1}$ and $u_2^{t_2}$ to refer to the two accounts in social network t_1 and t_2, respectively.

Definition 1 (Location Similarity, s_1)**.** *Let* $p_1^{t_1}$ *and* $p_2^{t_2}$ *be the locations set by* $u_1^{t_1}$ *and* $u_2^{t_2}$*,* *respectively. The location similarity score* s_1 *is defined as*

$$s_1(p_1^{t_1}, p_2^{t_2}) = \begin{cases} 1 & if\ p_1^{t_1} \text{ and } p_2^{t_2} \text{ are the same}, \\ 0 & otherwise. \end{cases} \tag{2}$$

In the social networks we consider, the location in a user's profile follows the format of "*City, State/Province, Country*" (a user is not mandatory to fill out all these three fields); a user can also upload the longitude and latitude of his location. In this study, we consider two locations to be the same if (1) cities are the same; or (2) states/provinces are the same; or (3) the Euclidean distance between two coordinates does not exceed 814 miles, which is the approximate diameter of the largest state, Alaska, in the US [70].

Definition 2 (Personal Website Link Similarity, s_2)**.** *Let* $p_1^{t_1}$ *and* $p_2^{t_2}$ *be the personal website links set by* $u_1^{t_1}$ *and* $u_2^{t_2}$*, respectively. The personal website link similarity score* s_2 *is defined as*

$$s_2(p_1^{t_1}, p_2^{t_2}) = \begin{cases} 1 & if\ p_1^{t_1} = p_2^{t_2}, \\ 0 & otherwise. \end{cases} \tag{3}$$

Definition 3 (Username Similarity, s_3)**.** *Let* $p_1^{t_1}$ *and* $p_2^{t_2}$ *be the usernames of* $u_1^{t_1}$ *and* $u_2^{t_2}$*, respectively. The username similarity score* s_3 *is defined as*

$$s_3(p_1^{t_1}, p_2^{t_2}) = \frac{|lcs(p_1^{t_1}, p_2^{t_2})|}{\max(|p_1^{t_1}|, |p_2^{t_2}|)} \tag{4}$$

where $lcs(str_1, str_2)$ *denotes the* longest common substring *of strings* str_1 *and* str_2.

According to the latent semantic analysis (LSA) in NLP [43], we defined the biography similarity in the following manner.

Definition 4 (Biography Similarity, s_4)**.** *Let $p_1^{t_1}$ and $p_2^{t_2}$ be the biographies of $u_1^{t_1}$ and $u_2^{t_2}$, respectively, where $p_1^{t_1}$ contains m_1 unique words and $p_2^{t_2}$ contains m_2 unique words. The frequency vector $\overrightarrow{fq}(p_1^{t_1,4}, p_2^{t_2})$ is defined as the number of times each word of $p_1^{t_1}$ appears in $p_2^{t_2}$, i.e.,*

$$\overrightarrow{fq}(p_1^{t_1}, p_2^{t_2}) = (z_1, z_2, \cdots, z_{m_1})^T$$

with z_i being the number of times the ith word of $p_1^{t_1}$ appearing in $p_2^{t_2}$. Let $F = \overrightarrow{fq}(p_1^{t_1}, p_2^{t_2}) \cdot \overrightarrow{fq}(p_1^{t_1}, p_2^{t_2})^T$. We perform the SVD decomposition against F and retrieve F's singular value vector, which is denoted by $\delta = (\delta_1, \ldots, \delta_{m_1})$. Then the biography similarity $s_4(p_1^{t_1}, p_2^{t_2})$ is defined as

$$s_4(p_1^{t_1}, p_2^{t_2}) = \min(\frac{\|\Sigma\|_F}{|p_1^{t_1}|\,|p_2^{t_2}|}, 1) = \min(\frac{\sqrt{\sum_{i=1}^{m} \delta_{m_1}^2}}{\sqrt{m_1 \cdot m_2}}, 1) \tag{5}$$

Lastly, due to the special nature of profile images, we consider a profile image with and without human faces separately.

Definition 5 (Profile Image Similarity, s_5)**.** *Let $p_1^{t_1}$ and $p_2^{t_2}$ be the profile images of $u_1^{t_1}$ and $u_2^{t_2}$, respectively. The profile image similarity score s_5 is defined as:*

$$s_5(p_1^{t_1}, p_2^{t_2}) = \begin{cases} s_5^F(p_1^{t_1}, p_2^{t_2}), \textit{both have faces}; \\ s_5^{\neg F}(p_1^{t_1}, p_2^{t_2}), \textit{none has face}; \\ 0, \textit{otherwise}. \end{cases} \tag{6}$$

For the case when both profile images contain human faces, our approach compares the facial similarity in three steps: (1) *face detection*, (2) *face aligment*, and (3) *face similarity comparison*. In the face dection step, we leverage the Haar Feature-based Cascade Classifier proposed by [66] [46]; in the face aligment step, we leverage the Multitask Cascaded Convolutional Network proposed and implemented by Zhang *et al.* [76]; and finally, we come up with the face similarity definition motivated by the FaceNet proposed by Schroff *et al.* [59]. Embedding [59] is a function f_{emb} whose L^2 norm satisfies $\|f_{emb}(x)\|_2 = 1$. In our study, we define f_{emb} as

$$f_{emb}(px_{i,j}) = \frac{px_{i,j}}{\sqrt{\sum_i \sum_j px_{i,j}^2}} \tag{7}$$

where $px_{i,j}$ is the pixel on the i-th row and j-th colum of an aligned face picture matrix and d is the dimension of the matrix, which is usually two. We define the loss function for the profile images with faces as

$$loss_f = \sum_{i,j} \left[\left\| f_{emb}(px_{i,j}^{p_1^{t_1}}) - f_{emb}(px_{i,j}^{p_2^{t_2}}) \right\|_2^2 \right] \tag{8}$$

Definition 5.1 (Profile Image Similarity with Face, s_5^F)**.** *The face image similarity is defined as*

$$s_5^F(p_1^{t_1}, p_2^{t_2}) = \max(1 - \sqrt{loss_f}, 0) \tag{9}$$

For the case when both images do not have human faces, we leverage the traditional algorithm, the scale invariant feature transform (SIFT) [47], used for object recognition in computer vision. Let $g(x)$ represent the number of SIFT features of picture x, and $m(x, y)$ be the number of matched features between picture x and y with Lowes's ratio test [48].

Definition 5.2 (Profile Image Similarity without Face, $s_5^{\neg F}$)**.**

$$s_5^{\neg F}(p_1^{t_1}, p_2^{t_2}) = \frac{m(p_1^{t_1}, p_2^{t_2})}{\min(g(p_1^{t_1}), g(p_2^{t_2}))} \tag{10}$$

Now we are ready to define our overall profile similarity:

Definition 6 (Profile Similarity)**.** *Let $u_1^{t_1} \in SN_1$ and $u_2^{t_2} \in SN_2$, the profile similarity*

$S : SN_1 \times SN_2 \rightarrow [0, 1]$ *of two user accounts is defined as*

$$S(u_1^{t_1}, u_2^{t_2}) = \sum_{k=1}^{5} w_k s_k(p_1^{t_1}, p_2^{t_2}) \tag{11}$$

where $s_k(p_1^{t_1}, p_2^{t_2})$ *represents the* k*-th attribute similarity of the two user accounts and* w_k *is the corresponding weight that captures the impact of* s_k *on the overall profile similarity.*

In the following we present a learning based model to calculate the weights in Definition 11. Intuitively, the weights should be determined by maximizing the probability of correct matches and minimizing the probability of incorrect matches. Suppose we have a ground truth dataset which includes n_{gt} users from SN_1 and n_{gt} users from SN_2, with the ith user from SN_1 and the ith user from SN_2 are indeed the same person. Based on the profile similarity defined in Definition 11, we can get a matrix $\boldsymbol{S}$ over the pairwise profile similarity scores between SN_1 and SN_2.

$$\mathbf{S}(SN_1, SN_2) = \begin{pmatrix} S(u_1^1, u_2^1) & S(u_1^1, u_2^2) & \dots & S(u_1^1, u_2^{n_{gt}}) \\ S(u_1^2, u_2^1) & S(u_1^2, u_2^2) & \dots & S(u_1^2, u_2^{n_{gt}}) \\ \vdots & \vdots & \ddots & \vdots \\ S(u_1^{n_{gt}}, u_2^1) & S(u_1^{n_{gt}}, u_2^2) & \dots & S(u_1^{n_{gt}}, u_2^{n_{gt}}) \end{pmatrix} \tag{12}$$

where each entry $S(u_1^i, u_2^j)$ is the profile similarity score defined in Definition 11.

In order to find the proper weights, instead of considering only the incorrect matches or correct matches, we came up with a novel loss function which takes into consideration both the correct matches and the incorrect matches, making it more efficient and effective.

The loss function is defined as follows:

$$loss = l(w_1, w_2, \ldots, w_k) =$$

$$\left\langle \left[\begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}_{n_{gt}\times 1} - diag(\boldsymbol{S}) + \frac{1}{n_{gt}-1} \begin{pmatrix} \sum\limits_{j,j\neq 1} S(u_1^1, u_2^j) \\ \sum\limits_{j,j\neq 2} S(u_1^2, u_2^j) \\ \vdots \\ \sum\limits_{j,j\neq n_{gt}} S(u_1^{n_{gt}}, u_2^j) \end{pmatrix} \right]^2 \right\rangle \tag{13}$$

where $\langle\cdot\rangle$ represents the expectation sign. The weight vector is determined by solving the following system of equations.

$$\nabla l = 0 \Longrightarrow \begin{cases} \frac{\partial l}{\partial w_1} = 0 \\ \frac{\partial l}{\partial w_2} = 0 \\ \vdots \\ \frac{\partial l}{\partial w_k} = 0 \end{cases} \tag{14}$$

In our approach, we solve the system of equations (14) by TensorFlow with the gradient descent optimization [58].

4.3 Evaluation

In this subsection, we report the evaluation results of our approach to demonstrate its effectiveness on the DS-SN attack within a single social network and the SN-SN attack across two social networks.

According to the design elaborated in Section 4.2, we implemented the following programs: a malicious app in Java with 3280 lines of code (LOC) for collecting the publicly available Android system state data, which deals with the first step of the DS-SN attack; a server code for activity transition inference and social network event inference with 1800

LOC in Python, which handles the second and third steps of the DS-SN attack; a program of 1595 LOC in Python to crawl the databases of Twitter, Instagram, and Flickr, and perform the device-account matching, which corresponds to the fourth step of the DS-SN attack; and finally, a program with 707 LOC in Python, which performs the SN-SN user account association analysis.

In our experiments, we used five different Android devices: a Nexus 7 with Android version 6.0.1, a HTC One Sense 6 with Android version 5.0.2, a Blu R1 HD with Android version 6.0, a Huawei Honor 8 Lite with Android version 7.0, and a Samsung Galaxy S4 with Android version 4.2.2. In the server side, we used a Dell Inspiron 5559 Specs with Intel Core i7-6500U CPU @ 2.50GHz x 4 running OS Ubuntu 16.04 LTS to conduct the experiments.

4.3.1 Effectiveness of the DS-SN Correlation Attack

We evaluated the effectiveness of our DS-SN attack by i) validating the memory regression learning model, ii) inferring the posting timestamps, iii) inferring the number of characters in a tweet, and iv) inferring the possible list of social network accounts associated with the victim device via multiple posting events.

Performance of the Memory Regression Model

As mentioned in Section 4.2, we leveraged a 5-layer feedforward neural network to learn the patterns of activity transitions in Twitter and Instagram, i.e., from `MainActivity` to `ComposerActivity` in Twitter and from `MediaCaptureActivity I` to `MediaCaptureActivity II` in Instagram. We wrote an automatic program in Python using the Android debug bridge (`adb`) [17] with the UI Automator framework for the generation and collection of the training data and test data. We ran the automatic `adb` Python program in all five Android devices for collecting the training data. As a result, for each of the 5 Android devices, we collected 1,900 pieces of training data for Twitter and 2,340 pieces of training data for Instagram. Thus, in total we have 9,500 pieces of training data

Table 5: Averaged Inference Results on the Posting Timestamps and the Number of Characters in Tweeter

App Name	Averaged Inference Results on the Posting Timestamps and the Number of Characters in Tweeter			
	Types of Random Noisy Actions Performed	Accuracy of Inferrring Posting Timestamps	Average Difference of Timestamps (in seconds)	Accuracy of Inferring Number of Characters in Tweets
Twitter	1. Swiping 2. Tapping 3. Commenting	86.99%	16.35	37.18% ($\epsilon = 0.05$) 91.86% ($\epsilon = 0.15$) 98.92% ($\epsilon = 0.3$)
Flickr	1. Swiping 2. Tapping 3. Commenting	96.81%	15.05	N/A
Instagram	1. Swiping 2. Tapping 3. Commenting	96.73%	19.75	N/A

for Twitter and 11,700 pieces of training data for Instagram. Note that each piece of the training data contains only the target activity transition - no other noisy actions such as swiping. To determine n as mentioned in Section 4.2, we tried multiple different values for the threshold δ and found that the values around 1000 work pretty well for both Twitter and Instagram; thus we set $\delta = 1000$. Also, we set the cessation threshold for the mean squared error to be 10^{-6}. As a result, the mean squared error for the learning process of the Twitter activity transition drops to below 10^{-6} after approximately 30 minutes and the one for the Instagram activity transition drops to below 10^{-6} after approximately 40 minutes, which are reasonably low for deep learning models.

Inference of the Posting Timestamps

The automatic `adb` Python program performed 5,625 tests for all five Android devices, with each device performing 1,125 tests. Each test is for one target social network app and collects a block of data with 6 rows: *VSS*, `tcp_snd`, and `tcp_rcv` for the social network app, `utime` and `stime` for the keyboard process, and *RSS* for the media process, and each row contains 600 values. Therefore, for each Android device, we conducted 375 tests for each target app. It took 5 to 6 minutes to perform one test, during which our automatic `adb` program controlled the target social network app to tweet with a random number of characters (Twitter) or to post a photo (Instagram and Flickr)), and recorded the

Table 6: Inference Results on the Posting Timestamps for Each Device

Device Name	App Name		
	Twitter	Instagram	Flickr
Nexus 7	90.67%	98.67%	99.73%
HTC One Sense 6	82.93%	91.20%	90.40%
Blu R1 HD	91.47%	87.47%	98.67%
Huawei Honor 8	84.27%	94.93%	98.40%
Samsung Galaxy S4	90.93%	85.87%	98.93%

system timestamps of the tweet/photo-post event (ground truth timestamps) for evaluation. The `adb` program also generated random noisy actions such as swiping, tapping, and commenting in the target apps to imitate human beings' actions on these apps during each block collection. Meanwhile, our malicious app collected *VSS*, `tcp_snd`, and `tcp_snd` for the three social network apps and the *RSS* of Android media process as shown in Table 4 for the inference attack.

We considered that a correct inference is made if the difference between the ground truth posting timestamp and our inferred posting timestamp is less than 30 seconds. Table 5 reports the averaged inference results of all Android devices. As one can see, the numbers of correct inferences of the posting timestamps for Twitter, Flickr, and Instagram are 86.99%, 96.81%, and 96.73%, respectively. Table 6 shows the inference results of each Android device against the three target apps. As one can see, generally speaking, our inference of posting timestamps is most effective in Nexus 7 device, and least effective in HTC One Sense 6.

Note that in our memory regression model we set $\sigma = 10^5$, which was determined based on many trials.

Inference of the Number of Characters in a Tweet

To evaluate the inference accuracy on the number of characters in a tweet through CPU `utime` and `stime` of the Keyboard process, we denote the actual number of characters of a tweet by N_{act}, the inferred number by N_{inf}, and set an error parameter ϵ such that our inference is considered a success if the error percentage is less than ϵ, i.e., $\frac{|N_{act}-N_{inf}|}{N_{act}} \leq \epsilon$.

Table 7: Attack Results on Volunteers.

Attack Results on Volunteers												
# of Timestamps	Number of Potential Twitter Identities (Without Number-of-Character Filter)						Number of Potential Twitter Identities (With Number-of-Character Filter)					
	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Average	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Average
1	52,120	61,810	55,132	51,205	50,037	54,060	2,313 ($\epsilon = 0.3$)	2,538 ($\epsilon = 0.3$)	2,272 ($\epsilon = 0.3$)	1,971 ($\epsilon = 0.3$)	2,067 ($\epsilon = 0.3$)	2,232 ($\epsilon = 0.3$)
2	369	403	331	324	347	354	21 ($\epsilon = 0.3$)	19 ($\epsilon = 0.3$)	22 ($\epsilon = 0.3$)	12 ($\epsilon = 0.3$)	10 ($\epsilon = 0.3$)	16 ($\epsilon = 0.3$)
3	52	61	71	43	57	56	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	2 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)
4	11	8	13	9	7	9	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)
5	4	3	5	4	3	3	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)
6	3	2	2	1	1	1	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$))	1 ($\epsilon = 0.3$)
7	3	2	1	1	1	1	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)
8	1	2	1	1	1	1	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)
9	1	1	1	1	1	1	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)	1 ($\epsilon = 0.3$)

In order to evaluate the accuracy of our inference attacks, we randomly sampled 5,000 tweets from those of the 500,008 Twitter users collected in Section 4.2, and re-tweeted these 5,000 tweets using our Android devices. By doing so, we can guarantee that the distribution of the number of characters of our posted tweets follows the one in real-world. The average number of characters of these 5,000 tweets was 189. The results are reported in Table 5, which indicate that if we set ϵ to be 0.05, the possibility of correct inference is 37.18%; if ϵ is set to 0.15, the possibility of correct inference is 91.86%; while if ϵ is set to 0.3, the possibility increases to 98.92%.

Associating Device with Social Network Accounts

We used Twitter to illustrate the performance of associating the victim device with a list of candidate social network accounts. In this evaluation, we conducted two studies, with one on the Twitter celebrities and one on five volunteers. Note that the study on Twitter celebrities cannot identify any device that is associated with a social network account because we do not install our malicious app in any legitimate user in Twitter; nevertheless, the tweeting activities of celebrities can facilitate us to figure out how many people simultaneously tweet at two or more timestamps, which provides a perfect justification to explain why we can correlate the tweeting activities at different timestamps to identify the user account associated with a victim device.

For the study on celebrities, we collected tweets from 5 active Twitter celebrities, with 10 tweets for each at different timestamps, from November 2016 to February 2017, to get

Table 8: Attack Results on Celebrities.

Attack Results on Celebrities	
# of Timestamps	Average Number of Potential Twitter Identities
1	52,016
2	359
3	61
4	6

the data at 50 different timestamps. Each celebrity has at least 20 million followers. We considered celebrities as our study subjects because they not only use real identities, but also are active on Twitter. The results are reported in Table 8, which indicate that the average number of Twitter users who tweeted at one timestamp is 52,016, at the same two timestamps is 359, at the same three timestamps is 61, and at the same four timestamps is 6. These results are interesting as one can see that the number of users who tweeted simultaneously at multiple timestamps drops significantly as the number of considered timestamps increases.

Besides using celebrities as testing subjects mentioned above, we also recruited five volunteers to participate in our evaluation process. Each participant used one of our five Android devices to tweet 10 times at 10 different timestamps with his/her own Twitter account. Each tweet contained a random number of characters. The detailed results are shown in Table 7. If we do not apply the number of tweeted characters as a filter, we found that in average there are 54,060 users who tweeted at one timestamp as the volunteers did. Then the number goes to 354 at two same timestamps, 56 at three same timestamps, 9 at four same timestamps, 3 at five same timestamps, and 1 at six same timestamps. These numbers indicate that an attacker needs to conduct the inference attack for an average of six times in order to associate the Twitter account of one of our voluteers with his/her device without applying the number of characters as a filter. If we apply this filter, as one can see from the results, an attacker only needs to conduct the inference attack for an average of three times to successfully associate one volunteer with his/her Twitter account.

4.3.2 Performance of the SN-SN Correlation Attack

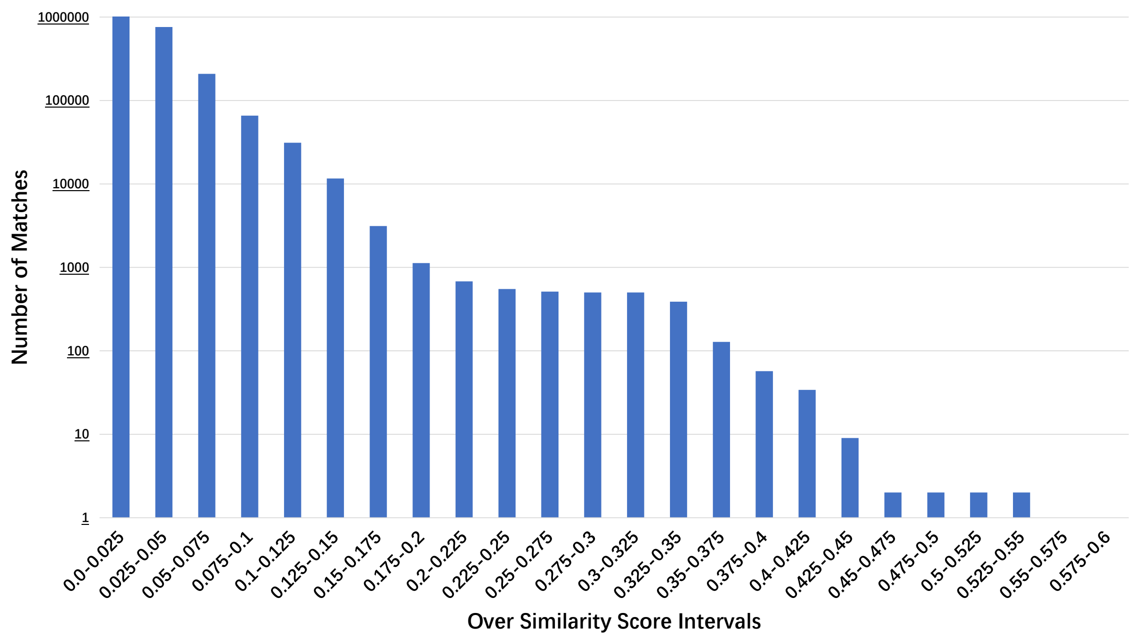
As one can see from the results shown above, an attacker needs to infer at least three timestamps on average to correlate a user's social network account with his/her device with the estimated number of characters as a filer. Without applying this filter, on average at least 6 timestamps need to be inferred. To speed up this process, SN-SN correlation attack can be launched. In this subsection, we evaluate the performance of the SN-SN correlation attack.

Since Instagram does not provide an interface via which we can retrieve posts at any given timestamp, we considered the other two social networks: Twitter and Flickr. To proceed, we need to first derive the weight for each attribute in order to calculate the profile similarity. For this purpose we need ground truth data to train our learning model proposed in Section 4.2.2. We manually checked the tweets of a large amount of Twitter users and filtered out those who do not explicitly provide their Flickr links in one of their tweets. Finally, we obtained 20 pairs of Twitter and Flickr users we were certain that each pair actually refers to the same person. Then we applied these data to minimize the loss function in equation (13). Finally, we obtained the following weights when the loss function reaches its stationary minimum of 0.108886: $\omega_1 = 0.509717$, $\omega_2 = 0.194603$, $\omega_3 = 0.51427$, $\omega_4 = 0.171117$ and $\omega_5 = 0.30354$. Lastly, we normalized these weights by setting $\omega_i^* = \frac{\omega_i}{|\sum_{i=1}^{5} \omega_i|}$, and obtained $\omega_1^* = 0.301029$, $\omega_2^* = 0.114929$, $\omega_3^* = 0.303718$, $\omega_4^* = 0.101058$, and $\omega_5^* = 0.179265$.

As one can see from the above, username (ω_3) plays an extremely important role in determining whether two users from two different social networks are actually the same person, followed by location (ω_1). The least important factor is biography (ω_4), which indicates that people tend to write differently for different social networks.

Next we analyzed the Twitter-Flickr matching performance. We asked each of the five volunteers to send a post in Flickr with his/her Flickr account after he/she finished the 10 tweeting experiments mentioned above. Then we correlated Flickr with Twitter by

Figure 12: Distribution of the Matches



The distribution of the matches of the fifth participant. The x-axis is the intervals of overall similarity scores while the y-axis is the number of matches.

considering only the first timestamp of the Twitter data of all the participants. Next we crawled all the posts in Flickr at the five timestamps corresponding to those at which our five participants sent posts. As a result, we found that there were totally 611 Flickr users who also posted at the same timestamp at which the first participant posted. The numbers of Flickr users who posted at the same time as the second, third, fourth, and fifth participant were 533, 421, 498, and 506, respectively. Therefore, for the first participant, we analyzed a similarity matrix $\boldsymbol{S}_{p_1}$ defined in Section 4.2 with 1,413,243 entries $(2,313 \times 611)$. Similarly, the numbers of entries in the similarity matrices for the second, third, fourth, and fifth participant were 1,352,754 $(2,538 \times 533)$, 956,521 $(2,272 \times 421)$, 945,204 $(1,898 \times 498)$, and 1,112,188 $(2,198 \times 506)$, respectively. Ideally, the majority of the entries in these five matrices should contain very small values, i.e., close to zero; while the entries for the correct matches contain big values, i.e., close to one.

After we evaluated the five matrices with the trained weights obtained above, the similarity matrix of the first participant shows that 1,332,575 entries (94.29%) are below 0.1;

only 3 pairs whose similarity scores are greater than 0.5; and only 1 pair whose similarity score is greater than 0.65, which is the correct matching, i.e., the Twitter account and the Flickr account of the first participant. Similarly, our results showed that over 90% of all the entries of the similarity matrices of the other four participants are below 0.1; and the correct matches corresponds to the pairs with the highest scores. Among the results of all the five participants, the highest overall similarity score is the one for the fifth participant, which is 0.84. Figure 12 illustrates the detailed distribution of the matches of the fifth participant as an example. These results not only indicate that both our definitions of the attribute similarities and the weights work well in real cases, but also imply that our SN-SN correlation attack is effective in facilitating the DS-SN attack to speed up the device-identity association attack.

4.4 Related Works

In this subsection, we summarize the most related work from two aspects: side-channel attacks in mobile systems and privacy attacks in social networks.

Side-channel Attacks in Mobile Systems. Side-channel attacks in mobile systems have been causing more and more concerns in recent years. Chen *et al.* [9] managed to infer the UI state and deploy multiple attacks based on the side-channel information such as memory, CPU, and network statistical data stealthily obtained by a zero-permission malicious app residing in the victim's smartphone. Li *et al.* [45] proposed a keystroke inference attack targeting mobile devices by performing the principal component analysis (PCA) on the channel state information that could be affected by the finger motions through a public WiFi hotspot. Zhou *et al.* [78] exploited the side-channel information of Android devices to infer a user's private information, e.g., rough location, health condition, investment, and driving route. Yang *et al.* [75] presented an approach to discover which website a user is browsing by analyzing the USB power while the smartphone is charging. Song *et al.* [60] managed to crack a 3D printer by reconstructing the physical prints and their corresponding G-code through scrutinizing acoustic and magnetic information obtained from Android built-in sensors. Gruss *et al.* [35] exploited the prefetch instructions to defeat address space layout randomization (ASLR), which is a technique to make the memory address unpredictable for an attacker to launch code-reuse attacks such as return-oriented programming (ROP). Van *et al.* [65] took a step further to attack hardware by launching a row hammer attack and using timing inference to make the row hammer attack deterministic compared to prior attacks which can only succeed on a probabilistic sense. Li *et al.* [44] proposed a side-channel attack to infer the basic living activities by analyzing the changes of the traffic sizes of encrypted video streams in smart home surveillance. Yang *et al.* [74] identified and systematically analyzed a new security issue in HTML5-based hybrid mobile applications, which is termed the Origin Stripping Vulnerability (OSV), and proposed an OSV detection mechanism, namely OSV-Hunter, which leverages the `postMessage` API to defend against OSV from the root. Zhang *et al.* [77] conducted an empirical study on the problem of cross-principal manipulation (XPM) of web resources, and designed a toll named XPM-Checker to automatically detect XPM. An analysis generated by XPMChecker reveals that nearly 49.2% apps from Google Play were affected by the XPM issue [77]. Side-channel information is now an essential ingredient of the mobile system security.

Social Network Privacy. Studies showed that privacy of social networks can be breached in multiple ways. Backes *et al.* [3] proposed a novel link prediction inference attack between any pair of individuals in social networks based on their mobility profiles. Wondracek *et al.* [71] demonstrated that it is possible to de-anonymize a user on a social network through his membership in specific social network groups by exploiting the browser cache in order to detect whether a user has visited certain URLs of a group. Nilizadeh *et al.* [53] proposed a community-based large scale de-anonymization attack using structural similarity. Ji *et al.* [40] presented the seed quantification requirements for perfect de-anonymizability and partial de-anonymizability of real-world social networks, which consolidate the de-anonymization at theory level. Srivatsa *et al.* [61], on the other hand, leveraged mobility traces with social networks as side-channel information to uncover users' real identities. Lai *et al.* [42] exploited a user's interest group information to de-anonymize the users on social networks. Existing studies also tended to reveal hidden attributes for social network users. Chaabane *et al.* [5] exploited a user's interest to explore hidden attributes of a user, e.g., age, gender, and so on. Mei *et al.* [51] reported an inference attack framework that integrates and modifies the existing state-of-the-art convolutional neural network models to infer a user's age. Gong *et al.* [16] proposed a new type of inference against user's hidden attributes such as location, occupation, and interest by analyzing both his social friending and behavioral records. Hassan *et al.* [37] analyzed the privacy threats in fitness-tracking social networks and developed an attack against Endpoint Privacy Zones (EPZs) to extract a user's sensitive locations. They also presented an EPZ fuzzing technique based on geo-indistinguishability to mitigate a user's privacy leak through fitness-tracking social networks. Many of these attacks mainly de-anonymize users based on their social relationships in a large dataset, while our approach can precisely identify the social network accounts of a target user behind a device based on the device's side-channel information, the public social network events, and the profile similarity of the user's accounts in different social networks.

Conclusion and Future Work

In this proposal, we mainly studied the security of Android smartphones from the perspective of system design flaw and from the perspective of side-channel information exploit. More specifically, we study the design flaw of Android Task Mechanism resulting in privilege leakage, and the public side-channel system states which leads to uncover of a user's social network identities.

Our future research focuses on studying security and privacy for other smart devices and their corresponding systems, especially smart home devices and smart home systems. We plan to excavate the security vulnerabilities and privacy concerns of current most popular smart home devices and smart home systems, and develop new attack vectors. Afterward, we plan to design defense mechanisms to guarantee the safety and privacy of these smart home devices and systems.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, 2017. USENIX Association.

[3] M. Backes, M. Humbert, J. Pang, and Y. Zhang. walk2friends: Inferring social links from mobility profiles. In *Proceedings of the 2017 ACM SIGSAC conference on Computer & communications security*, pages 1943–1957. ACM, 2017.

[4] G. Bai, J. Sun, J. Wu, Q. Ye, L. Li, J. S. Dong, and S. Guo. All your sessions are belong to us: Investigating authenticator leakage through backup channels on android. In *20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 60–69, 2015.

[5] A. Chaabane, G. Acs, M. A. Kaafar, et al. You are what you like! information leakage through users' interests. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS)*. Citeseer, 2012.

[6] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 892–903, 2014.

[7] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applica-

tions with permission event graphs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[8] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security)*, pages 1037–1052, 2014.

[9] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1037–1052, 2014.

[10] ELINUX. Android toolbox, 2014. http://elinux.org/Android_toolbox.

[11] Facebook. Facebook access token, 2018. https://developers.facebook.com/docs/facebook-login/access-tokens#usertokens.

[12] Facebook. Facebook graph api, 2018. https://developers.facebook.com/docs/graph-api/overview.

[13] N. Feske and C. Helmuth. A nitpicker's guide to a minimal-complexity secure GUI. In *21st Annual Computer Security Applications Conference (ACSAC)*, pages 85–94, 2005.

[14] Frida. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers., 2018. https://www.frida.re/.

[15] A. Y. Fu, L. Wenyin, and X. Deng. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (emd). *IEEE Transactions on Dependable and Secure Computing*, 3(4):301–311, Oct 2006.

[16] N. Z. Gong and B. Liu. You are who you know and how you behave: Attribute inference attacks via users' social friends and behaviors. *arXiv preprint arXiv:1606.05893*, 2016.

[17] Google. Android debug bridge. https://developer.android.com/studio/command-line/adb.html. Observed in June 2016.

[18] Google. Overview of android memory management, 2016. https://developer.android.com/topic/performance/memory-overview.html.

[19] Google. Recents screen, 2016. https://developer.android.com/guide/components/activities/recents.html.

[20] Google. Activity manager getrunningtasks, 2018. https://developer.android.com/reference/android/app/ActivityManager.html#getRunningTasks(int).

[21] Google. Android broadcast receiver, 2018. https://developer.android.com/reference/android/content/BroadcastReceiver.html.

[22] Google. Android launch mode, 2018. https://developer.android.com/guide/topics/manifest/activity-element.html#lmode.

[23] Google. Dalvik debug monitor server, 2018. https://developer.android.com/studio/profile/ddms.html.

[24] Google. Debug.memoryinfo, 2018. https://developer.android.com/reference/android/os/Debug.MemoryInfo.html.

[25] Google. Facebook login for android, 2018. https://developers.facebook.com/docs/facebook-login/android.

[26] Google. finishandremovetask, 2018. https://developer.android.com/reference/android/app/ActivityManager.AppTask.html#finishAndRemoveTask().

[27] Google. Flag_window_is_obscured, 2018. https://developer.android.com/reference/android/view/MotionEvent.html#FLAG_WINDOW_IS_OBSCURED.

[28] Google. Instrumentation, 2018. https://developer.android.com/reference/android/app/Instrumentation.html.

[29] Google. killbackgroundprocesses, 2018. https://developer.android.com/reference/android/app/ActivityManager.html#killBackgroundProcesses(java.lang.String).

[30] Google. Mediaprojection, 2018. https://developer.android.com/reference/android/media/projection/MediaProjection.html.

[31] Google. Read_frame_buffer, 2018. https://developer.android.com/reference/android/Manifest.permission.html#READ_FRAME_BUFFER.

[32] Google. Requesting permissions at run time, 2018. https://developer.android.com/training/permissions/requesting.html.

[33] Google. Runningappprocessinfo importance, 2018. https://developer.android.com/reference/android/app/ActivityManager.RunningAppProcessInfo.html#importance.

[34] Google. Tasks and back stack, 2018. https://developer.android.com/guide/components/tasks-and-back-stack.html.

[35] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.

[36] R. HACKETT. Massive android malware outbreak invades google play store, 2017. http://fortune.com/2017/09/14/google-play-android-malware/.

[37] W. U. Hassan, S. Hussain, and A. Bates. Analysis of privacy protections in fitness tracking social networks-or-you can run, but can you hide? In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 497–512. USENIX Association, 2018.

[38] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 639–652, 2011.

[39] Iwo Banaś. Android tapjacking. http://www.iwobanas.com/2015/07/android-tapjacking-fix/. Observed in June 2016.

[40] S. Ji, W. Li, N. Z. Gong, P. Mittal, and R. A. Beyah. On your social network de-anonymizablity: Quantification and large scale evaluation with seed knowledge. In *NDSS*, 2015.

[41] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[42] S. Lai, H. Li, H. Zhu, and N. Ruan. De-anonymizing social networks: Using user interest as a side-channel. In *Communications in China (ICCC), 2015 IEEE/CIC International Conference on*, pages 1–5. IEEE, 2015.

[43] T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.

[44] H. Li, Y. He, L. Sun, X. Cheng, and J. Yu. Side-channel information leakage of encrypted video stream in video surveillance systems. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[45] M. Li, Y. Meng, J. Liu, H. Zhu, X. Liang, Y. Liu, and N. Ruan. When csi meets public wifi: Inferring your mobile phone password via wifi signals. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1068–1079. ACM, 2016.

[46] R. Lienhart and J. Maydt. An extended set of haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, volume 1, pages I–I. IEEE, 2002.

[47] D. G. Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[48] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[49] MediaPost. North american consumers to have 13 connected devices, 2018. https://www.mediapost.com/publications/article/302663/north-american-consumers-to-have-13-connected-devi.html.

[50] MediaPost. People will take 1.2 trillion digital photos this year - thanks to smartphones, 2018. https://www.businessinsider.com/12-trillion-photos-to-be\-taken-in-2017-thanks-to-smartphones-chart-2017-8.

[51] B. Mei, Y. Xiao, R. Li, H. Li, X. Cheng, and Y. Sun. Image and attribute based convolutional neural network inference attacks in social networks. *IEEE Transactions on Network Science and Engineering*, 2018.

[52] M. Niemietz and J. Schwenk. Ui redressing attacks on android devices. In *Blackhat Asia*, 2014.

[53] S. Nilizadeh, A. Kapadia, and Y.-Y. Ahn. Community-enhanced de-anonymization of online social networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 537–548. ACM, 2014.

[54] NIST. Cve-2017-0561 detail, 2017. https://nvd.nist.gov/vuln/detail/CVE-2017-0561.

[55] C. Ren, P. Liu, and S. Zhu. Windowguard: Systematic protection of gui security in android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[56] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security)*, pages 945–959, 2015.

[57] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *the 22nd USENIX Security Symposium (USENIX Security)*, pages 97–112, 2013.

[58] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[59] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 815–823, 2015.

[60] C. Song, F. Lin, Z. Ba, K. Ren, C. Zhou, and W. Xu. My smartphone knows what you print: Exploring smartphone-based side-channel attacks against 3d printers. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 895–907. ACM, 2016.

[61] M. Srivatsa and M. Hicks. Deanonymizing mobility traces: Using social network as a side-channel. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 628–637. ACM, 2012.

[62] Statista. Instagram: number of monthly active users 2013-2016, 2018. https://www.statista.com/statistics/253577/number-of-monthly-active-instagram-users/.

[63] Statista. Smart home market revenue in selected countries worldwide in 2018 (in million u.s. dollars), 2018. https://www-statista-com.proxygw.wrlc.org/statistics/483712/global-comparison-smart-home-revenue-digital-market-outlook/.

[64] Statista. Smartphone sales in the united states from 2005 to 2018 (in billion u.s. dollars)), 2018. https://www-statista-com.proxygw.wrlc.org/statistics/191985/sales-of-smartphones-in-the-us-since-2005/.

[65] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689. ACM, 2016.

[66] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE, 2001.

[67] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, pages 365–379, 2012.

[68] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online - security analysis of cashier-as-a-service based web stores. In *32nd IEEE Symposium on Security and Privacy (S&P)*, pages 465–480, 2011.

[69] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

[70] WIKIPEDIA. Alaska, 2018. https://en.wikipedia.org/wiki/Alaska.

[71] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 223–238. IEEE, 2010.

[72] Y. Xiao. demo, 2016. https://www.youtube.com/watch?v=pY_0Sz3AA9Y.

[73] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os x and ios. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 31–43, 2015.

[74] G. Yang, J. Huang, G. Gu, and A. Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 742–755. IEEE, 2018.

[75] Q. Yang, P. Gasti, G. Zhou, A. Farajidavar, and K. Balagani. On inferring browsing activity on smartphones via usb power analysis side-channel. *IEEE Transactions on Information Forensics and Security*, 2016.

[76] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters*, 23(10):1499–1503, 2016.

[77] X. Zhang, Y. Zhang, Q. Mo, H. Xia, Z. Yang, M. Yang, X. Wang, L. Lu, and H. Duan. An empirical study of web resource manipulation in real-world mobile applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1183–1198. USENIX Association, 2018.

[78] 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*, pages 1017–1028. ACM, 2013.