Tian *et al. Cybersecurity* (2018) 1:5
https://doi.org/10.1186/s42400-018-0007-6

Cybersecurity

RESEARCH Open Access

Using IM-Visor to stop untrusted IME apps from stealing sensitive keystrokes

Chen Tian[1], Yazhe Wang[1,2*], Peng Liu[3], Qihui Zhou[1] and Chengyi Zhang[1,2]

Abstract

Third-party IME (Input Method Editor) apps are often the preference means of interaction for Android users' input. In this paper, we first discuss the insecurity of IME apps, including the Potentially Harmful Apps (PHAs) and malicious IME apps, which may leak users' sensitive keystrokes. The current defense system, such as I-BOX, is vulnerable to the prefix substitution attack and the colluding attack due to the post-IME nature. We provide a deeper understanding that all the designs with the post-IME nature are subject to the prefix-substitution and colluding attacks. To remedy the above post-IME system's flaws, we propose a new idea, pre-IME, which guarantees that "Is this touch event a sensitive keystroke?" analysis will always access user touch events prior to the execution of any IME app code. We design an innovative TrustZone-based framework named IM-Visor which has the pre-IME nature. Specifically, IM-Visor creates the isolation environment named STIE as soon as a user intends to type on a soft keyboard, then the STIE intercepts,Android event sub translates and analyzes the user's touch input. If the input is sensitive, the translation of keystrokes will be delivered to user apps through a trusted path. Otherwise, IM-Visor replays non-sensitive keystroke touch events for IME apps or replays non-keystroke touch events for other apps. A prototype of IM-Visor has been implemented and tested with several most popular IMEs. The experimental results show that IM-Visor has small runtime overheads.

Keywords: TrustZone, Android app security, User privacy

Introduction

Nowadays, people are experiencing a booming growth of Android smartphone apps and enjoying their convenience. According to Google Play's statistics(Google 2007), the number of available apps in the Google Play Store was most recently placed at 3.3 million apps in September 2017, after surpassing 1 million apps in July 2013. Most popular apps in the real world can be categorized into six groups: tools, communication, social interaction, efficiency, anime and sports. For example, ES app as a tool app can help users transfer their files in smartphone to a PC desktop.

An Input Method Editor (IME) app is a user-installed app that provides a soft keyboard to receive user input in mobile devices. As shown in Fig. 1, a default IME app appears when a user intends to type characters in a user app (e.g., type a location name in a map searching app). Besides the default IME app, there are many kinds of third party apps in Android market that a user can download from. These third party apps can provide value added features to a user app, such as cloud-based auto correcting, word association and clipboard.

Although an IME app provides great convenience to users, they can introduce serious security problems. An attacker can use this kind of apps to steal users' sensitive keystrokes. As shown in Fig. 2, a keystroke processing in Android works as follows: When a user types a character (e.g., a "K") in a soft keyboard, the touch screen driver will receive a coordinate(x,y), then the event subsystem transfers it into a touch event. Then, the input dispatcher thread will send the event to the target IME app. Finally, IME app will translate the event into a character (i.e., a "K"), and sends it to the target user app. After sending, an IME app can still revisit the buffer of that user app. Here, we can see that an IME app is always the first service to receive (sensitive or non-sensitive) keystrokes prior to

*Correspondence: wangyazhe@iie.ac.cn
[1]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, People's Republic of China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, People's Republic of China
Full list of author information is available at the end of the article

Springer Open

© The Author(s). 2018 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

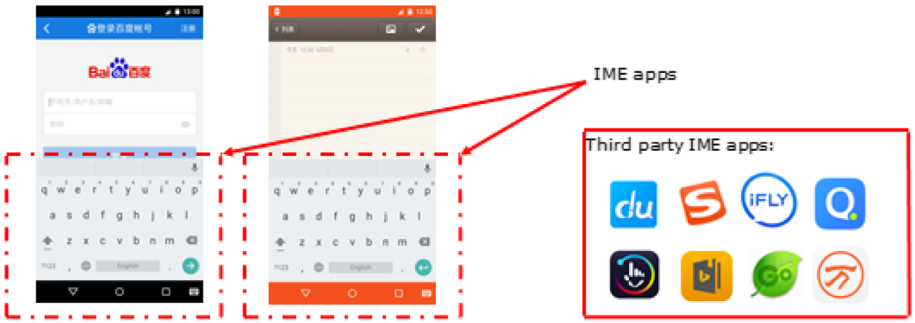

Fig. 1 IME apps in the real world. An IME app is capable to provide a soft keyboard for user input. Typically, there are many kinds of third party apps in Android market that a user can download with his own preference

a user app. Hence, if a user is typing sensitive information (e.g., password, bank number, etc.) in a user app, a malicious IME app can work as a key logger to record and translate sensitive keystrokes, then store them in local file system or send them to a remote server. This is a typical man-in-middle attack. There are several ways to construct such malicious IME apps, and repackaging is a common way which has been widely used by attackers. (Zhou and Jiang 2012) Besides malicious IME apps, there are also threats posed by potentially harmful IME apps (PHAs). Without users' consent, they collect sensitive keystrokes and send them to an ad network doing targeted advertising based on the keywords in user inputs.

To get rid of the above attacks, researchers have recently proposed post-IME defenses. Figure 3 shows the work flow of I-BOX(Chen et al. 2015), which is a well-known post-IME defense. Specifically, it saves the process state of an IME app periodically and analyzes the translated keystrokes from IME apps each time an input transaction happens. If sensitive ones are found, it will let the IME app "forget" sensitive keystrokes by a process roll-back. The I-Box always checks whether a rollback is needed *after* the IME has already processed keystrokes. And the salient feature of the post-IME nature is that sensitive keystrokes appear in the dynamically allocated memory of an IME app at least once.

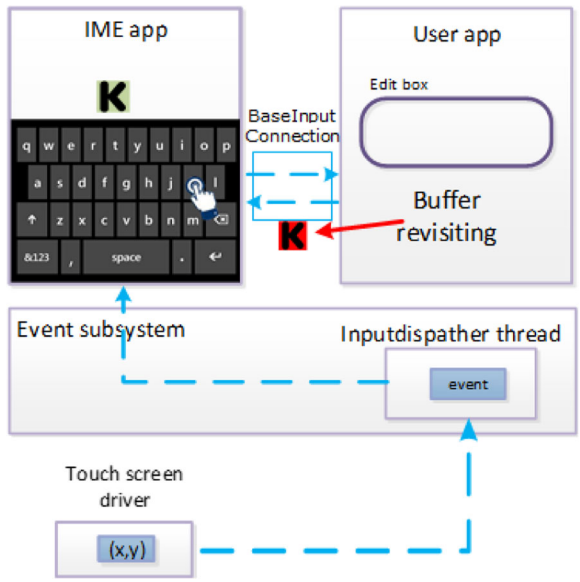

Fig. 2 Keystrokes porcessing in Android. An IME app obtains the coordinate(x,y) from touch screen driver and translate it into a character, then send it to a user app. It is always the first service to receive (sensitive or non-sensitive) keystrokes prior to a user app

Although post-IME defenses can prevent the sensitive data leakage in most common cases, there are still three security holes (discovered so far) in current defense systems, that is, prefix substitution attack, colluding attack and sandbox bypassing attack (newly discovered attack).

Prefix Substitution Attack. Figure 4 is an example of prefix substitution attack to I-BOX. The policy engine in I-Box is a status machine to detect whether the output string of an IME app is sensitive. Assuming the current input is sensitive data, but IME app developers use obfuscated code to replace the prefix of the typed string with a non-sensitive one, then the policy engine is fooled and the roll-back will not be triggered. So the sensitive data obtained by the IME will not be cleaned and can still be sent to a remote server.

Colluding Attack. Figure 5 is an example of colluding attack to I-BOX. To launch a colluding attack, as a post-IME design won't do anything until it gets some output from the IMEs, an IME app needs to send sensitive text to a colluding app before it commits any text to a user app. So it is really easy to launch the attack in the real world.

Sandbox Bypassing Attack. The "revisit" threat is discovered by us and I-Box was not aware of it yet. It is a threat for both post-IME and pre-IME defenses. From the view of the I-Box, it regards the user input process as a transaction, which begins when a user starts to enter the input and ends when the input session ends. When a user is typing sensitive data by a third party IME, the

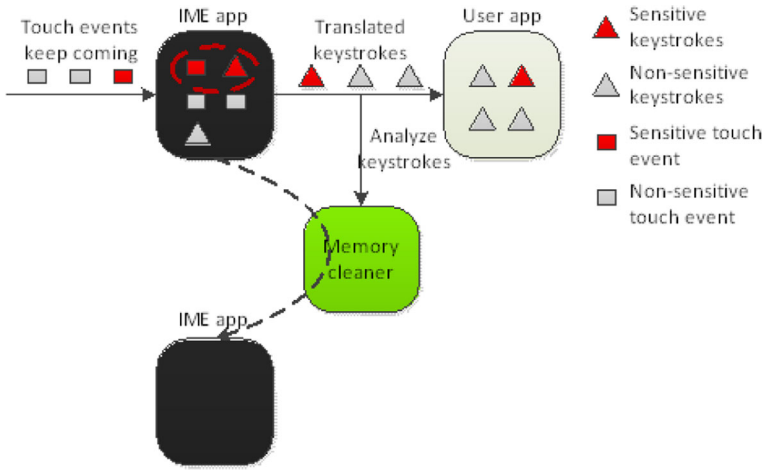

Fig. 3 The work flow of a post-IME defense namely I-BOX. It always checks if a rollback is needed *after* the IME has already processed keystrokes. And the salient feature of the post-IME nature is that sensitive keystrokes appear in the dynamically allocated memory of an IME app at least once

current transaction will be marked as sensitive by I-Box. During this sensitive transaction, I-Box believes that the restriction of network and roll-back can prevent sensitive keystroke leakage. However, the sensitive text exist not only in an IME app while also in the buffer of a user app. The roll-back only cleans the sensitive text in the IME app but remains the one in the user app. In light of the fact that some functions like `getTextBeforeCursor` in BaseInputConnection can be used to revisit the buffer of a user app, an IME app can launch a sandbox bypassing attack by calling revisited APIs at the beginning of the next new transaction. If the user app has not flush the buffer yet, the IME can obtain the sensitive text committed in the last transaction. As a result, the sandbox of I-Box has been bypassed. Figure 6 shows how does the sandbox bypassing attack case work. It is worth noting that the bypass attack is not universally true. In other words, only when the user app does not flush the buffer, there exists such an attack.

Problem statement. How to fill the three security holes through providing the following security property: the analysis on *whether a touch event is a sensitive keystroke* will always access the touch event prior to the execution of any IME app code.

This work seeks to solve the above problem by designing, implementing and evaluating the first pre-IME defense based on 3 key ideas. The defense should ensure that touch events are intercepted before arriving at the system **(Key idea 1)**. Sensitive touch events are never sent to IME apps **(Key idea 2)**. Insensitive touch events should be replayed **(Key idea 3)**.

Challenges. To leverage the above three key ideas, we are facing three main challenges. First, in the existing modern mobile devices, an IME app is the first service to receive (sensitive or non-sensitive) keystrokes from Android event subsystem, and translates them to text. Distinct from a post-IME design which does a rollback after the IMEs translating keystrokes, in a pre-IME design, how can we intercept and isolate sensitive keystrokes ahead of IME translation? This is called the "Isolation ahead of IME translation issue" **(Challenge 1)**. Second, after we succeeding in intercepting and isolating those

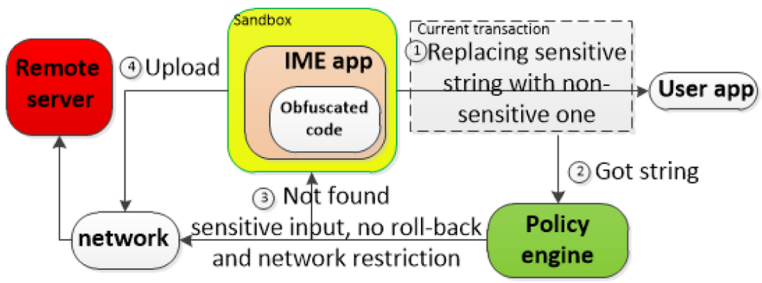

Fig. 4 Prefix-substitution attack. I-Box uses a policy engine to search substring in the output of an IME app. Malicious IME apps can obfuscate sensitive string into non-sensitive string to fool I-Box and leak it out to a remote server

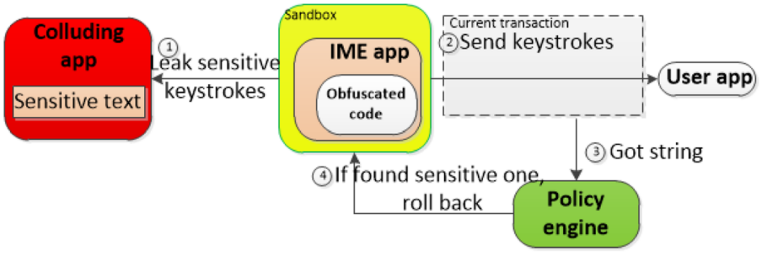

Fig. 5 Colluding attack. To lauch such attack, an IME app just needs to send sensitive text to a colluding app before it commits any text to a user app

sensitive keystrokes, how can we build a trusted path for user apps to access these sensitive keystrokes? We call it the "Trusted path issue" **(Challenge 2)**. Finally, recalling the reason why users got incentives to use IMEs in the first paragraph, an IME app does provide convenience and extra benefits. In a pre-IME design, how can we retain the value added feature for user apps? We call it the "Benefits retaining issue" **(Challenge 3)**.

To address Challenge 1, we leveraged Trustzone and achieved interception ahead of IME translation. The isolation mechanism includes detection of soft keyboards, initialization of STIE (Secure Typed Isolation Environment, touch event processing and keystrokes translation, and sensitiveness analysis. To address Challenge 2, we built a trusted path for sensitive keystrokes to be transferred to the user app through creating a new IPC between the commit-proxy and the user app. To address Challenge 3, we proposed a keystroke replay mechanism.

Our main contributions are summarized as follows.

- We propose a new idea "pre-IME", which guarantees that "Is this touch event a sensitive keystroke?" analysis will always access user touch events prior to the execution of any IME app code.
- We provide a deeper understanding that all the designs with the post-IME nature are subject to the prefix-substitution and colluding attacks. Addressing the two attacks, designs with the pre-IME nature have a clear security advantage over post-IME designs. This is a key new insight of this work.
- We build a concrete pre-IME defense named IM-Visor which leverages TrustZone to isolate sensitive keystrokes before the IMEs could access them. IM-Visor resolves three main challenges: the "Isolation ahead of translation issue", the "Trusted path issue" and the "Benefits retaining issue".
- By noticing that sensitive keystrokes can generally flow both way (i.e., from IME apps to user apps and from user apps to IME apps), we discover a new sandbox bypassing vulnerability of I-Box.
- We perform a thorough evaluation of IM-Visor. We test a set of popular IME apps and the related user apps, no sensitive keystroke leakage caused by IME apps is found. The experimental results show that IM-Visor has small runtime overheads.

Background

Android IME, Input Method Framework (IMF) and event subsystem

Android IMF arbitrates interaction between applications and the current input method (InputMethodManager 2016). A user app can use the standard `TextView` or its subclass to interact with an IME app. InputMethodManagerService (IMMS) in the IMF is a global system service that manages the interaction across the above processes. When a user touches on the TextView of a

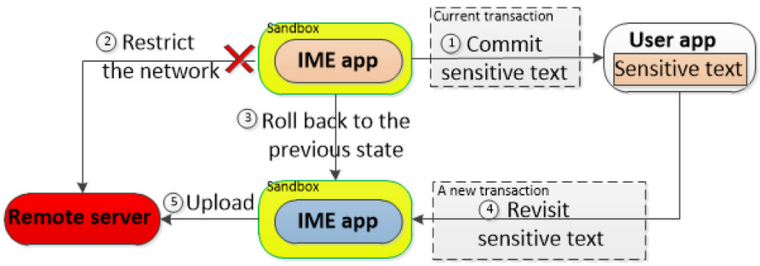

Fig. 6 Sandbox bypassing attack. The pink and blue color represent two different state of an IME app. As shown in blue color, after the roll-back, an IME app can still access the user app's data buffer for sensitive text by some revisited APIs and leak it out to a remote server at the beginning of next input transaction

user app, IMMS will start an IME app. What's more, some functions in IMMS such as `showSoftInput`, `hideCurrentInputLocked` can control when a soft keyboard will be shown up or hidden. If a user types on the soft keyboard, `TouchInputMapper` in the Android event subsystem is the first entity to handle user touch events. After the process of `TouchInputMapper`, an input dispatch thread in WindowManagerService (WMS)(Windowmanager) is responsible to dispatch keystrokes to the active IME app. Then the IME can translate keystrokes to text and commits them to a user app by BaseInputConnection (BIC)(InputConnection). BIC is the connection between a user app and an IME app. BIC provides some functions such as `getTextBeforeCursor`, `getSelectedText` for IME apps to revisit the data buffer in a user app. The reason why these functions exist is that an IME app may need to change some character before finally committing or it just wants to verify the committing. In this paper, we put hooks in some functions in the IMF and event subsystem so that the "Is this touch event a sensitive keystroke?" analysis can be invoked before the IMEs access keystrokes.

TrustZone

Processor state isolation. As hardware-level security isolation, TrustZone provides Secure Monitor Call (SMC) instruction for the processor to enter secure world from normal world. The SMC instruction is a privileged instruction which is invoked in normal world. Program in secure state can access resources across the system including I/O, memory, etc. Normal program has a lower execution privilege.

I/O device and memory isolation. A major feature of TrustZone is that it can flexibly configure the secure state of I/O devices using software. This function involves TrustZone Protection Controller (TZPC) and TrustZone Address Space Controller (TZASC). TZASC allows secure and non-secure area partition for the mobile device DRAM memory. In existing mobile devices, touch screen and display controller are usually configured as non-secure.

Trustlets. An application in secure world is known as a trustlet. It can access the normal world memory but not vice-versa. Considering the TCB size of secure world, a trustlet is usually designed to provide some higher secure operation such as displaying trusted UI or encryption.

Threat model and assumptions

Threat model

As mentioned, in the Android IMF, due to extra benefits, user apps got incentives to use an IME app to access a soft keyboard. However, an IME app is capable of logging and uploading whatever a user types on a soft keyboard. So there is a risk of sensitive keystroke leakage through third patry IMEs. A current defense with the post-IME nature intends to discover sensitive input by analyzing the output of an IME app and cleans it by a roll-back. However, an IME app can fool the defense by committing a replaced text (**Prefix-substitution Attack**) or leaking out sensitive keystrokes with a colluding app before the analysis is triggered (**Colluding Attack**). In "Introduction" section, we have pointed out that all the designs with the post-IME nature are subject to the above two attacks. And a key motivation of our work is that we intend to build a more secure defense to get rid of the above attacks. Besides, we discover a new data leakage path from a user app to an IME app by some revisit APIs (**Sandbox Bypassing Attack**). So the "revisit" is another threat to our security concerns.

It is possible that a malicious user app can collude with an IME app to steal sensitive keystrokes. However, we consider this out of the scope of this paper. Because a user app can get whatever a user types in a soft keyboard, it is unnecessary to steal sensitive keystrokes through an hacked IME app. Besides, from an attacker's point of view, it is much more easier to attack a single IME app than attacking all kinds of user apps which often use an IME keyboard. If an IME app is hacked, all user apps are hacked since an IME app processes all of a user's input in modern mobile devices.

Assumptions

As third party IME apps may cause sensitive keystroke leakage, we consider all third party IMEs (i.e., malicious and PHAs) as untrusted. The goal of IM-Visor is to prevent IME apps from accessing data when a user types sensitive keystrokes, so we assume that the user app which employs an IME app for keystroke translation is trusted. Although there are lots of attacks targeting at user apps (Zhou and Jiang 2012; Suarez-Tangil et al. 2013), such threats are not in the scope of this work. We assume that the Android System Server and the kernel are not on the target list of attackers. As IM-Visor is a security scheme based on TrustZone, so we assume the device is equipped with TrustZone and the function of TrustZone has been correctly implemented on the device. Considering TrustZone is an isolation solution with hardware support, we assume the hardware of the device is trusted. Hardware attacks which may prevent the normal operation of TrustZone are out of the scope of this work.

Although OS is not on the list of attack targets, considering the following facts, we still use TrustZone to implement our defense. First, TrustZone is widely deployed. Data from Samsung shows that millions of modern devices are outfitted with TrustZone (Azab et al. 2014). We hypothesize that more and more devices will use the ARM

TrustZone in the future. Second, the Trusted Execution Enviroment (TEE) is already deployed, there seems less a need to reinvent wheels. Comparing to adding system or kernel code, it is really more convenient to put our critical code as a trustlet in secure world and only put some hooks in Android. Third, it ensures minimum kernel modification. In our design, only a TrustZone driver is needed to be installed in kernel. No kernel instrumentation is needed. Forth, no significant impact on system overheads by testing with most popular IMEs. Fifth, TrustZone does reduce our attack surface. For example, using a gravity sensor to launch a side channel attack is possible when a user types on a soft keyboard. TrustZone can configure related hardware as secure to thwart such attack.

Overview

Figure 7 shows the system components of IM-Visor, which includes a *Secure Typing Isolation Environment (STIE)* in secure world, a system service named *commit-proxy*, a daemon thread named *replay executor* in the event subsystem and some hooks. The STIE includes two parts: secure hardware drivers and a trustlet named *pre-IME guard*. As mentioned in "Introduction" section, in order to create a defense with the pre-IME nature, there are three main challenges: "Isolation ahead of IME translation issue", "Trusted path issue", "Benefits retaining issue". Now we give a high-level overview of how IM-Visor resolves them.

Isolation ahead of IME Translation Issue. In existing mobile devices, an IME app is the first entity to receive user touch events, and then translates keystrokes to text. To achieve a pre-IME design, we must recognize sensitive keystrokes and isolate them before an IME app could access them.

One possible way is to leverage TrustZone to implement a trusted IME app with a trusted GUI. When users intend to type sensitive data, let them switch to the trusted IME. However, this approach brings two disadvantages. First, it is a burden for users to constantly keep this switch in mind. Second, a friendly trusted GUI means a lot of extra coding work, such as efficient graphics rendering. So we have to look for a new approach.

In the light of the fact that keystrokes will be preprocessed by the event subsystem before an IME app could access them, we put some hooks in event subsystem and leverage TrustZone to achieve the pre-IME nature. Subsystem hooks make SMC calls and jump to secure world. In secure world, IM-Visor provides the STIE in which the touch screen and display devices are only controlled by secure world. For touch input, we implement a separate touch driver in secure world. Hence, whenever a touch input interrupt arrives, IM-Visor would be the first to access keystrokes prior to the execution of any IME app code. The pre-IME Guard receives keystrokes, translates them and analyzes whether the char string is sensitive. Concerning about the flexibility and efficiency, the STIE will be created only when a user intends to type in a soft keyboard (see "STIE initialization" section).

Compared to the development of an trusted IME app, the STIE helps IM-Visor avoid the above two disadvantages. First, as the STIE can be initialized automatically when a user intends to type in a soft keyboard, a user does not have to keep the keyboard switch in mind. Second,

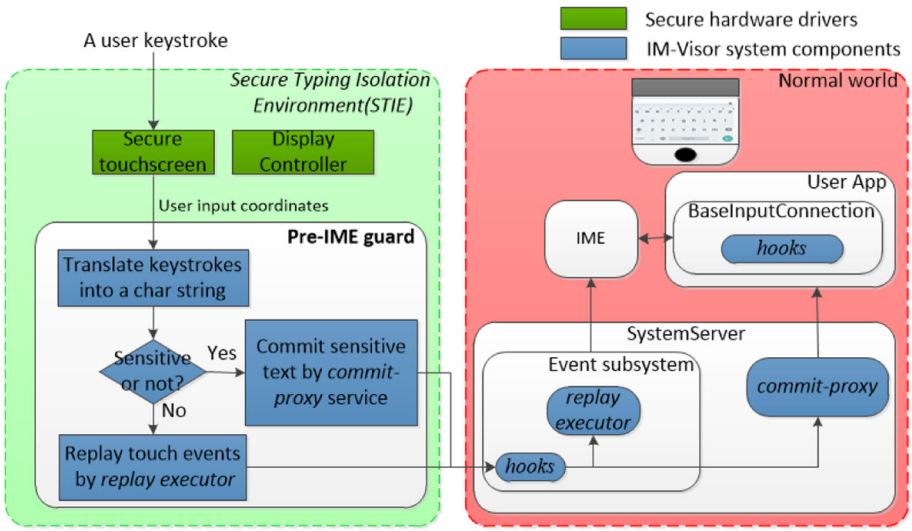

Fig. 7 IM-Visor consists of the *STIE* in secure world, a new service named *commit-proxy*, a daemon thread named *replay executor* in event subsytem and some hooks

because the STIE reuses the UI of a soft keyboard and isolates touch input, no trusted GUI lib is needed.

Trusted path issue. As mentioned in "Introduction" section, after the isolation of sensitive keystrokes, we must build a trusted path from the pre-IME Guard to a user app (Zhou et al. 2012; trustonic). Obviously, we cannot use untrusted IME apps to commit sensitive text as this violates our security principle. So we have to find another data path isolated from IME apps. In light of the fact that TextView of a user app uses a local binder named `IInputContext.Stub` to receive text, we put some hooks in the IMF and create a new connection between a user app and our newly added service named *commit-proxy*. In other words, we create a new interprocess communication (IPC) between a user app and the commit-proxy to commit sensitive text.

Benefits retaining issue. As an IME app does provide convenience and extra benefits, in a pre-IME design, we must retain the value added feature for user apps. The key idea of IM-Visor is to replay a keystroke as soon as the pre-IME Guard determines it as non-sensitive and let the IMEs work for non-sensitive keystrokes. To achieve this, we design *replay executor* running in System Server process for replay. Specifically, the Replay Executor gets touch event coordinates from the pre-IME Guard and encapsulates them into Android touch event format, then triggers event subsystem to dispatch events to IME apps. Another issue related to replay is that we must replay non-keystroke touch events for the other apps.

Design and implementation

Workflow of IM-Visor

As a pre-IME design, IM-Visor always recognizes and isolates sensitive keystrokes *before* the IMEs could access them. To achieve this, whenever a user intends to type in a soft keyboard, the STIE will be initialized to intercept touch events and analyze whether it is a sensitive keystroke. From the perspective of how touch events (i.e., keystrokes or non-keystrokes) are handled, Fig. 8 shows the workflow of IM-Visor *after* the STIE has been initialized. The red data path indicates the trusted path from touch screen to a user app. On the other hand, as shown in green color, when non-sensitive touch events (i.e., non-sensitive keystrokes or non-keystrokes) are found, the pre-IME Guard asks the Replay Executor to replay the corresponding touch event to the targeted apps (e.g., IME apps or other apps).

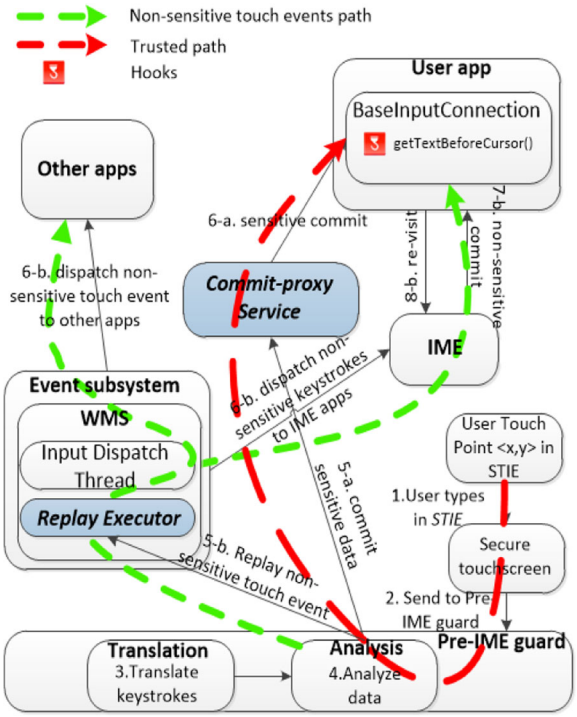

Fig. 8 Workflow and data paths under the IM-Visor protection. For sensitive keystrokes as shown in red color, a trusted path from the secure touch screen to a user app is created by the STIE and commit-proxy. For non-sensitive keystrokes, the Replay Executor dispatches them to the targeted IME app

Address challenge 1: isolation ahead of IME translation

At first, let's recall some backgrounds about the IMF and event subsystem in "Android IME, Input Method Framework (IMF) and event subsystem" section. A keystroke would be preprocessed by event subsystem before any IME app could access it. Specifically, `TouchInputMapper` in event subsystem is the class for touch event processing. InputMethodManagerService (IMMS) in the IMF is a global system service that manages the interaction across IME apps and user apps. Anytime a user app requests a soft keyboard, IMMS would ask an IME app to show a soft keyboard by calling `showSoftInput`.

STIE initialization

The primary technical challenge of the STIE initialization is guaranteeing that IM-Visor is always aware of when a user is typing in a soft keyboard prior to the execution of any IME app code. If IM-Visor can create the STIE as soon as a user firstly puts his or her finger on a soft keyboard in normal world, then the pre-IME Guard is able to intercept user keystrokes from the start of input and the pre-IME nature can be ensured. To address this challenge, the key idea is to check whether a soft keyboard has been shown up each time a touch event arrives in event subsystem. In modern mobile devices with a touch screen, we assume that a user intends to type text when he or she taps on touch screen after a soft keyboard has been shown

up. And the keyboard display information is maintained in secure world.

Figure 9 shows how we initialize the STIE. The first user tap on the edit box of a user app will ask IMMS to start up an IME app. This process in fact invokes two hooks: `sync` and `showSoftInput`. IM-Visor will ignore the touch but update keyboard display information in secure world. At this moment, the STIE has not been initialized yet. Then the user may taps on a soft keyboard. This behaviour of course invokes `sync` again. At this moment, the STIE must be initialized, because tapping on an IME soft keyboard is obviously a keystroke. We reconfigure peripherals like display controller and touch screen as secure. Then the pre-IME Guard receives touch events directly through secure touch screen.

Touch event processing and keystroke translation

In order to intercept user keystrokes in secure world, the touch screen is reconfigured to be only accessed by secure world, and a separate touch screen driver is implemented in secure world. As a result, anytime a touch interrupt arrives, the driver will be the first to receive user touch coordinates. In order to figure out which keystroke a user types, we need two pieces of information: touch coordinates and the current soft keyboard layout. The touch screen driver in secure world provides a secure way to obtain touch coordinates. Now we explain how the pre-IME Guard gets the soft keyboard layout securely. The soft keyboard layout is a piece of display data in normal world that an IME app puts in `framebuffer`. And `framebuffer` is a region of memory which is allocated by a Linux display driver. The display controller is a peripheral to generate the necessary control signals for data display. To obtain the soft keyboard layout on which a user is typing, the pre-IME Guard takes two steps. Step 1) The display controller is reconfigured as secure by TrustZone TZPC so that normal world cannot change it. This is important because display controller provides information about the start region of `framebuffer`. If the display controller is not controlled by secure world, normal world software can deceive the pre-IME Guard into a wrong `framebuffer` and translated in a wrong soft keyboard layout. Step 2) After a touch event happened, the pre-IME Guard reads `framebuffer` and checks correctness of the layout. As a proof-of-concept prototype, IM-Visor preloads the layout information of popular IME apps and determines whether the layout is correct by comparing the hash of the current layout with the preloaded standard one. As a future work, in step 2, instead of the "preload&check" way, we will obtain the current layout by an efficient optical character recognition (OCR).

Leaving `framebuffer` in normal world is not a secure concern as it seems. Supposing an untrusted IME app intends to figure out which keystroke a user types, it also needs the above two pieces of information. But sensitive touch coordinates only stay in secure world. So an IME app cannot succeed in finding sensitive keystrokes without touch coordinates.

After identifying a keystroke, the pre-IME Guard will translate it into a character. Now we give an example of keystroke translation. For Latin language, every keystroke can directly correspond to a character, but for non-Latin languages candidate words often need to be shown. Here, we only discuss Latin language translation with a qwerty keyboard. Supposing the user types "a" in the soft keyboard, then secure touch screen gets the touch point C(x, y). The preloaded keyboard layout helps the pre-IME Guard determine whether this point falls in the geo

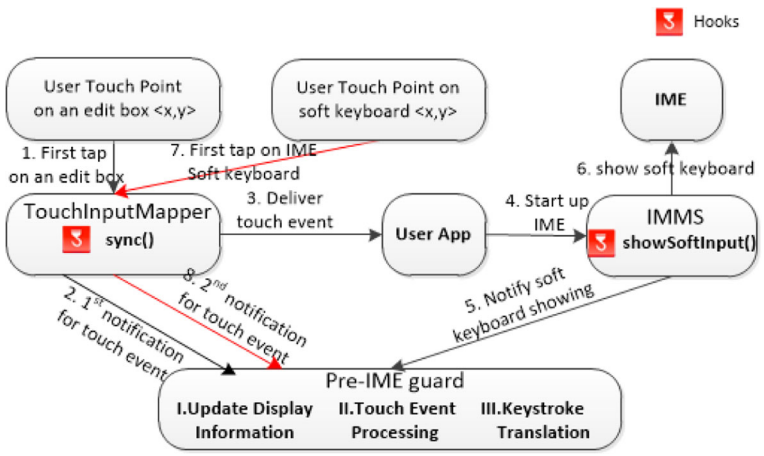

Fig. 9 STIE initialization. Hooks in the IMF and event subsystem are invoked to notify the pre-IME Guard in secure world to initialize the STIE. In modern mobile devices with a touch screen, we assume that a user intends to type text when he or she taps on touch screen after a soft keyboard has been shown up

range of "a" key button, which is defined by the top-left point A(x1, y1) and the bottom-right point B(x2, y2). If it falls in, the pre-IME Guard translate the keystroke as a character "a".

Sensitive keystroke analysis

In order to analyze whether keystrokes are sensitive, we accept the I-BOX's policy engine, which enforces a specific context-based policy and a specific prefix-matching policy. In the IMF, text fields in user apps have different types, such as dates and passwords. IM-Visor can leverage these information to decide whether current input is sensitive or not. Specifically, the hook `startInput` in IMMS can provide information of text fields. If the current edit box works for passwords (or something sensitive like that), the pre-IME Guard will know it from the start of a soft keyboard display and treat all following keystrokes as sensitive. This is called the "Context-based Policy". User activities such as logging in is a typical case that IM-Visor can enforce such policy. For general user input stream, after translating keystrokes to string, IM-Visor leverages prefix-matching to search all possible substrings when a new char is typed (Aho and Corasick 1975). The sensitive data set used for searching is defined by users. As a user could consider large numbers of data instances as sensitive, IM-Visor uses a trie-like structure to maintain it in secure world. This is called the "Prefix-matching Policy".

Address challenge 2: trusted path

After isolating and translating sensitive keystrokes, we should commit them to the targeted user app. Obviously, as a pre-IME defense, we cannot use untrusted IME apps to commit sensitive string. So we have to find another data path isolated from IME apps. Our main idea here is to add an independent system service that can commit sensitive string from secure world to a user app for the trusted path.

Normally, the edit box of a user app uses a local binder named `IInputContext.Stub` to receive char strings. And the client of `IInputContext.Stub` is initialized in the IMMS at the start of input. In light of the above fact, we add some code in the IMF to make the IMMS create an extra binder client for our newly added service *commit-proxy*. And then the commit-proxy is capable of committing sensitive string to the user app. Because the new IPC and new service are independent of an IME app, sensitive string in this data path cannot be accessed by any IMEs.

Create a new IPC. Figure 10 shows how the commit-proxy creates a new IPC with a user app. When a user taps on the edit box, the user app asks IMMS to callback functions in the current active IME app. Hooks in `startInput` will be invoked. The pre-IME Guard creates a token to IMMS, which contains a unique id to identify the current user app. Then IMMS requests the commit-proxy to bind the user app with two parameters (token, `InputConnection`). When the commit-proxy receives this bind request, it makes a SMC call to check whether the id in token is valid. If it is valid, the commit-proxy will add the new connection. Otherwise the bind request will be refused. If any sensitive string is found, the pre-IME Guard sends sensitive string to the commit-proxy by a shared memory, and then the commit-proxy will commit it with the above new IPC.

Address challenge 3: benefits retaining

To retain the extra benefits of IME apps (e.g., auto correcting and word association), one possible way is to implement the value added feature in IM-Visor. However, this way makes all IME apps useless and means a lot of extra coding works for IM-Visor (e.g., cloud-based word association and local trusted GUI lib). We look for a more efficient and elegant approach.

Our key idea here is to design a replay mechanism and let IME apps work for non-sensitive keystrokes. We designed a daemon thread named *replay executor* running in System Server process to replay touch events. If some touch events need to be replayed, the pre-IME Guard puts them in a shared memory and then the Replay Executor reads and replays them. We explain the detail as follows.

In Android system, every activity or service maintains a thread loop to receive touch events or other input events by an input channel. If the Replay Executor intends to replay a touch event directly, it needs to maintain the input channels and selects which activity or service will receive the touch event. The selection is based on not only touch coordinates but also window layouts and the current window focus.

The key point is that the Replay Executor only receives touch events from the pre-IME Guard and then triggers event subsystem to complete the "maintain&selection". As mentioned in "Android IME, Input Method Framework (IMF) and event subsystem" section, an input dispatch thread in WindowManagerService is responsible for touch events dispatching. In most cases, the input dispatch thread sleeps on an input event queue. When a touch event needs to be dispatched, it wakes up and dequeues an event, then selects an activity or service for dispatching. If we can handle this input event queue and wake up the thread when a replay is needed, we are able to let the event subsystem do the "maintain&selection" work. This is exactly how the Replay Executor works. The context of WindowManagerService provides the event queue of input dispatch thread. The Replay Executor encapsulates non-sensitive keystrokes as required Android touch event format and enqueues them. Then it simply wakes

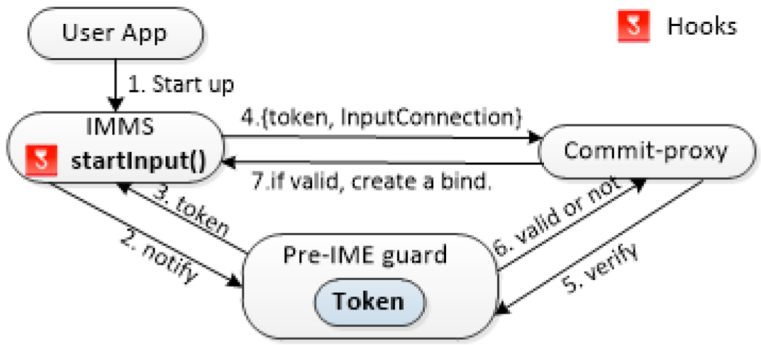

Fig. 10 How a user app binds the commit-proxy. Note that a token is necessary to detect the legitimacy of a user app for the binding

up the dispatch thread to do the rest work for our replay. As a related issue, non-keystroke touch events also can be replayed by this way.

Minor challenge 4: buffer revisiting threat

As discussed in "Threat model" section, we discover a new data leakage path from a user app to an IME app by some revisit APIs. To prevent this threat, we hook all revisit APIs (see Listing 1) and analyze the revisited char string again to detect and block sensitive text when a third-party IME app revisits the buffer of a user app.

Implementation

We have implemented IM-Visor on Samsung 4412 development board equipped with ARM TrustZone. Android and kernel version on the board are 4.0 and 3.0.2 respectively.

Pre-IME guard and services. Specifically, the pre-IME Guard runs as a trustlet in secure world and Android runs in normal world. The commit-proxy is a system service in Android System Server process. The Replay Executor is a daemon thread running in System Server process. Both of them are passively waiting to receive data from the pre-IME Guard. When a user types in the STIE, the pre-IME Guard receives keystrokes from touch screen and translates them into a char string. Corresponding to its sensitiveness, we return it through green path or red path. (see Fig. 8).

Hooks in the IMF and event subsystem. To minimize system overhead, we have to hook as less as possible. Specifically, only three classes in Android has been hooked: `InputMethodManagerService`, `BaseInputConnection` and `TouchInputMapper`. In order to jump into secure world, a TrustZone driver is installed in Linux kernel. Hooks make SMC calls through the newly installed driver. When these hooks are invoked, the pre-IME Guard can intervene the data-flows in Android IMF. Listing 1 shows all the hooks we put in Android.

Secure touch screen and display controller reconfiguration. When a user intends to type in an IME soft keyboard, some reconfiguration should be done for the STIE initialization. We reconfigure Interrupt Security Register(ICDISR), Priority Mask Register (ICCPMR) and Enable Set Register (ICDISER) to make the touch input as a secure interrupt and mask all non-secure intterupt. In CPU Interface Control Register (ICCICR), FIQEn, EnableS are set to 1 to enable FIQ interrupt. FIQ bit in Secure Configuration Register (SCR) is also set to 1 to ensure FIQ interrupt routing to TrustZone monitor mode. Besides, touch screen and display controller are set to be secure peripherals with TZPC. As a proof-of-concept prototype, we only implement single-touch in the separate touch driver and leave multi-touch as a future work.

Evaluation

Security evaluation

Malicious IME apps and PHAs will upload user sensitive data to remote servers, which cause harm to users. To evaluate the defense effectiveness of IM-Visor against malicious IME apps, we construct malicious IME apps by repackaging some popular ones to make them send sensitive keystrokes to a remote server. We want to see whether they can still leak out sensitive keystrokes during the IM-Visor protection. To evaluate the defense effectiveness of IM-Visor against PHAs, we analyze the commonly used IME apps' network packets. As we can see, without IM-Visor, these IME apps may send user sensitive keystrokes outside, and with IM-Visor, user sensitive keystrokes won't be found in their network packets.

Defense against malicious IME Apps

We use repackaging to design malicious IME apps and the targets of repackaging are the popular third party IME apps. As many IME apps have their own different defenses

Listing 1: Hooks in the IMF and event subsystem

```
// hooks in TouchInputMapper
// Callback received by IM−Visor for interception.
void sync(nsecs_t when);

// hooks in InputMethodManagerService
// Callback to create token for the new IPC binding
// and text fields of context−based policy.
InputBindResult startInput(
           IInputMethodClient client,
           IInputContext inputContext,
           EditorInfo attribute,
           int controlFlags);

// Callback to notify IM−Visor, about soft keyboard showing.
boolean showSoftInput(
           IInputMethodClient client,
           int flags,
           ResultReceiver resultReceiver);

// Callback to notify IM−Visor, about soft keyboard hiding
boolean hideCurrentInputLocked(
           int flags,
           ResultReceiver resultReceiver);

// Callback to destroy token.
void finishInput(IInputMethodClient client);

// hooks in BaseInputConnection
// Callback received by IM−Visor for buffer revisiting threat
CharSequence getTextBeforeCursor(
           int length,
           int flags);

// Callback received by IM−Visor for buffer revisiting threat
CharSequence getSelectedText(int flags);

// Callback received by IM−Visor for buffer revisiting threat
CharSequence getTextAfterCursor(
           int length,
           int flags);
```

against repackaging, the difficulty of repackaging on different IME apps is different. Some IME apps just design simple defense which are not difficult to be cracked, and some construct complex solutions which will cost much time to be repackaged. We repackaged three IME apps, one is Sogou IME which is the most popular third party IME, and the other two are QQ IME and TouchPal IME which are also very popular third party IME apps. We can get the smali code for each app after decompiling the APK file and then add some code in several critical locations in the smali file. For example, as most IME apps use the Android IMF which provides various classes and APIs, we can hook the API `commitText` to intercept all the user input. The added code is inserted into the location of `commitText` and the functionality of the added code is to upload each user keystroke to a remote server by socket connections. Finally, the modified code is recompiled, signed and installed in the terminal. After installation, the repackaged IME can be set as the default IME by modifying the system settings. Now we open an app which

needs a user to type user name and password, and we find that the entered user name and password are sent to the remote server when the user is typing.

To verify the defense effectiveness of IM-Visor, we repeat the above operations in the development board with IM-Visor for several times. Figure 11 shows the defense effectiveness of IM-Visor against malicious IME apps. It is clearly shown from the above that IM-Visor can prevent malicious third party IME apps from stealing sensitive data.

Defense against PHAs

Most commercial-off-the-shelf (COTS) IME apps actually collect the user input to improve user experience by analyzing the user input habits or to do targeted advertising. To verify this, we use Wireshark to intercept the network packets when users enter data using IME apps. After experiments on commonly used IME apps, we indeed find that a continuous sequence of packets will be captured by Wireshark when user is typing. For further verification, we need to analyze the content of captured packets. Although some IME apps such as Baidu IME and iFly IME use encryption to prevent the content analysis, there are still other IME apps which upload users' input in plain-text with HTTP protocol. After experiments, IME apps include Sogou (v8.0), QQ (v5.4.0), Octopus (v4.2.6) and TouchPal (for pad, v5.4.5) have dawn our attention. Taking the Sougou IME app as an example, after typing the word "password" in the SMS, we use Wireshark to intercept the network packets of Sogou IME app. Figure 12 shows the intercepted packets. This indicates that sensitive keystrokes have been leaked out by the IME app. With IM-Visor, these potentially harmful IME apps can no longer access the user input when the input is sensitive, as we don't see Wireshark capturing any packets containing sensitive keystrokes.

Correctness evaluation

IM-Visor is a pre-IME design, it intervenes in the communication between user apps and IME apps. As an IME app cannot trigger input by itself, it must be employed by a user app which has edit boxes. So in this section, we need test if user apps and IME apps can normally run with IM-Visor deployed.

First, we need test if user apps and IME apps can run without crashing. To implement this, we first download and install the top 10 IME apps from Android Market. Then we use the Android automated testing tool MonkeyRunner to download 100 user apps from the Android Market. As the touch events triggered by MonkeyRunner are random, we restrict the screen area where touch events can happen based on the location analysis of edit boxes in many user apps. In this way, MonkeyRunner can trigger more keystrokes. For each IME app, we use MonkeyRunner to install and run these 100 user apps. After experiments, we find only 3 user apps crashed and none of the 10 IME apps crashed. For the 3 crashed user apps, we manually run them in

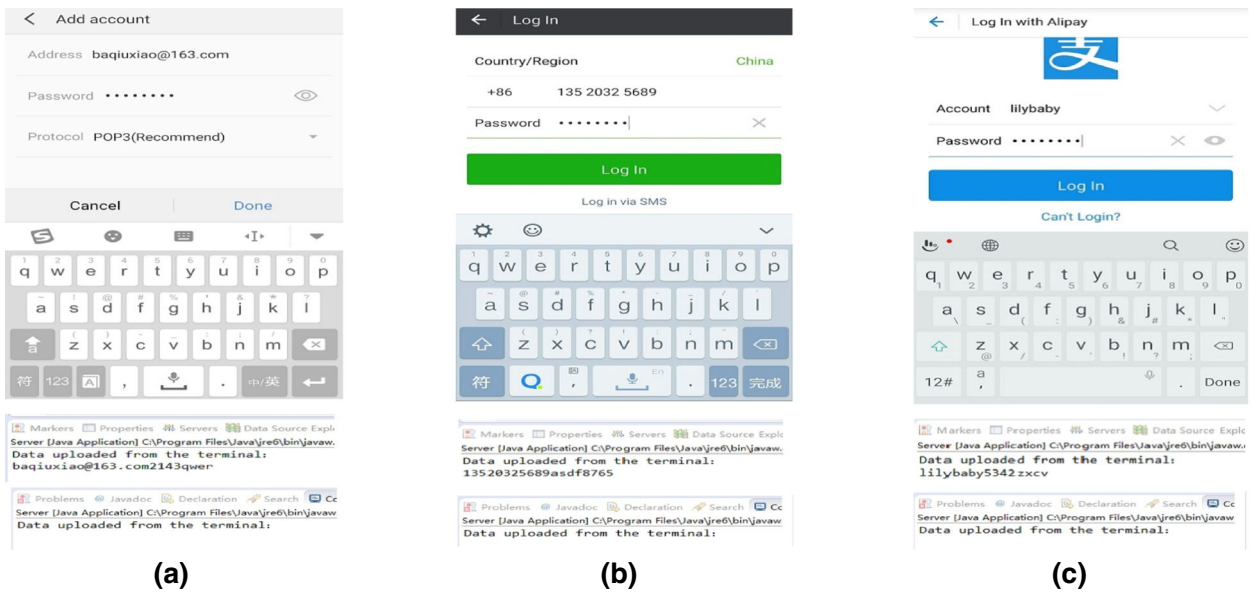

Fig. 11 The defense of IM-Visor on repackaged IME apps. The repackaged IME apps are capable of uploading user names and passwords to the remote server without IM-Visor. However, with the IM-Visor protection, they cannot leak out sensitive information. **a** Email log-in using the repackaged Sogou IME. **b** WeChat log-in using the repackaged QQ IME. **c** AliPay log-in using the repackaged TouchPal IME

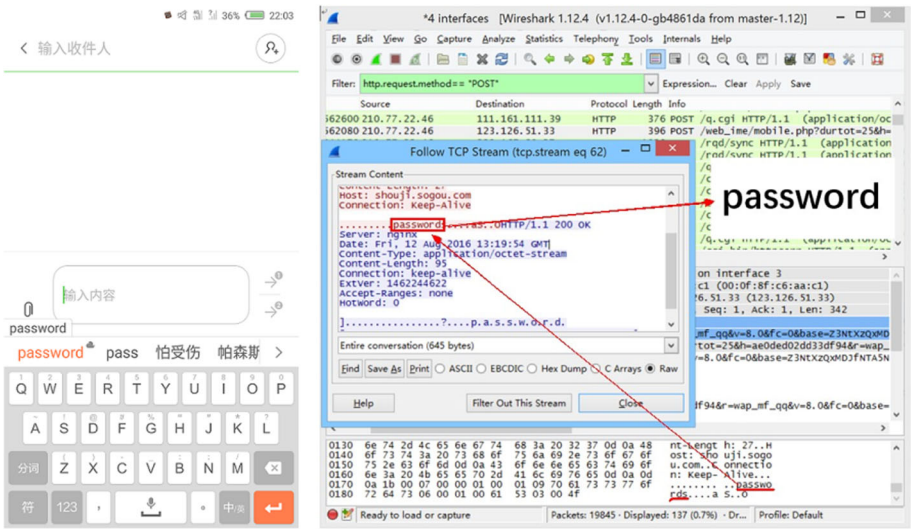

Fig. 12 The analysis on Sogou IME app's network packets. The leaked data "password" appears in one of Sogou IME app's packets

the development board without IM-Visor, however, they still crashed. So we think these 3 user apps crashed because of their bad compatibilities with our development board.

Besides the crash problem, we also need to test if IM-Visor can guarantee that a user app is able to run without any input data missing or input data disorder (the input data are from IM-Visor and IME apps). For each user app, we design several different use cases, and for each use case, we use some commonly used IME apps including Sogou, QQ, TouchPal, Baidu and iFly to test. We have tested 10 typical user apps including the Email Client and SMS. For the Email Client we design two use cases including normal log-in and resumed log-in (i.e., the user is typing and then he picks up a phone call and resumes to log in after hanging up). For the SMS, we also design two use cases including normal text-edit and resumed text-edit. After experiments for 20 times, we manually verify and find that a user app can work normally without any data missing or data disorder.

Usability evaluation

In this evaluation, we need to test how long it costs when a user app can get user input data. This refers to the duration from the time when the first keystroke in the test string happens, to the time when the full string is committed to the user app. The IME apps used for test are Sogou IME, Baidu IME, iFly IME, QQ IME and TouchPal IME. The sensitive data set used for test contains phones numbers, ID numbers, bank card numbers, and email addresses. We select a phone number (11 characters) and an email address (19 characters) for sensitive data test.

Excluding irregular touches (e.g., fumbling phones) and multi-touch behaviour (e.g., zooming gestures), our test is focused on the most common case that a user types characters on a qwerty soft keyboard with his or her single-touch behaviour. The user app used for test is SMS. When a user types text in SMS, the translation results of keystrokes will be analyzed by IM-Visor to decide whether the keystrokes are sensitive. The conclusions can be classified into two types: Sensitive keystrokes and non-sensitive keystrokes.

Sensitive keystrokes. We choose a phone number of 11 characters and an email address of 19 characters which are in the sensitive data set. Then for each IME app, we type the phone number and email address 50 times separately and calculate the average elapsed times, respectively. The results are shown in the left half of Table 1.

Based on the results for sensitive keystrokes in Table 1, we find that the elapsed time taken for the user app to get user input data in IM-Visor is 1.84% longer than the time without IM-Visor deployed. This is mainly due to the overhead of world switches between secure world and normal world. The secure kernel we port is a Linux-like kernel, it takes about 110ms to switch from user mode in secure world (the context of pre-IME Guard) to user mode in normal world (the context of java hooks).

One additional issue is about user experience. For the whole sensitive phone number, although IM-Visor brings only 1.84% reception latency, the display latency may be user-perceptible. Recalling policies in "Sensitive keystroke analysis" section, IM-Visor enforces two different policies (i.e., context-based policy and prefix-matching policy) to

Table 1 Elapsed time for the user app to get the data. We compare the time without/with IM-Visor

IME apps	Sensitive Keystrokes				Non-Sensitive Keystrokes			
	Phone Number		Email Address		Phrases of 15 Characters		Phrases of 25 Characters	
	Without IM-Visor (ms)	With IM-Visor (ms)	Without IM-Visor (ms)	With IM-Visor (ms)	Without IM-Visor (ms)	With IM-Visor (ms)	Without IM-Visor (ms)	With IM-Visor (ms)
Sogou	6143	6256	11028	11132	8565	9315	14658	15972
Baidu	6090	6192	10960	11063	8543	9316	14632	15933
iFly	6302	6408	11332	11433	8890	9632	15130	16456
QQ	6085	6184	10971	11079	8507	9275	14601	15899
TouchPal	6098	6112	10948	11061	8513	9269	14613	15925

analyze user input. With the context-based policy, IM-Visor will treat every single number as sensitive and commit it to a user app one by one, so the display latency is non-perceptible. With the prefix-matching policy, the display latency is user-perceptible for sensitive keystrokes. For a sensitive numeric string like phone number, the prefix-matching of IM-Visor cannot determine the sensitiveness of input until the last number has been typed. Hence, from the view of a user, no character is displayed until the last number of whole sensitive data has been typed. To strike a balance between user privacy and experience, those long sensitive string in user-defined sensitive data set will be maintained in the form of shorter pieces to alleviate the uncomfortable display latency. For example, a sensitive phone number "1320469299" will be automatically maintained in the form of two shorter pieces like "13204" and "69299".

Non-sensitive keystrokes. We select some phrases from the commonly used sentences set (Braden 1969). In order to facilitate the average time calculation, we select 50 different phrases of 15 characters and calculate the average time to input these 50 phrases. Then we select 50 different phrases of 25 characters and calculate the average time. The sensitive data set is the same as the above test, that is, a phone number of 11 characters and an email address of 19 characters. The results are shown in the right half of Table 1.

Based on the results for non-sensitive keystrokes in Table 1, we find that the elapsed time taken for the user app to get user input data in IM-Visor is 9.5% longer than the time without IM-Visor deployed. This is also mainly due to the overhead of world switches between secure world and normal world. Under the pre-fix matching policy, the world switch for sensitive string only needs twice(i.e., from normal world to secure world, and return to normal world from secure world), but for non-sensitive string, this switch may happen many times as the replay mechanism results in a switch from secure world to normal world, so the time taken by IM-Visor for non-sensitive keystrokes is usually longer than that for sensitive keystrokes.

For non-sensitive keystrokes, whether the display latency is user-perceptible depends on how the prefix of non-sensitive typed string matches the prefix of user-defined sensitive data set (i.e., phone numbers in our case). If there is no long common prefix between non-sensitive string and items of sensitive data set, the display latency is non-perceptible. Otherwise, it is perceptible.

Non-keystroke touch events. The above evaluation is about keystrokes, but there are also non-keystroke touch events which will be intercepted by IM-Visor. With the display information in secure world, we optimized the secure kernel to prevent trapping in user mode in secure world when a non-keystroke touch event happened, that is, when no keyboard is shown, the secure kernel will return to normal world imediately without trapping into the pre-IME guard. The optimized world switch here is only 27ms and it will not affect the Android touch event system to distinguish user gesture as the default timeout of a long press in Android is 500 ms .

Performance evaluation

In order to test the performance of the impact of IM-Visor on Android system, we use the CaffeineMark benchmark and compare it to original Android. CaffeineMark is a popular Android benchmarking tool that runs a series of tests and gives an assessment score (John and Eeckhout 2005). We run the benchmark 15 times, each time with a reboot to eliminate impact caused by different system workload, then calculate the average score. The results are in Fig. 13. Overall, IM-Visor performs only 1.53% worse than stock Android. This is mainly due to the reason that the IME is an event-driven service which makes IM-Visor keep idle in most time.

Discussion and limitation

Discussion

We leverage TrustZone to implement the first "pre-IME" defense with hooking Android framework. Recalling the

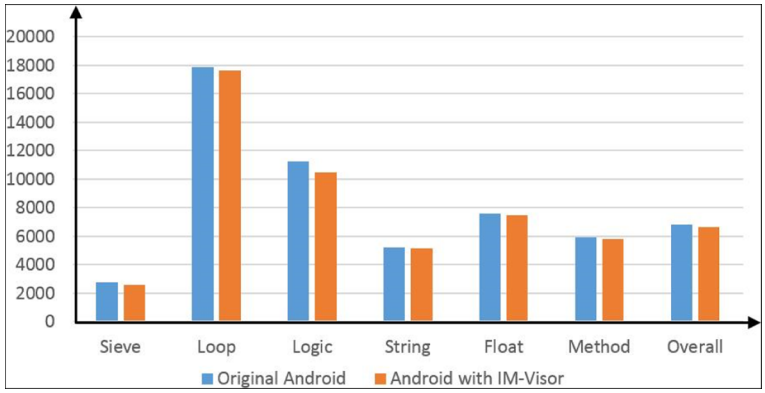

Fig. 13 CaffeineMark results for original Android and Android with IM-Visor

goal of an attacker is to steal sensitive keystrokes, another option is to implement it entirely inside OS kernel without using the TrustZone. However, it would be unsecure considering the following reasons: First, an IME app may get the coordinates from the linux driver interface "/dev/input/event0" directly, which may result in the leakage of sensitive keystrokes. Without the secure touch screen driver in STIE, it can not guarantee the "pre-IME" if not trusting the linux kernel touchscreen driver. Second, the keyboard layout is a key to figure out which keystroke is the user typing, without reconfiguring the display controller by TrustZone, it can not ensure that the `framebuffer` is the right one in current typing environment.

Limitation

Although IM-Visor has made the first "pre-IME" attempt to prevent sensitive keystroke leakage against third party IMEs, some limitations still exist in its design and implementation.

SystemServer attacks Recently, some vulnerabilities have been discovered to attack System Server (Horn 2014; Huang et al. 2015b; Ren et al. 2015; Shao et al. 2016). But none of them can achieve a control flow hijacking, so malicious code cannot modify hooks in System Server to stop IM-Visor from intercepting touch events.

GUI attacks A malicious app may mimic the user app's UI to mount phishing or click-jacking. However, at present, there are quite a few prior systems which can detect such attacks (Bianchi et al. 2015; Akhawe et al. 2014; Huang et al. 2015a).

Side channel attacks Malicious apps may use gravity sensor and acceleration sensor to launch a side channel attack. IM-Visor provides the STIE for user typing, in which we can also reconfigure those related sensors as secure peripherals to thwart such threat (Nahapetian 2016; Aviv et al. 2012).

Related work

Defense against the Android third-party IME apps belongs to a relatively new problem, I-Box(Chen et al. 2015) tries to establish a sandbox mechanism for third-party IME apps, by analyzing the user keystrokes to determine whether to rollback the IME app. As a post-IME design, I-Box is vulnerable to the prefix-substitution attack and colluding attack. In contrast, IM-Visor is a defense with the pre-IME nature and it can defend against the above attacks. Also the solution does not notice the "Buffer revisiting threat", so it can be cracked by sandbox bypassing attack. With hooks in revisiting APIs, IM-Visor can block the data leakage path from a user app to an IME app.

To implement secure password entry, ScreenPass (Liu et al. 2013) designs a trusted software keyboard to enter the password. The use of trusted keyboard in ScreenPass is guaranteed by using the Optical Character Recognition (OCR), but the OCR itself can be cracked by attackers, so the security of ScreenPass cannot be guaranteed. What's more, using a new keyboard instead of the original keyboard will inevitably harm the user experience and the likelihood the user will adopt the new keyboard cannot be guaranteed. In contrast, IM-Visor adopts TrustZone to provide secure isolation, so the security of IM-Visor can be guaranteed. Also IM-Visor reuses the original UI of an IME soft keyboard.

For password and other privacy data protection, researchers have also tried other solutions. Taint-tracking (Kang et al. 2011) is a commonly used method. Taint-tracking tracks the sensitive information flow in the target app and sets appropriate strategies to prevent the outflow and abuse of sensitive data. TaintDroid (Enck et al. 2010) is the first taint-tracking method used in Android and it tracks the flow of sensitive data by

tagging these data. ScreenPass (Liu et al. 2013) also uses taint-tracking to monitor the password flow to prevent illegal outflow. SpanDex (Cox et al. 2014) tracks how password information flows in an app, and compared to the previous work, SpanDex focuses on the implicit information flow in apps. Although the taint-tracking method can get detailed information about sensitive data circulation, it is not very suitable for tracking sensitive keystroke leakage. IME apps usually use native code in their key function such as the send of sensitive inputs, but taint-tracking cannot track the data flow in native code. Regulating ARM (Brasser et al. 2016) thwarts the sensitive information leakage through misused sensors or peripherals on smart personal devices. It replaces the original peripheral drivers by a remote update when a user enters restricted spaces such as a federal building, and doesn't cancel the enforcement of usage policies until the user checks out. App Guardian (Zhang et al. 2015) thwarts the runtime-information-gathering of malicious apps by blocking the runtime monitoring attempt. To realize this, App Guardian pauses the malicious app when sensitive app is running. In contrast, IM-Visor will not pause the normal run of malicious IME apps which results in little impact on Android system. Screenmilker (Lin et al. 2014) constructs an app which exploits the malicious use of the Android ADB capabilities to monitor the screen and pick up a user's password when he or she is typing. Then it presents a mitigation mechanism that controls the exposure of the ADB capabilities only to authorized apps. While IM-Visor and Screenmilker both aim to protect the sensitive keystrokes, there are substantial differences: The threat in Screenmilker is caused by the flaws of the Android permission system, whereas IM-Visor regards IME apps as the threat. The complicated construction of the attacks in Screenmilker makes the attacks difficult to apply widely, while the attacks in IM-Visor commonly exist and can be built using repackaging.

In recent years, TrustZone has obtained lots of research and applications in many aspects. Some researchers aim to improve the security and usability of TrustZone. SecReT (Jang et al. 2015) mainly solves the establishment of secure communication between the Rich Execution Environment (REE) and Trust Execution Environment (TEE). ICE (Sun et al. 2015b) runs the secure code in the non-secure domain by designing isolated secure environment to restrict the code size of TEE environment.

Besides the above ones, more researchers aim to apply TrustZone to protect the sensitive kernel operations and sensitive service. Hypervision (Azab et al. 2014) uses TrustZone to reinforce the Linux kernel by replacing sensitive instructions in Linux kernel and controlling access to sensitive kernel data. TrustOTP (Sun et al. 2015a) uses TrustZone to protect the full process from generation to use for one-time key. TrustDump (Sun et al. 2014) is a TrustZone-based memory acquisition mechanism to detect and prevent the newest malware, and the isolation between the OS and the memory acquisition tool is achieved by TrustZone. These solutions focus on the underlying system especially the kernel, and they have little relation to the Android frameworks. In contrast, IM-Visor does much modification on the Android framework besides the kernel. AdAttester (Li et al. 2015) uses TrustZone to secure online mobile Ad attestation, leveraging the secure world of TrustZone to implement unforgeable clicks and verifiable display. (Marforio et al. 2014) uses TrustZone to ensure the trusted execution environment for the payment process. Similar to the two solutions, IM-Visor aims to protect one certain functional service in Android, but IM-Visor is more comprehensive as the trustlet in IM-Visor needs to complete some functional operation and needs more interaction with Android framework while the trustlet in other two solutions mainly complete the operation such as signature and encryption.

Conclusion

In this paper, we discuss the insecurity of IME apps, including the Potentially Harmful Apps (PHAs) and malicious IME apps. We provide a deeper understanding that all the designs with the post-IME nature are subject to the prefix-substitution and colluding attacks. To remedy the above post-IME system flaws, we propose a new idea, pre-IME, which guarantees that "Is this touch event a sensitive keystroke?" analysis will always access user touch events prior to the execution of any IME app code, and designed an innovative TrustZone-based framework named IM-Visor which has the pre-IME nature. A prototype of IM-Visor has been implemented and tested with several most popular IMEs. The experimental results show that IM-Visor has small runtime overheads.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and suggestions. Yazhe Wang's work was supported by the National Key Research and Development Program of China NO.2017YFB0801900 and Youth Innovation Promotion Association of CAS. Peng Liu was supported by NSF CNS-1422594, NSF CNS-1505664, and NSF SBE-1422215 (social).

Authors' contributions

CT conceived of the study and participated in the design of IM-Visor. YW and PL participated in the implementation of IM-Visor and drafted the manuscript. QZ carried out the evaluation for IM-Visor. CZ participated in drafting the manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, People's Republic of China. [2] School of Cyber Security, University of Chinese Academy of

Sciences, Beijing, People's Republic of China. [3]College of Information Sciences and Technology, Pennsylvania State University, University Park 16802, PA, USA.

Received: 4 January 2018 Accepted: 17 April 2018
Published online: 05 June 2018

References

Aho AV, Corasick MJ (1975) Efficient string matching: an aid to bibliographic search. Commun ACM 18(6):333–340

Akhawe D, He W, Li Z, Moazzezi R, Song D (2014) Clickjacking revisited: A perceptual view of ui security. Usenix Conference on Offensive Technologies. USENIX Association

Aviv AJ, Sapp B, Blaze M, Smith JM (2012) Practicality of accelerometer side channels on smartphones. In: Proceeding ACSAC '12 Proceedings of the 28th Annual Computer Security Applications Conference. ACM, Orlando. pp 41–50

Azab AM, Ning P, Shah J, Chen Q, Bhutkar R, Ganesh G, Ma J, Shen W (2014) Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In: ACM Sigsac Conference on Computer and Communications Security. ACM, Scottsdale. pp 90–102

Bianchi A, Corbetta J, Invernizzi L, Fratantonio Y, Kruegel C, Vigna G (2015) What the app is that? Deception and Countermeasures in the Android User Interface. In: 2015 IEEE Symposium on Security and Privacy. IEEE, San Jose. pp 915–930

Braden WW (1969) Random common sentences. http://www.englishinuse.net/. Accessed Aug 2016

Brasser F, Kim D, Liebchen C, Ganapathy V, Iftode L, Sadeghi AR (2016) Regulating ARM TrustZone Devices in Restricted Spaces. International Conference on Mobile Systems, Applications, and Services. ACM, Singapore. pp 413–425

Chen J, Chen H, Bauman E, Lin Z, Zang B, Guan H (2015) You shouldn't collect my secrets: thwarting sensitive keystroke leakage in mobile ime apps. In: Proceeding SEC'15 Proceedings of the 24th USENIX Conference on Security Symposium. USENIX Association, Washington, D.C. pp 675–690

Cox LP, Gilbert P, Lawler G, Pistol V, Razeen A, Wu B, Cheemalapati S (2014) Spandex: Secure password tracking for android. Usenix Conference on Security Symposium. USENIX Association

Enck W, Gilbert P, Chun BG, Cox LP, Jung J, Mcdaniel P, Sheth AN (2010) Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. In: Usenix Conference on Operating Systems Design & Implementation. USENIX Association, Vancouver. pp 393–407

Google (2007) Number of available applications in the google play. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed Nov 2017

Horn J (2014) Cve-2014-7911: Privilege escalation using objectinputstream. https://www.reddit.com/r/netsec/comments/2mr9cz/cve20147911_android_50_privilege_escalation_using /. Accessed Nov 2014

Huang J, Li Z, Xiao X, Wu Z, Lu K, Zhang X, Jiang G (2015a) SUPOR: Precise and scalable sensitive user input detection for android apps. Usenix Conference on Security Symposium. In: 24th USENIX Security Symposium (USENIX Security 15). USENIX Association, Washington, D.C. pp 977–992

Huang H, Zhu S, Chen K, Liu P (2015b) From system services freezing to system server shutdown in android: All you need is a loop in an app. ACM Sigsac Conference on Computer and Communications Security. ACM, Denver

InputConnection. Android Developer. https://developer.android.com/reference/android/view/inputmethod/InputConnection.html/. Accessed Nov 2016

InputMethodManager (2016) The reference of android developer. https://developer.android.com/reference/android/view/inputmethod/InputMethodManager.html. Accessed Nov 2016

Jang J, Kong S, Kim M, Kim D, Kang BB (2015) SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. Network and Distributed System Security Symposium. The Internet Society, San Diego

John LK, Eeckhout L (2005) Caffeinemark 3.0. http://www.benchmarkhq.ru/cm30/info.html. Accessed Nov 2016

Kang MG, Mccamant S, Poosankam P, Song D (2011) DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. Network and Distributed System Security Symposium, NDSS. The Internet Society, San Diego

Li W, Li H, Chen H, Xia Y (2015) Adattester: Secure online mobile advertisement attestation using trustzone. In: MobiSys '15 Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services. ACM, Florence. pp 75–88

Lin CC, Li H, Zhou XY, Wang XF (2014) Screenmilker: How to Milk Your Android Screen for Secrets. Network and Distributed System Security Symposium. The Internet Society, San Diego

Liu D, Cuervo E, Pistol V, Scudellari R, Cox LP (2013) ScreenPass: secure password entry on touchscreen devices. In: Proceeding of the, International Conference on Mobile Systems, Applications, and Services. ACM, Taipei. pp 291–304

Marforio C, Karapanos N, Soriente C, Kostiainen K, Čapkun S (2014) Smartphones as Practical and Secure Location Verification Tokens for Payments. Network and Distributed System Security Symposium. The Internet Society, San Diego

Nahapetian A (2016) Side-channel attacks on mobile and wearable systems. IEEE Consumer Communications & NETWORKING Conference. IEEE, Las Vegas

Ren C, Zhang Y, Xue H, et al (2015) Towards discovering and understanding task hijacking in android. In: Usenix Conference on Security Symposium. USENIX Association, Washington, D.C. pp 945–959

Shao Y, Ott J, Chen QA, Qian Z, Mao ZM (2016) Kratos: Discovering inconsistent security policy enforcement in the android framework. In: Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS'16). ISOC

Suarez-Tangil G, Tapiador JE, Peris-Lopez P, Ribagorda A (2013) Evolution, detection and analysis of malware for smart devices. IEEE Commun Surv Tutor 16(2):961–987

Sun H, Sun K, Wang Y, Jing J (2015a) TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In: ACM Sigsac Conference on Computer and Communications Security. ACM, Denver. pp 976–988

Sun H, Sun K, Wang Y, Jing J, Jajodia S (2014) TrustDump: Reliable Memory Acquisition on Smartphones. European Symposium on Research in Computer Security. Springer, Wroclaw

Sun H, Sun K, Wang Y, Jing J, Wang H (2015b) Trustice: Hardware-assisted isolated computing environments on mobile devices. In: Ieee/ifip International Conference on Dependable Systems and Networks. IEEE Computer Society, Rio de Janeiro. pp 367–378

trustonic Trustzone, tee and trusted video path implementation. http://www.arm.com/files/event/Developer_Track_6_TrustZone_TEEs_and_Trusted_Video_Path_implementation_considerations.pdf. Accessed Nov 2016

Windowmanager. Android Developer. https://developer.android.com/reference/android/view/WindowManager.html/. Accessed Nov 2016

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

Zhou Y, Jiang X (2012) Dissecting android malware: Characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy. IEEE, San Francisco. pp 95–109

Zhou Z, Gligor VD, Newsome J, McCune JM (2012) Building verifiable trusted path on commodity x86 computers. In: 2012 IEEE Symposium on Security and Privacy. IEEE, San Francisco. pp 616–630

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com