Mind-Reading: Privacy Attacks Exploiting Cross-App KeyEvent Injections* **

Wenrui Diao[1], Xiangyu Liu[1], Zhe Zhou[1], Kehuan Zhang[1], and Zhou Li[2]

[1] Department of Information Engineering,
The Chinese University of Hong Kong, Hong Kong
{dw013,lx012,zz113,khzhang}@ie.cuhk.edu.hk
[2] IEEE Member, Boston, MA, USA
lzcarl@gmail.com

Abstract. Input Method Editor (IME) has been widely installed on mobile devices to help user type non-Latin characters and reduce the number of key presses. To improve the user experience, popular IMEs integrate personalized features like reordering suggestion list of words based on user's input history, which inevitably turn them into the vaults of user's secret. In this paper, we make the first attempt to evaluate the security implications of IME personalization and the back-end infrastructure on Android devices. In the end, we identify a critical vulnerability lying under the Android KeyEvent processing framework, which can be exploited to launch cross-app KeyEvent injection (CAKI) attack and bypass the app-isolation mechanism. By abusing such design flaw, an adversary is able to harvest entries from the personalized user dictionary of IME through an ostensibly innocuous app only asking for common permissions. Our evaluation over a broad spectrum of Android OSes, devices, and IMEs suggests such issue should be fixed immediately. All Android versions and most IME apps are vulnerable and private information, like contact names, location, etc., can be easily exfiltrated. Up to hundreds of millions of mobile users are under this threat. To mitigate this security issue, we propose a practical defense mechanism which augments the existing KeyEvent processing framework without forcing any change to IME apps.

Keywords: Mobile security · Smart IME · Privacy leakage · System flaw

1 Introduction

Smartphone is becoming the major device for handling people's daily tasks like making calls, sending/receiving messages and surfing the Internet. Of particular importance in supporting these features are input devices. Among them,

* Responsible disclosure: We have reported the CAKI vulnerability and the corresponding exploiting schemes to the Android Security Team on January 7th, 2015.

** The video demos can be found at https://sites.google.com/site/imedemo/.

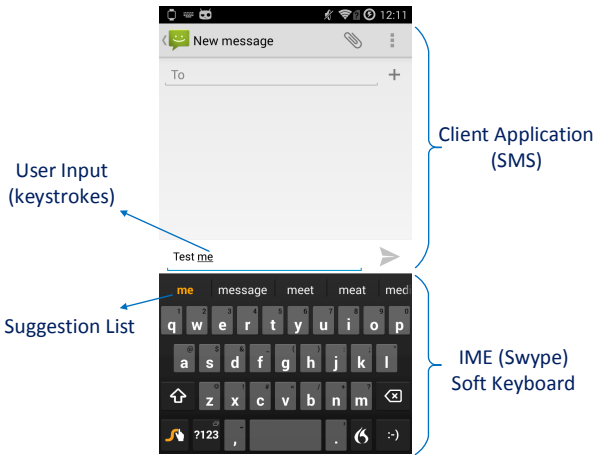

Fig. 1. Smart IME on Android

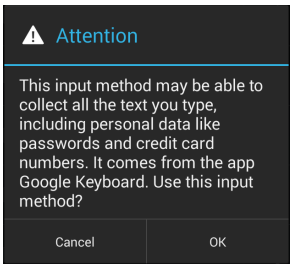

Fig. 2. Warning message

keyboard, either hardware keyboard integrated within mobile phone or soft keyboard displayed on touch screen, receives a significant volume of users' input. These keyboards are mostly tailored to users speaking Latin languages. Users in other regions like Chinese and Japanese have to use Input Method Editor (or IME) to type non-Latin characters. In fact, a large number of IME apps[3] have emerged since the advent of smartphone and been installed by enormous population. The capabilities of IME are continuously extended to optimize users' typing experience. The present IME (see Fig. 1) is able to learn the words a user has inputted, customize the suggested words, and predict the words the user plans to type. These user-friendly features help the IME gain popularity even among Latin-language users.

The wide adoption of IME, however, does not come without cost. Previous research has raised the privacy concerns with **shady IMEs** which illegally spy on users' input [30,35,38,34]. Indeed, they could cause security and privacy issues if installed by common users, but their impact is limited as the majority of IMEs are well-behaved and there have been efforts in warning the risk of enabling a new IME (see Fig. 2). The question not yet answered is whether **legitimate IMEs** are bullet-proof. If the answer is negative, they can be exploited by adversary as stepping stones to breach the privacy of mobile users. In this work, we examine smart IMEs (the ones supporting optimization features) and the backend framework in an attempt to verify their security implications. We choose Android as a target platform given its popularity and openness.

KeyEvent Processing. We first look into the underlying framework which handles input processing. In short, each key press on hardware keyboard triggers a sequence of `KeyEvents` [11] on Android. As for the purpose of automated testing, a mobile app can also simulate key presses by directly injecting

[3] We use IME and IME app interchangeably in this paper.

KeyEvents. Without a doubt, such behavior should be confined to prevent a malicious app from illegally injecting KeyEvents to another victim app. Android consolidates KeyEvent dispatching by ensuring that either the KeyEvent sender app and receiver app are identical or sender app has a system-level permission (`INJECT_EVENTS`) which cannot be possessed by third-party apps. Failing to pass such check will cause KeyEvent being discarded and an exception thrown.

Our Findings. Unfortunately, this seemingly invulnerable framework can be cracked. If a malicious app injects KeyEvents to its owned `EditText` widget with IME turning on, the KeyEvents will be redirected to the IME, resulting in **cross-app KeyEvent injection (CAKI)** attack. Following this trail, attacker can peep into IME dictionary (usually stored in the internal storage protected by app-isolation mechanisms) and know user's favorite words or even the words taken from other sensitive sources, like phone contact. The root cause of this vulnerability is that Android only performs security checks before KeyEvent is dispatched but misses such when KeyEvent is delivered. For this special case, because of the discrepancy between the point of checking and the point of delivering, IME is turned into the final receiver on the fly when KeyEvent is delivered, therefore the security check at the beginning is bypassed. Since this issue exists in the system layer, all IMEs are potentially under threat.

Attack Against IME. Even knowing this vulnerability, a successful attack against IME is not trivial. The challenges include how to efficiently extract words related to personal information or interest and how to hide the malicious activities from user. Towards solving the first challenge, we devise new technique to automatically enumerate the combinations of prefix letters and use differential analysis to infer the words private to user. This technique can be adapted to different language models and all achieve good results. To address the second challenge, we combine several well-founded techniques to make the attack context-aware and executed till user is absent.

We implemented a proof-of-concept malicious app named *DicThief* and evaluated it against 11 very popular IMEs and 7 Android OS versions. The result is quite alarming: all the Android versions we examined are vulnerable and most of the IMEs we surveyed are not immune. The population under threat is **at a scale of hundreds of millions** (see IME popularity in Table 3). Towards mitigating this urgent issue, we propose an origin-checking mechanism which augments the existing Android system without forcing any change to IME apps.

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

• *New Vulnerability.* We discovered a fundamental vulnerability in the Android KeyEvent processing framework leading to CAKI attack.

• *New Attack Surface.* We show by launching CAKI attack, an attacker can steal a variety of private information from IME dictionary. Differing with previous IME-based attacks, our attack is the first to exploit the innocent IMEs.

• *Implementation, Evaluation, and Defense.* We implemented the attack app DicThief and demonstrated the severeness of this problem by testing under different real-world settings. We also propose a defense scheme as a remedy.

2 Background and Adversary Model

2.1 IME and Personalized User Dictionary

IMEs have emerged to support users speaking different languages like English and Chinese. A smartphone is usually shipped with pre-installed IMEs, but alternatively, users could download and use other IME apps. IME have gained massive popularity: top IMEs like Sogou Mobile IME [16,17] has more than 200 million active users.

The IMEs used today have been evolved from solely soft keyboard to versatile input assistant with many new features to improve users' typing experience. The goals of these new features are to reduce the number of keys a user needs to type. For instance, current mainstream IMEs (such as SwiftKey [18], TouchPal [21], Sogou Mobile IME, etc.) implement features like dynamic suggestions order adjustment, contact names suggestions, next-word prediction and new word saving to provide suggestions for current or subsequent words. Hence, a user could select a word among them without typing the complete text. These features are called "optimization features" and we elaborate them below:

– **Dynamic Order Adjustment.** This feature adjusts the order of suggested words dynamically according to user's input history. For example, as shown in Fig. 3, two typed characters "ba" lead to different lists of suggested words. "bankruptcy" is the first suggestion in the upper picture while "banquet" is the first suggestion in the lower one.

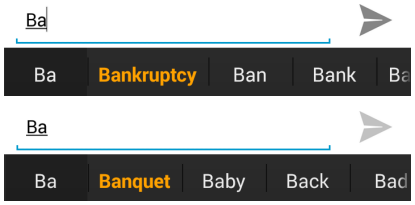

Fig. 3. Dynamic order adjustment

– **Contact Names Suggestion.** IME can suggest a name from user's phone contact when part of the name is typed. In addition, suggestions also pop up when an unknown name is typed for correction. The `READ_CONTACTS` permission needs to be granted to support this feature.

– **Next-word Prediction.** IME attempts to predict the next word user wants to input based on the previous words typed. Fig. 4 shows an example that IME gives a prediction "Newcastle" based on the previous input "Fly to".

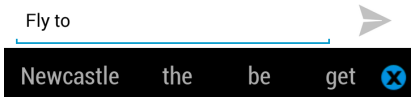

Fig. 4. Next-word prediction

– **New Word Saving.** When a word not existing in the dictionary is typed, IME automatically adds this word to its dictionary.

To summarize, all the above features are driven by user's personalized information, like user's input history. Furthermore, when the permissions shielding user's sensitive data are granted, IMEs can customize their dictionaries using various data sources, including SMS, Emails, and even social network data. It is very likely that the names of user's family members and friends and nearby locations are recorded by the IME after using for a while. We manually examined the settings and permissions of several IMEs and summarizes the data sources

Table 1. Data sources of mainstream IMEs for optimization features

Production Name	Version	Input History	Contacts	Emails / SMS	Social Network	Location
Go Keyboard [7]	2.18	√	√			
TouchPal [21]	5.6	√	√	√	√	
Adaptxt - Trial [1]	3.1	√	√	√	√	√
Google Keyboard [8]	4.0	√	√	√	√	
SwiftKey Keyboard [18]	5.0	√	√	√	√	
Swype Keyboard Free [19]	1.6	√	√		√	
Fleksy Keyboard Free [5]	3.3	√	√	√	√	
Google Pinyin Input [9]	4.0	√	√			
Sogou Mobile IME [16]	7.1	√	√			√
Baidu IME [3]	5.1	√	√			
QQ IME [14]	4.7	√	√			

of mainstream IMEs (each of them has over 1 million installations) in Table 1. Apparently, the personalized dictionary should be considered private assets and protected in the safe vault. In fact, most of the IME apps we surveyed keep their dictionaries in the internal storage of mobile phone which is only accessible to the owner app.

2.2 Adversary Model

The adversary we envision here is interested in the dictionary entries of IME deemed private to the user, like contact names, and aims to steal and exfiltrate them to her side. We assume the user has installed a victim IME which is "**benign**" and "**smart**".

1. "Benign" means this IME exercises due diligence in protecting user's private data. The measures taken include keeping its dictionary in app's private folder (internal storage). This assumption differs fundamentally from previous IME-based attacks which assume IME itself is malicious.
2. "Smart" means this IME can learn unique word-using habits and build a personalized user dictionary based on user's input history, contacts, etc.

At the same time, we assume this user has downloaded and installed a malicious app named *DicThief* aiming to steal entries from victim IME. The default (enabled) IME on the device could be identified through the system class `Settings.Secure` [15]. This malware only claims two permissions: `INTERNET` and `WAKE_LOCK`. Both permissions are widely claimed by legitimate apps and unlikely to be rejected by users: nearly all apps declare the `INTERNET` permission, and `WAKE_LOCK` is also widely claimed by apps like alarm, instant messenger (e.g., WhatsApp and Facebook Messenger), etc. With the `WAKE_LOCK` permission, our attack can be launched **when the phone is in sleep mode and locked with password**.

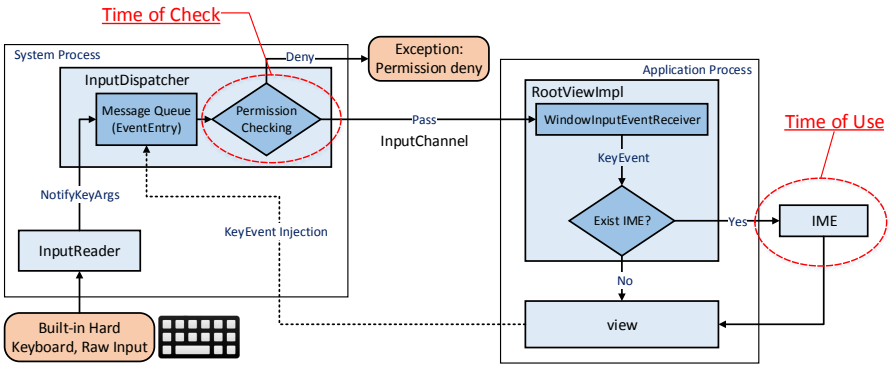

Fig. 5. Android KeyEvent processing framework and CAKI vulnerability

3 Vulnerability Analysis

While direct access to the dictionary of IME is prohibited if coming from a different and non-system app, our study shows this security guarantee can be violated. By revisiting the keystroke processing framework of Android, we discover a new vulnerability lying under Android OS, allowing us to launch **C**ross-**A**pp **K**eyEvent **I**njection (CAKI) attack. In essence, by exploiting such vulnerability, a malicious app can simulate user keystrokes on an IME, and read the suggested words. Below we describe the mechanism of keystroke processing in Android and the new vulnerability we identified.

3.1 Android KeyEvent Processing Flow

The internal mechanism of input processing in Android is quite sophisticated and here we only overview how KeyEvents[4] are processed. At a high level, when a key is pressed on hardware (built-in) keyboard, a sequence of KeyEvents will be generated by wrapping the raw input, and then sent to the corresponding handlers (e.g., IME). These two steps are called *KeyEvent pre-processing* and *KeyEvent dispatching*. We illustrate the process in Fig. 5 and then elaborate the details below[5]:

KeyEvent Pre-processing. As soon as a hardware key (e.g., built-in keyboard) is pressed, a raw input event is sent to a system thread `InputReader` (initiated by `WindowManagerService`) and encapsulated into an object of type `NotifyKeyArgs`. Then, this object is passed to thread `InputDispacher` (also initiated by `WindowManagerService`) and a `KeyEntry` object is generated. Lastly, this object is converted to `EventEntry` object and posted to the message queue

[4] IME accepts another kind of input event – MotionEvent [12], coming from soft keyboard (see Fig. 1). Its processing flow is different and not covered in this paper.

[5] Our descriptions are based on Android 4.4.4_r2.0.1 [2]. For other versions, the flows are mostly the same. Only the paths of source code could be different.

(`InboundQueue`) of `InputDispacher` to be distributed to right handlers. If a key press is simulated by an app, the corresponding KeyEvents are initiated directly and finally sent to the message queue (in the format of `EventEntry`).

KeyEvent Dispatching. Before event dispatching, there is a permission checking process on the corresponding `EventEntry` to ensure its legitimacy. The code undertaking such check is shown below (excerpted from the Android code repository [24]):

```
bool checkInjectionPermission(...) {
  if (injectionState
     && (windowHandle == NULL
          || windowHandle->getInfo()->ownerUid !=
             injectionState->injectorUid)
     && !hasInjectionPermission(injectionState->
        injectorPid, injectionState->injectorUid)) {
       ...   // Code omitted due to space limit
       return false; // Permission denied
  }
  return true; // Pass checking
}
```

This routine first verifies whether the event is generated by a hardware device (checking `injectionState`). If `injectionState` is `NULL`, the check is passed and the event is directly sent to handlers. Otherwise (the event is generated by an app), this routine verifies whether the sender app owns the required permission or if it is identical to the receiver app (we fill in the details in Section 3.2).

An input event passing the above check will be dispatched via a system IPC mechanism `InputChannel` to the receiver app, which should run in the foreground and take the input focus. In particular, the `WindowInputEventReceiver` component belonging to the receiver app process will be notified and then forward the received KeyEvent to other components of `ViewRootImpl`, a native OS class handling GUI updates and input event processing, for further processing. When an IME is activated and also brought to the foreground (see Fig. 1), there exists a special policy: `ViewRootImpl` will redirect the KeyEvent to the IME, which absorbs the event and renders the resulting text (suggested word or the raw character) on the client app's view. This policy guarantees the KeyEvents are processed by IME with high priority.

3.2 Cross-App KeyEvent Injection Vulnerability

Since the simulated key-presses could come from a malicious app, Android enforces much stricter checking. Still, the checking routine is not flawless. Below, we elaborate a critical vulnerability in this routine:

KeyEvent Injection. An app can simulate various input events using the APIs provided by Android instrumentation library [10]. This is supposed to support automated app testing. For example, an app can invoke the function

`Instrumentation.sendKeyDownUpSync()` to simulate user's keystrokes, such as "d", "7", "!", and the corresponding KeyEvents will be injected into the message queue subsequently.

Verification. Injected KeyEvent needs to be vetted. Otherwise, one app can easily manipulate the input to the app taking focus. If a KeyEvent is not originated from hardware keyboard, at least one of the security checks has to be passed (see the code block of `checkInjectionPermission` in Section 3.1):

1. The KeyEvent injector and receiver are the same.
2. The KeyEvent injector is granted with the `INJECT_EVENTS` permission.

As `INJECT_EVENTS` is a system-level permission, a non-system-level app (installed by user) simulating key-press has to meet the other requirement: the receiver app is itself.

CAKI Vulnerability. At first glance, the above verification process is sound. However, it fails when IME is in the picture, and as such, a malicious app can launch **Cross-App KeyEvent Injection (CAKI)** attack.

A non-system-level malicious app (named app_x) running in the foreground first activates IME (named IME_y) set as default by user, which could be achieved by setting the focus on an `EditText` widget [4] in app_x's window. After IME_y is ready and its virtual keyboard is displayed, app_x injects a KeyEvent to the message queue. At this point (Time of Check), the KeyEvent receiver is also app_x as it takes input focus (another party, IME, cannot take focus by design). The projected event flow turns out to be $\{app_x \rightarrow system \rightarrow app_x\}$ and clearly passes the check of routine `checkInjectionPermission`. Then (Time of Use), the KeyEvent is sent to `RootViewImpl` of app_x . Given IME_y is activated at this moment, this KeyEvent is redirected to IME_y (see Section 3.1), turning the actual event flow into $\{app_x \rightarrow system \rightarrow \texttt{RootViewImpl}\ of\ app_x \rightarrow IME_y\}$. In this stage, no additional checks are enforced and IME_y will respond to the KeyEvent. Obviously, the security guarantee is violated because app_x and IME_y are not identical. This vulnerability allows a malicious app to send arbitrary KeyEvents to IME.

This CAKI vulnerability can be attributed to a big class of software bugs, namely **time-of-check to time-of-use (TOCTTOU)** [31,44,41,39]. However, we are among the first to report such bugs in Android platform[6] and our exploitation showcase indicates this CAKI vulnerability could cause serious consequences.

4 Attack

In this section, we describe the design and implementation of the proof-of-concept app *DicThief*, which exploits the CAKI vulnerability and steals dictionary entries.

[6] We found only one vulnerability disclosure by Palo Alto Networks' researchers [42] regarding TOCTTOU in Android, which was reported in March 2015.

After DicThief is run by the victim user, it starts to flood KeyEvents to an `EditText` widget which pops up the default IME when the owner app DicThief is in the foreground. IME will commit words to the `EditText` and they are captured by DicThief. When the number of stolen entries hit the threshold, DicThief will stop flooding KeyEvents and exfiltrate the result (compressed if necessary) to attacker's server. Since KeyEvent injection has been discussed in the previous section, here we elaborate how to harvest meaningful entries from the dictionary and our context inference technique in making the attack stealthy.

4.1 Enumerating Entries from Dictionary

Given the huge size of IME dictionary (hundreds of thousands of words), the biggest challenge is how to identify the entries comprehending user's private information efficiently. These entries could be added from user's typed words, imported from user's private data (e.g., contact names) or reordered according to user's type-in history. We refer to such entries as *private entries* here. Through manually testing several popular IME apps, we observed one important insight regarding these private entries: they usually show up after 2 or 3 letters/words typed and they are placed in 1st or 2nd position in the suggestion list. In other words, by enumerating a small number of letter/word combinations, a large number of private entries can be obtained. We design two attack modes based on such insight:

- **Attack Mode 1 – Word Completion:** For each round, DicThief injects 2 or 3 letters and then injects the space key or number "1" to obtain the first word from the suggestion list of IME, which is then appended to the list of collected results. After all the valid combinations are exhausted or the threshold is reached, the list is sent to attacker's server. This attack works based on the dynamic order adjustment feature of IME: e.g., if a user frequently chooses "bankruptcy" from suggestion list, when she types "ba", the suggestion list will become {bankruptcy | ban | bank | bad}, and the private entry can be easily determined.
- **Attack Mode 2 – Next-word Prediction:** This time, DicThief injects a complete word (or several words) for each round and selects the first word prompted by IME. Similarly, the space key or number "1" is used to obtain the first suggestion, and the attack ends when a certain number of rounds is reached. This attack exploits IME's next-word prediction feature: e.g., the injected words "fly to" will trigger the list {Newcastle | the | be | get} if IME concludes that "Newcastle" is user's favorite choice.

The generated list comprehends both private entries and the entries irrelevant to customization. We need to filter out the latter ones. To this end, we carry out a differential analysis. We run DicThief against a freshly installed IME app which has not been used by anyone and compile all the first words in two modes. Next, we find the different words between the list collected from victim's phone with ours. The words left are deemed private entries. This procedure runs on

attacker's server, but it can be executed on victim's phone instead to save the bandwidth.

4.2 Attack in Stealthy Mode

When DicThief is launched, it has to be displayed and run in the foreground in order to turn on IME. If user is using the phone at the same time, the malicious activities will be noticed easily. In this section, we propose several techniques to reduce the risks of notice.

Context Inference. DicThief is designed to run when user falls asleep. At that time, the phone should be placed on a flat platform (no acceleration observed), the screen should be dimmed, and the time should be at night. All of these information can be retrieved using APIs from system classes (`SensorManager` for accelerator metrics, `PowerManager` for screen status and `Calendar` for current time respectively) without any permission. These techniques are also exploited by other works [28,36] to make their attacks stealthy.

When DicThief is opened by user, it stays in the background and is periodically awakened to infer the running context. DicThief will not start to inject key-presses until the current context meets the attack criteria.

Circumventing Lock Screen. Our attack has to be executed even if the phone is asleep and locked with password. To proceed the attack, DicThief requires the `WAKE_LOCK` permission being granted first. As discussed in Section 2.2, user will not reject such request in most cases. Besides, DicThief needs to add the `FLAG_SHOW_WHEN_LOCKED` flag in its window setting, making it take precedence over other lock screens [22].

Yet, common apps will not be brought to the top of foreground when phone is locked. Each app has a corresponding object `WindowState`, which stores `Z`-order regarding its order of layer in display. The window with the bigger `Z`-order will be shown in a higher layer. A general app window is set to 2 while key guard window (lock screen) is set to 13, therefore, key guard window will always display in front of other general apps. `WindowState` is managed by `WindowManagerService` and `Z`-order cannot be tweaked by app. Nevertheless, when an app invokes an IME, it will be brought to the top of the client app disregarding its assigned `Z`-order due to one policy of Android [25]. Hence, our attack can succeed even when the screen is securely locked with password.

4.3 Case Study of IMEs for Non-Latin Languages

Not only is our attack effective against IMEs for English, IMEs for non-Latin languages are vulnerable as well. Apart from English users, the users who type in non-Latin words have to rely on alternative IMEs since the language characters are not directly mapped to English keys. In this section, we demonstrate a case study on attacking Chinese IMEs. It turns out just a few adjustments need to be applied to the enumeration algorithm and private entries can be effectively obtained, albeit the complexity of such language.

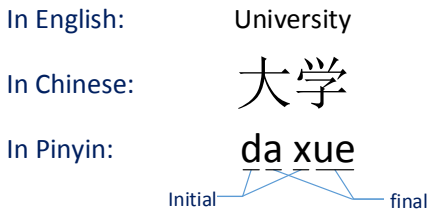

Fig. 6. Example of Chinese Pinyin

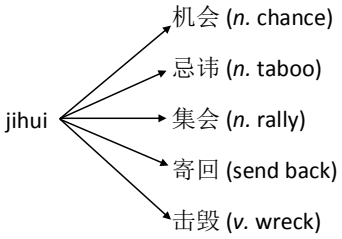

Fig. 7. One-to-many Mapping

Chinese and Pinyin. Pinyin is the official phonetic system for transcribing the Mandarin pronunciations of Chinese characters into the Latin alphabet [13]. Pinyin-based IMEs are, in fact, the most popular IMEs used by Chinese users. Except for some special cases, every Chinese syllable can be spelled with exactly one *initial* followed by one *final* [13,23]. In total, there are 23 initials[7] and 37 finals. Fig. 6 describes an example.

Each Chinese character has a unique syllable, but one syllable is associated with many distinct characters. Each Chinese word is composed of multiple characters (usually two to three). An example is shown in Fig. 7. The character combination poses a big challenge in harvesting meaningful Chinese entries: a prefix (e.g., "ji") might only reveal one Chinese character, far from meaningful words. On the other side, a prefix in English (e.g., "mis") can yield the the list of meaningful words with viable size.

Attack. Fortunately, Pinyin-based IME optimizes the input experience. By providing several syllable initials, the suggestion list of words with the same initials will be produced. For instance, typing "j'h" (initial j plus initial h) will yield the list of 5 Chinese words shown in Fig. 7. It motivates us to enumerate the combination of initials instead of the leading Pinyin letters. Here, we show the algorithm of attacking word-completion mode of Pinyin-based IME in Algorithm 1.

Algorithm 1: Enumerating 2-character words of Pinyin-based IMEs

```
1 for Key_1=InitialSet.first; Key_1<=InitialSet.last; Key_1=Key_1.next() do
2     for Key_2=InitialSet.first; Key_2<=InitialSet.last; Key_2=Key_2.next() do
3         injectKeyEvent(Key_1) ;        // initial of the first character
4         injectKeyEvent(APOSTROPHE) ;          // divide two characters
5         injectKeyEvent(Key_2) ;      // initial of the second character
6         injectKeyEvent(KEYCODE_SPACE) ;        // commit the suggestion
7     end
8 end
```

[7] The initial set: $\{w, y, b, p, m, f, d, t, n, l, g, k, h, j, q, x, zh, ch, sh, r, z, c, s\}$.

Table 3. Evaluation against IMEs

Production Name	Version	Language(s)	Vulnerable	Installations
Go Keyboard [7]	2.18	Multi-language	Yes	50,000,000+
TouchPal [21]	5.6	Multi-language	Yes	10,000,000+
SwiftKey Keyboard [18]	5.0	Multi-language	Yes	10,000,000+
Adaptxt - Trial [1]	3.1	Multi-language	Yes	1,000,000+
Google Pinyin Input [9]	4.0	Chinese, English	Yes	10,000,000+
Sogou Mobile IME [16,17]	7.1	Chinese, English	Yes	200,000,000+
Baidu IME [3]	5.1	Chinese, English	Yes	1,000,000+
QQ IME [14]	4.7	Chinese, English	Yes	1,000,000+
Swype Keyboard Free [19]	1.6	Multi-language	No	1,000,000+
Fleksy Keyboard Free [5]	3.3	Multi-language	No	1,000,000+
Google Keyboard [8]	4.0	Multi-language	No	100,000,000+

5 Evaluation

We analyzed the scope of attacks (the vulnerable Android versions and IMEs) and evaluated the effectiveness of the two attack modes described in Section 4.1.

5.1 Scope of Attack

The CAKI vulnerability discovered in this paper derives from the design flaw of Android framework. Thus, in theory, all Android devices should suffer from this vulnerability. We examined 7 different versions of Android OS on 4 physical Android phones and 2 Android images on an emulator, and it turns out all versions ranging from very old (2.3.7) to the latest (5.0) are vulnerable without exception. The list of vulnerable phones and OS versions is shown in Table 2.

Table 2. Evaluation against Android OSes

Phone Model	Android Version	Attack Result
Nexus 6 (Genymotion Emulator [6])	AOSP Android 5.0	success
Sony Xperia Z3	Sony official 4.4.4	success
Samsung Galaxy S3	CyanogenMod 4.4.4	success
	Samsung official 4.3	success
Meizu MX2	Meizu official 4.2.1	success
Sony Xperia ion	Sony official 4.1.2	success
Nexus S (Genymotion Emulator)	AOSP Android 2.3.7	success

Also, our attack is not limited to a specific language or a specific IME. All smart IMEs equipped with optimization features should be potentially vulnerable. We tested our attack on 11 popular IMEs and 8 among them are vulnerable, as shown in Table 3. Our attack does not succeed on 3 IMEs because they only respond to taps on soft keyboard, but ignore the key-presses simulated by app. These IMEs, however, may have compatibility issues since hardware keyboard is not supported well. We suspect such lucky escape is probably due to design flaw rather than protection enforced.

5.2 Experiment on Word Completion Attack Mode

In this mode, DicThief injects 2 or 3 random letters and selects the first word suggested by IME. The victim IME we chose is Sogou Mobile IME [16], a domi-

Table 4. User study – word completion attack mode

User	Age	Gender	Installation Time	Feeling of Info Leakage	Category of Info Leakage	Personalized Entries	% of Personalization
1	25∼30	male	1 year+	Many	①②③⑤	416	78.6%
2	25∼30	male	8 months+	Many	① ②	239	45.2%
3	25∼30	male	1 year+	Some	② ⑤	358	67.7%
4	18∼25	male	2 months+	Many	② ③ ⑤	107	20.2%
5	18∼25	male	2 months+	Some	② ⑤	436	82.4%

nant Pinyin-based IME in China with 200 million monthly active users [17]. The information leakage and overhead caused by DicThief are assessed separately:

Information leakage. We conducted a user study[8] to portrait and quantify the leaked information. We recruited 5 Sogou Mobile IME users (labeled as $User_1$ – $User_5$) to participate in our experiments. All of them are native Chinese speakers (the mother tongue of $User_5$ is Cantonese, which is a dialect of Chinese). Their basic information and the final results are shown in Table 4.

All the participants installed a modified version of DicThief on their phones before the experiment. All 2-initial combinations are probed, counting up to 529 rounds (23×23 combinations, see Section 4.3). To address the privacy concerns of our human subjects, we did not collect any word entries from their phones. Instead, we asked them to report the type and quantity of personalized entries (calculated by DicThief). The detailed result is presented in two aspects:

1. **Intuition: Severity of Leaked Information.** DictThief shows the extracted words to the volunteers directly after the attack finishes and then the volunteers are asked to fill a survey. Questions include: how much sensitive information are extracted [“*Many*” /“*Some*” /“*None*”]? which categories can be used to summarize the leaked information [①“*Occupation*”/ ②“*Contacts*”/ ③“*Location*”/ ④“*Hobby*”/ ⑤“*Other personalized information*”]? The result is shown in the 5th & 6th columns of Table 4.
2. **Quantification: Percentage of Personalized Entries.** For each volunteer, MD5 for all extracted words (529 entries total) are generated and compared with the MD5 of extracted words from the IME freshly installed. Personalized entry is counted if discrepancy identified. The result is shown in the last two columns of Table 4.

Apparently, a plenty of sensitive information will be leaked if the CAKI vulnerability is exploited by real attackers. On average, 58.8% of the words extracted are indeed personalized. Besides, all volunteers report that contact names are listed in the result, which are definitely sensitive to users.

Time and Battery Consumption. The time spent for KeyEvent injection is negligible, but DicThief has to pause for a while after a round of key injection till the IME renders its UI. The actual time overhead depends on the implementation of IME apps and the performance of phone's hardware. We measure it on

[8] The experiments have followed the IRB rules, and all human subjects fully understood the privacy implication of the experiments and agreed to participate.

Table 5. Simulation experiment – next-word prediction attack mode

Sample	Crawled words	Author Info & Blog Topics	Personalized Entries	% of Personalization	Blog URL
1	31581	female, professor, work experience	273	63.3%	http://ge***hd.blogspot.com/
2	31661	male, cooking, food	39	9.0 %	http://ff***od.blogspot.com/
3	35606	male, American football	73	17.0 %	http://fo***og.blogspot.com/
4	40913	male, personal life	54	12.5 %	http://li***gy.blogspot.com/
5	32347	female, traveling	208	48.3 %	http://www.st***ls.com/

Samsung Galaxy S3 and set the waiting period to 70 ms based on manual testing a priori. The total time consumed adds up to 221 s for all 2-initial combinations injections against Sogou Mobile IME. Meanwhile, the battery consumption is also slim, costing less than 1% of total battery life. The whole attack process will hardly be detected by victim user if DicThief runs under the right context.

5.3 Experiment on Next-word Prediction Attack Mode

In this mode, DicThief injects one or more words and choose the first word from the list of predictions provided by IME. The IME evaluated is TouchPal [21], an English IME with over 10 million installations worldwide. Since it is hard for us to recruit enough native English speakers as volunteers in our region, we decided to use public web resources to create virtual user profiles and customize IME dictionary with them. It brings the extra benefit that now we are allowed to look into what exact private entries are leaked. We document the steps for generating user profiles as below:

1. In a real-world scenario, an IME is customized by the text a user inputs or information left by the user. Likewise, for each virtual user, we compile the text she could enter and dump it to IME. In all, we create 5 users (labeled as $Sample_1$ – $Sample_5$) and use content scraped from 5 blogs to externalize them separately. The blogs are carefully chosen so that the topic focused on by each one is different. Table 5 shows statistics of the prepared data.
2. Since TouchPal is able to read messages and customize itself, we dump the collected blog content into the SMS sent box of the test phone (Samsung Galaxy S3) using an Android app developed by ourselves. We use one paragraph to fill one text message.
3. Now, TouchPal can proceed to customize its dictionary. We tick the options "*Learn messages automatically*" and "*Only learn sent messages*", and run the "*Learn from messages*" function of TouchPal. It takes one hour on average for the customization process to end.

When a predicted word is selected, TouchPal will prompt a new predicted word. Hence, a user can type one word and continuously choose the words provided by

TouchPal to build a long phrase. We leverage this feature to carry out 3-level prediction attack. For example, DicThief injects one word "want" and then chooses three predicted words – "to", "go" and "shopping" – prompted consecutively (through injecting number "1"). A meaningful phrase "want to go shopping" will be revealed. Our empirical study suggests that starting from a verb, we have higher chances to capture a meaningful phrase. Therefore, we select 431 words from 1000 frequently used English verbs [20] (from "is" to "wrap") to bootstrap our attack. The remaining 569 words are not selected as they are either other tenses of the selected verbs or largely used as nouns.

Information Leakage. We followed the same leakage quantification method used in the last experiment and the result is shown in Table 5. Since the data comes from virtual users, we take a close look at the personalized words this time. We use $Sample_1$ as an example and show its leaked information in the list below. (Another example is described in Appendix).

Table 6. Examples of private entries – $Sample_1$

– know what to do	→	know I always advise
– want to go to	→	want to publish an
– become a better place	→	become a professional editor
– relate to the gym and	→	relate to the New York
– create a new one	→	create a gmail account
– invest in a few days	→	invest in a recent talk
– rely on the way	→	rely on the tenure
– buy a new one	→	buy a new university
– wish I could have	→	wish I could write

For a sequence of injected keys, we compare the phrase generated from fresh IME (left-side of "→") and $Sample_1$'s IME (right-side of "→") (see Table 6). One can easily find words related to $Sample_1$'s occupation, such as "tenure", "professional editor", "advise", "university" and "recent talk". Expectedly, the privacy of a real-world user will be under severe threat if such attack is launched.

Time and Battery Consumption. The time overhead is also small. Injection of single key costs 35 ms (counting waiting time) and the whole attack takes around 401 s. The battery consumption is also negligible (less than 1%).

6 Defense

Our reported attack exploits a critical vulnerability of Android KeyEvent processing framework as the security checks fail to cover the complete execution path. However, it is not a trivial task to fix this vulnerability due to the highly sophisticated design of Android. Adding a new permission to limit such behavior is unhelpful. Injecting KeyEvent to the app itself should be permitted as usual for the purpose of automated testing unless IME is involved in the process. Yet, there is no way to ensure this when app installation. Simply modifying IME app code and rejecting all the injected KeyEvents is not a viable solution either, as the injections from system-level apps owning the `INJECT_EVENTS` permission should be allowed.

To mitigate such threat, we propose to augment the current KeyEvent processing framework. Currently, the information about KeyEvent sender is limited.

It only tells whether KeyEvent is injected by one app or coming from hardware-keyboard, turning out to be too coarse-grained. We argue that the identity of the source app (i.e., package name, signature) should be enclosed in KeyEvent as well, which can be fulfilled by adding a new field to its data structure. Before a KeyEvent is dispatched, Android OS automatically attaches the sender's identity to it. Prior to forwarding KeyEvents to IME, Android OS verifies the sender and discard the injected KeyEvents if the sender is neither system app owning the `INJECT_EVENTS` permission nor hardware-keyboard. Attaching origin has also been explored by Wang et al. [40] to prevent one app from sending unauthorized intents to another app in Android. Their approach requires modifications to Android OS and app's code, and the policy setting process is delegated to the app side. In contrast, our approach only calls for modification to Android OS, as the policy should be identical to all IME apps, which protects them transparently.

Meanwhile, we examine other possible countermeasures, but they all come with the loss of usability or compatibility. One possible solution is to prohibit IME being invoked when the phone is securely locked, but this will disable the quick-reply feature of the default SMS app and third-party IM apps. We can also force IME to commit words to text controls only if the word displayed on touch screen is tapped, which, however, will block the input from hardware keyboard.

7 Related Works

IME Security Issues. All user typed text can be collected by IMEs, and user's privacy will be breached if an IME sends out the collected key presses out of malice. In fact, there have been questionable behaviors of IMEs observed in the wild [35,34]. Suenaga [38] and Mohsen et al. [30] also studied key-logging threats of malicious IMEs on Windows and Android platform respectively. In this work, we identify a totally different venue to abuse IME: rather than enticing users to install malicious IME, an adversary is able to exfiltrate the sensitive information through a new system design flaw and a novel IME probing technique.

Key-logging Attacks. A non-system app on Android cannot obtain keystrokes directly. However, previous works show that it is possible to infer keystrokes through various side-channels. A touch on the phone surface, especially the soft keyboard will cause vibrations and touching on different positions will introduce distinctive vibration patterns. Previous works monitor the motion sensor like accelerometers to collect vibration statistics and infer what keys are pressed [27,43,33,26]. Besides, other sources are also exploited for keystrokes inference, including sound collected by microphone [32] and video camera [37]. On the other hand, our work steals user's input history in plain text and a large amount of typed text can be unveiled in a short time.

Untrusted Input. A plethora of key functionalities in mobile devices are driven by user's input and the modules handling such data are usually entailed with very high privileges. A natural path for a malicious app to elevate its privileges is impersonating human and injecting false input. Diao et al. [28] discovered that

an adversary can inject prerecorded voice commands to the built-in voice assistant module (Google Voice Search) of Android and bypass permission checks. Jang et al. [29] investigated accessibility (a11y) support framework of popular desktop and mobile platforms and identified a number of system vulnerabilities in handling user's input. In this work, we identify a new channel to inject fake input and bypass permission checks. We believe this type of threats is not yet over and encourage future research in identifying other exploitable sources and building better input validation mechanisms.

8 Conclusion

In this paper, we identify a new cross-app KeyEvent injection vulnerability against IMEs installed on Android devices. By exploiting such flaw, an adversary can infer words frequently used by a user or coming from other sensitive sources. We implement DicThief, a prototype app and evaluate it under real-world settings. The result shows that all Android versions and most of popular IMEs are vulnerable, putting a large amount of users into danger. Such issue should be fixed immediately and we propose a solution only requiring changes to Android system. In the end, we believe this vulnerability is only the tip of the iceberg, and the security of input processing framework and IME needs to be further studied.

Acknowledgments. We thank anonymous reviewers for their insightful comments. This work was supported in part by Internal Grants C001-4055006 and C001-2050398 from The Chinese University of Hong Kong.

Appendix: Additional Data for Experiments on Next-word Prediction Attack Mode

For $Sample_5$, several terms related to traveling could be found. They reflect what topics the author often types. Especially, they point to a location – "Utah" and a type of transportation – "Sky train" which is the rapid transit railroad system operating in Bangkok, Thailand. After examining the blog content of $Sample_5$, we found the location was visited and the transportation was boarded before.

Table 7. Examples of private entries – $Sample_5$

– see you soon then	→	see the stars and
– supply of the day	→	supply of our trip
– prepare for the first	→	prepare for the flight
– intend to do it again	→	intend to do in Utah
– behave like a good	→	behave like a packing
– rest of the day	→	rest of the city
– use the bathroom and	→	use the Sky train
– travel to the gym	→	travel tips for the
– search for the first	→	search for the flight

References

1. Adaptxt - Trial. https://play.google.com/store/apps/details?id=com.kpt.adaptxt.beta
2. Android Open Source Project: android-4.4.4_r2.0.1. https://android.googlesource.com/platform/frameworks/base/+/android-4.4.4_r2.0.1
3. Baidu IME. https://play.google.com/store/apps/details?id=com.baidu.input
4. EditText. http://developer.android.com/reference/android/widget/EditText.html
5. Fleksy Keyboard Free. https://play.google.com/store/apps/details?id=com.syntellia.fleksy.kb
6. Genymotion. http://www.genymotion.com/
7. GO Keyboard. https://play.google.com/store/apps/details?id=com.jb.gokeyboard
8. Google Keyboard. https://play.google.com/store/apps/details?id=com.google.android.inputmethod.latin
9. Google Pinyin Input. https://play.google.com/store/apps/details?id=com.google.android.inputmethod.pinyin
10. Instrumentation. http://developer.android.com/reference/android/app/Instrumentation.html
11. KeyEvent. http://developer.android.com/reference/android/view/KeyEvent.html
12. MotionEvent. https://developer.android.com/reference/android/view/MotionEvent.html
13. Pinyin. http://en.wikipedia.org/wiki/Pinyin
14. QQ IME. https://play.google.com/store/apps/details?id=com.tencent.qqpinyin
15. Settings.Secure. http://developer.android.com/reference/android/provider/Settings.Secure.html
16. Sogou Mobile IME. https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou
17. SOUHU.COM Annual Report. http://mfiles.sohu.com/corp/2013%20Annual%20Report.pdf
18. SwiftKey Keyboard. https://play.google.com/store/apps/details?id=com.touchtype.swiftkey
19. Swype Keyboard Free. https://play.google.com/store/apps/details?id=com.nuance.swype.trial
20. Top 1000 Verbs. http://www.talkenglish.com/Vocabulary/Top-1000-Verbs.aspx
21. TouchPal. https://play.google.com/store/apps/details?id=com.cootek.smartinputv5
22. WindowManager.LayoutParams. http://developer.android.com/reference/android/view/WindowManager.LayoutParams.html
23. ISO 7098:1991 Romanization of Chinese. ISO/TC 46 Information and Documentation (1991)
24. Android Open Source Project: InputDispatcher.cpp. https://android.googlesource.com/platform/frameworks/base/+/android-4.4.4_r2.0.1/services/input/InputDispatcher.cpp

25. Android Open Source Project: PhoneWindowManager.java. https://android.googlesource.com/platform/frameworks/base/+/android-4.4.4_r2.0.1/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java
26. Aviv, A.J., Sapp, B., Blaze, M., Smith, J.M.: Practicality of Accelerometer Side Channels on Smartphones. In: Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC) (2012)
27. Cai, L., Chen, H.: TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In: Proceedings of the 6th USENIX Workshop on Hot Topics in Security (HotSec) (2011)
28. Diao, W., Liu, X., Zhou, Z., Zhang, K.: Your Voice Assistant is Mine: How to Abuse Speakers to Steal Information and Control Your Phone. In: Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM) (2014)
29. Jang, Y., Song, C., Chung, S.P., Wang, T., Lee, W.: A11y Attacks: Exploiting Accessibility in Operating Systems. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS) (2014)
30. Mohsen, F., Shehab, M.: Android Keylogging Threat. In: Proceedings of the 9th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom) (2013)
31. Mulliner, C., Michéle, B.: Read It Twice! A Mass-Storage-Based TOCTTOU Attack. In: Proceedings of the 6th USENIX Workshop on Offensive Technologies, (WOOT) (2012)
32. Narain, S., Sanatinia, A., Noubir, G.: Single-stroke Language-Agnostic Keylogging using Stereo-Microphones and Domain Specific Machine Learning. In: Proceedings of the 2014 ACM conference on Security and Privacy in Wireless and Mobile Networks (WiSec) (2014)
33. Owusu, E., Han, J., Das, S., Perrig, A., Zhang, J.: ACCessory: Password Inference using Accelerometers on Smartphones. In: Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications (HotMobile) (2012)
34. Rowe, I.: Chrome OS to warn users of privacy risks in alternate keyboard layouts. http://www.linuxveda.com/2014/06/20/chrome-os-warn-users-privacy-risks-alternate-keyboard-layouts/ (June 2014)
35. Sanders, J.: Japanese government warns Baidu IME is spying on users. http://www.techrepublic.com/blog/asian-technology/japanese-government-warns-baidu-ime-is-spying-on-users/ (January 2014)
36. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: Proceedings of the 18th Network and Distributed System Security Symposium (NDSS) (2011)
37. Simon, L., Anderson, R.: PIN Skimmer: Inferring PINs Through The Camera and Microphone. In: Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM) (2013)
38. Suenaga, M.: IME as a Possible Keylogger. Virus Bulletin November, 6–10 (2005)
39. Tsafrir, D., Hertz, T., Wagner, D., Silva, D.D.: Portably Solving File TOCTTOU Races with Hardness Amplification. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST) (2008)
40. Wang, R., Xing, L., Wang, X., Chen, S.: Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In: Proceedings of the 20th ACM Conference on Computer and Communications Security, (CCS) (2013)

41. Wei, J., Pu, C.: TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In: Proceedings of the FAST '05 Conference on File and Storage Technologies, (FAST) (2005)
42. Xu, Z.: Android Installer Hijacking Vulnerability Could Expose Android Users to Malware. http://researchcenter.paloaltonetworks.com/2015/03/android-installer-hijacking-vulnerability-could-expose-android-users-to-malware/ (2015)
43. Xu, Z., Bai, K., Zhu, S.: TapLogger: Inferring User Inputs On Smartphone Touchscreens Using On-board Motion Sensors. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks (WiSec) (2012)
44. Yang, J., Cui, A., Stolfo, S.J., Sethumadhavan, S.: Concurrency Attacks. In: Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism, (HotPar) (2012)