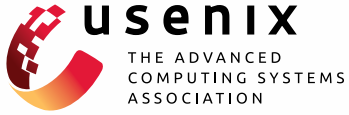


You Shouldn't Collect My Secrets: Thwarting Sensitive Keystroke Leakage in Mobile IME Apps

Jin Chen and Haibo Chen, ***Shanghai Jiao Tong University;*** **Erick Bauman and Zhiqiang Lin,** ***The University of Texas at Dallas;*** **Binyu Zang and Haibing Guan,** ***Shanghai Jiao Tong University***

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/chen-jin

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

You Shouldn't Collect My Secrets: Thwarting Sensitive Keystroke Leakage in Mobile IME Apps

Jin Chen[†], Haibo Chen[†], Erick Bauman[⋆], Zhiqiang Lin[⋆], Binyu Zang[†], Haibing Guan[†]
[†]Shanghai Key Kaboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
[⋆]Department of Computer Science, The University of Texas at Dallas

ABSTRACT

IME (input method editor) apps are the primary means of interaction on mobile touch screen devices and thus are usually granted with access to a wealth of private user input. In order to understand the (in)security of mobile IME apps, this paper first performs a systematic study and uncovers that many IME apps may (intentionally or unintentionally) leak users' sensitive data to the outside world (mainly due to the incentives of improving the user's experience). To thwart the threat of sensitive information leakage while retaining the benefits of an improved user experience, this paper then proposes I-BOX, an app-transparent oblivious sandbox that minimizes sensitive input leakage by confining untrusted IME apps to predefined security policies. Several key challenges have to be addressed due to the proprietary and closed-source nature of most IME apps and the fact that an IME app can arbitrarily store and transform user input before sending it out. By designing system-level transactional execution, I-BOX works seamlessly and transparently with IME apps. Specifically, I-BOX first checkpoints an IME app's state before the first keystroke of an input, monitors and analyzes the user's input, and rolls back the state to the checkpoint if it detects the potential danger that sensitive input may be leaked. A proof of concept I-BOX prototype has been built for Android and tested with a set of popular IME apps. Experimental results show that I-BOX is able to thwart the leakage of sensitive input for untrusted IME apps, while incurring very small runtime overhead and little impact on user experience.

1 INTRODUCTION

The Problem. With large touch screens, modern mobile devices typically feature software keyboards to allow users to enter text input. This is different compared to traditional desktops where we use the hardware keyboards. These soft keyboards are known as Input Method Editor (IME) apps, and they convert users' touch events to text. Since IME apps process almost all of a user's input in mobile devices, it is critical to ensure that they are not keyloggers and they do not leak any sensitive input to the outside world.

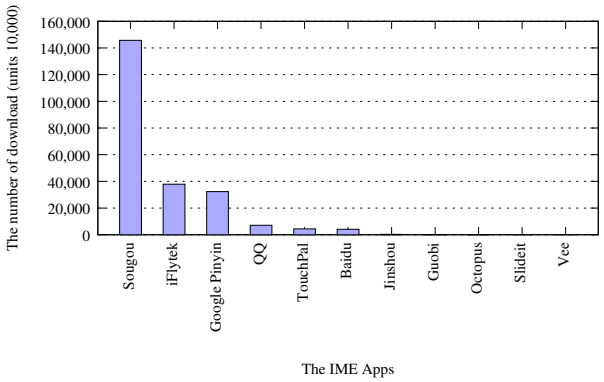


Figure 1: Download statistics of IME apps in our study.

While all mobile devices have a default IME app installed, users often demand third-party IME apps with expanded feature sets in order to gain a better user experience. This is especially common for non-Latin languages. In order to accommodate this need, mobile operating systems such as Android and iOS provide an extensible framework allowing alternate input methods. Due to the ease of making third-party IME apps and high demand for customization, there are currently thousands of IME apps in major App market like Google Play and Apple's App Store. Many of which have gained hundreds of millions downloads, as shown in Fig. 1. For instance, the Sogou IME apps has in total 1.6 billion downloads in Google Play and several third party app vendors such as 360, and Baidu. Meanwhile, a recent survey [13] found that 68.3% of smartphones in China are using third-party IME apps. This survey did not include statistics from Japan or Korea, where such apps are also very popular.

Unfortunately, despite these advantages, using a third-party IME app also brings security and privacy concerns (assume the default IME app does not have these problems). First, IME app developers have incentives to log and collect user input in order to improve the user's experience with their products, and user input is as valuable as email content, from which they can learn user's needs and push customized advertising or other business activities. Although an IME app may state a policy of not collecting certain input from a user, the policies imple-

mented in the app may unintentionally send sensitive input outside the phone. In §2.3 we show that such a threat is real by observing the output of a popular IME app that periodically sends out user input to a remote server. In addition, we collected the network activities of a set of IME apps during a user input study and showed that they also likely send out private data. In light of this information leakage threat, the Japanese government's National Information Security Center has warned its central government ministries, agencies, research institutions and public universities to stop using IME apps offered by the search engine provider Baidu [1].

Even if a user trusts benign IME apps to properly secure private data, there is still a risk from repackaging attacks targeting benign apps. In fact, prior study has shown that around 86% of Android malware samples are repackaged from legitimate apps [49]. It is also surprisingly simple to repackage an IME app with a malicious payload, as we demonstrate in §2. Essentially, a repackaged malicious IME app is essentially a keylogger, which has been one of the most dangerous security threats for years [39]. Also, evidence has shown that IME apps are popular for attackers to inject malicious code [29].

Challenges. While it may seem trivial to detect these repackaged malicious IME apps by comparing a hash of the code with the corresponding vendor in the official market, the widespread existence of third-party markets makes such checks more difficult. It is also easy for attackers to plant repackaged malware into these markets, as is shown by the fact that a considerable amount of repackaged malware has been found in them [48].

Of further concern is the fact that it is very challenging to analyze whether even "benign" IME apps will leak any sensitive data or not. There are several reasons why detecting privacy leaks in IME apps is challenging. First, many commercial IME apps use excessive amounts of native code, which makes it very difficult to understand how they log and process user input. Second, many of the IME apps use unknown, proprietary protocols, which makes it especially hard to analyze how they collect and transform user input. Third, many of them utilize encryption, and their algorithms are also unknown. Therefore, we eventually must treat the IME apps as black boxes for current privacy-preserving techniques on mobile devices, and users must either trust them completely (and risk leaking their private data) or switch to the default IME app (and lose the improved user experience).

At a high level, it would seem that existing techniques such as taint tracking would be viable approaches to precisely tracking and containing sensitive input. For example, TaintDroid [16, 17] and its follow-up work have been shown to very effective to track sensitive input and detect when it is leaked. There will still be the following additional challenges to be overcome. First, current IME apps tend to use excessive native code in their core logic, and TaintDroid currently does not track tainted data in native code. Second, it is a well-known problem that data-flow based tracking for taint-tracking systems to capture control-based propagation. In fact, many of the keystrokes are generated through lookup tables, as reported in Panorama [46]. Third, sensitive information is often composed of a sequence of keystrokes, making it challenging to have a well-defined policy to differentiate between sensitive and non-sensitive keystrokes in TaintDroid. Therefore, we must look for new techniques.

Our approach. In this paper, we present I-BOX, an app-oblivious IME sandbox that prevents IME apps from leaking sensitive user input. In light of the opaque nature of third-party IME apps, the key idea of I-BOX is to make an IME app oblivious to sensitive input by running IME apps transactionally; I-BOX eliminates sensitive data from untrusted IME apps when there is sensitive input during this process. Specifically, I-BOX checkpoints the states of an IME app before an input transaction. It then analyzes the user's input data using a policy engine to detect whether sensitive input is flowing into an IME app. If so, I-BOX rolls back the IME app's states to the saved checkpoint, which essentially makes an IME app oblivious to what a user has entered. Otherwise, I-BOX commits the input transaction by discarding the checkpoint, which enables the IME app to leverage users' input to improve the user experience.

One key challenge faced when building I-BOX is how to make the checkpointing process efficient and consistent, which is unfortunately complicated by Android's design, especially its hybrid execution (of Java and C), multi-threading, and complex IPC mechanism (e.g., Binder). Fortunately, I-BOX addresses this challenge by leveraging the *event-driven* nature of an IME app. More specifically, we present a novel approach by creating the checkpoint at a *quiescent point*, in which its execution states are inactive. Such a design significantly simplifies many issues such as handling residual states in the local stack of native code, the Dalvik VM and IPCs.

We have implemented I-BOX based on Android 4.2.2 running on a Samsung Galaxy Nexus smartphone. Performance evaluations show that I-BOX can checkpoint and restore a set of third-party popular IME apps within a very tiny amount of time, and thus cause little impact on user experience. A security evaluation using a set of popular IME apps shows that I-BOX mitigates the leakage of sensitive input. Case studies using a popular "benign" IME app and a repackaged IME app confirm that I-BOX accurately conforms to the predefined security policies to prevent sending of sensitive input data.

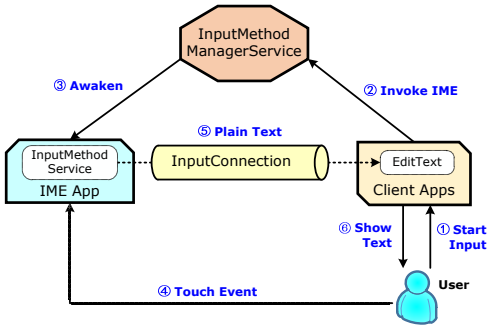


Figure 2: The workflow when using an IME app.

Contributions. In short, we make the following contributions:

- **New Problem**. This is the first attempt to systematically understand the threat caused by the leakage of private sensitive keystrokes in third-party IME apps. Our discovery shows the pervasive presence of such attacks, and the seriousness of the problem.
- **New Technique**. We introduce oblivious sandboxing for IME apps that embraces both security and usability and *quiescent points* based checkpoint/restore that significantly simplifies the design and implementation of I-BOX.
- **New System**. We demonstrate a working prototype of the techniques and a set of evaluations confirming the security threat of commercial IME apps and the effectiveness of I-BOX.

2 BACKGROUND AND MOTIVATION

In this section, we first describe the necessary background on IME architecture in Android, and then discuss why commercial IME apps have the incentive to collect a user's data, followed by the case studies showing how IME apps can leak users' sensitive data to remote parties.

2.1 Input Method Editor

Though Android provides a default IME app for each language, many end users prefer using third-party IME apps for better user experiences, such as changing the screen layout for faster input, generating personalized phrases to provide intelligently associational input, and providing more accurate translation from keystrokes to the target languages. As a result, mobile operating systems such as Android provide an extensible IME infrastructure to allow third-party vendors to develop their own IME apps.

Figure 2 gives an overview of the involved IME components when entering text in a client app. Specifically, third-party IME apps must conform to the IME framework so that the Android Input Method Management Service (IMMS) can recognize and manage them. For example, every IME app contains a class that extends from `InputMethodService`, which helps Android recognize it as an input service and add it into the system as an IME app. When an end user clicks a textbox to invoke an IME app, Android IMMS will start the default IME activity and build an `InputConnection` between the IME app and the client app that helps the IME app to commit the user input to the client app. In particular, the IME app first gets the touch event containing the position data and translates it to meaningful characters or words based on its keyboard layout and internal logic. Then it sends the keystrokes to the client app through `InputConnection`.

The IME architecture is clean with well-defined classes. This not only significantly saves programmer's effort in developing a new IME app, but also makes it easy for attackers to locate key points of a victim IME app. For instance, our study found that simply hooking the function `BaseInputConnection.commitText` can intercept all the user's input in many IME apps. This can be done by simply searching for the keyword `BaseInputConnection.commitText` in the decompiled code to locate all of its occurrences.

2.2 Why IME Apps Collect Users' Input

Third-party IME apps usually extend the standard IME apps with lots of rich features to provide a better user experience. Such features usually require collecting users' input data to learn users' habits to allow personalizing IME apps. Further, such data may also collectively be used to improve experiences of other users, i.e., pushing phrases learned from a set of users to others. In fact, there are many features that require collecting user input data. The following lists a few of them:

- **Personal dictionary**. Commercial IME apps usually remember the words and phrases from user input to speed up follow-up input (especially for non-Latin languages) by prompting potential results when input is not finished. To achieve this, they need to maintain a personal dictionary for each user to save frequently typed or self-made words.
- **Cloud input**. As users usually have multiple devices and need to synchronize personal dictionary among them, IME apps utilize cloud-based services to store the dictionary and to synchronize the dictionary and personal settings between different devices.

  Meanwhile, some non-Latin languages such as those eastern languages differ from English in that IMEs need to translate users' keystrokes to words in those languages. To accelerate input speed, IMEs

may usually need to leverage cloud services to analyze and predict users' intended words based on the current input.

In addition, for some latin-based languages, some IME apps provide a feature that leverages the current input to predict the intended phrases and adjust the layout of the soft keyboard to make the soft key of the next character close to users' current figure. To better predict user intent, some IME apps usually leverage the abundant resources in cloud to analyze and predict user input. Meanwhile, they also collect users' habits to improve the accuracy of prediction.

- **Search mediation**. Some IME apps have a new feature named "search mediation", which intercepts user input and returns some search result back to the user. However, this means that user inputs will be unrestrictedly sent to the search engine.

Note that due to the unstable network connectivity of mobile devices, almost all IME apps can work properly with and without network connections. When network is disconnected, an IME app may store current input (like frequently used phrases) for later use when the network connection is on. Besides, Android's configurable permission model indicates that an IME app usually works normally even without grants of certain permissions.

2.3 Possible Threats Posed by IME Apps

While third-party IME apps do offer useful features and better user experiences, they may unduly collect user data or be repackaged to be malicious. Next, we study the possible threats an IME app could impose.

Privacy leakage in "benign" IME apps. Conventional wisdom is to trust a respected service provider, in the hope that the provider will enforce policies in the cloud to faithfully provide user secrecy [30]. Unfortunately, this exposes users' sensitive keystrokes from two threats. First, a curious or malicious operator may stealthily steal such data [47, 41], which has been evidenced by numerous insider data theft incidents even from reputed companies [40]. Second, even reputed cloud providers provide no guarantee on the security of user data, which is evidenced by their user agreements. Hence, it is reasonable to not trust an IME app to securely protect users' data.

More specifically, a severe threat from "benign" IME apps is that they may have unduly collected user data without users' awareness. Given that we do not have their source code and they often use proprietary protocols with encryption, it thus remains opaque to end users how the IME apps really handle the sensitive input data. At a high level, since they have been collecting user data for better experiences (especially the personal dictionary and cloud input), it is highly likely that much of a user's sensitive input has been leaked to these IME providers.

To confirm our hypothesis, we conducted an experimental study by performing a man-in-the-middle attack on a popular IME app, namely TouchPal Keyboard (in version chubao 5.5.5.67049, cootek). This IME app provides multiple rich functionalities such as cloud input and a personal dictionary and has been installed more than 7.09 million times from a third-party market. By intercepting its network packages using Wireshark[1], we found that its cloud input is implemented using an HTTP POST command which carries several parameters in plain text. Therefore, we are able to see how it works without any protocol reverse engineering and packet decryption. A deep investigation revealed that these parameters include a `userid`, the `keycode` that a user just entered, and *the existing words* of the target input control that user is focusing on. This contradicts its privacy statement of "No collection of personal information that you type" in a prior statement[2], and thus poses a serious threat to user privacy.

We suspect there may be many other commercial IME apps that also leak users' sensitive input. Currently, we only used side-channel analysis [11] to analyze the packet size between the IME apps and their servers. We did notice there are notable differences in the number of packets (as reported in §5.2).

Privacy leakage in malicious IME apps. Even if all third-party IME apps did not leak any user's private data, there are still other attack vectors such as repackaging attacks. In fact, a prior study uncovers that repackaged malware samples account for 86% of all malware [49]. Moreover, there are also trojans that serve as key loggers but masquerade as IME apps [29]. Finally, IME apps may also be vulnerable to component-hijacking attacks. It has been shown that input methods have been a popular means to inject malicious code [29]. While currently we are not aware of any repackaged malicious IME apps in Android, we envision that there will be such malware given the large popularity of the official apps and the easiness of repackaging them as shown below.

To understand the repackaging threat of IME apps, we conducted an attack study by repackaging a popular commercial IME app called Baidu IME, which has been downloaded more than 100 million times in a third-party market. In this study, we repackage the IME app by inserting a malicious payload into the original program. The payload records all user input and sends them to a specific server.

While the core logic of the Baidu IME app is written

[1] http://www.wireshark.org/

[2] We noted that the newer versions of TouchPal changed their privacy statement indicating that they will collect user privacy data.

using C, the other components are written in Java which enables an easy reverse engineering of the bytecode especially with existing tools. Specifically, we used baksmali [2], a popular Dalvik disassembler to reverse `classes.dex` into an intermediate representation in the form of smali files. Then we directly modified smali code to insert our payload, which captures the text committed by the function `BaseInputConnection.commitText` and then sends the data out. A caveat in this study is that we found it would not work if we simply repackaged the app because the IME app has a checksum protection. However, the protection mechanism is rather simple, as it just calls a self-crash function when detecting repackaging. However, the self-crash function is not self-protected and thus we rewrote it to return directly to disable the protection.

We conducted our experiment in a contained environment and did not upload this repackaged IME app to any third-party Android market, but attackers can easily do this, as reported before [49, 48]. We installed this repackaged IME app on our test smartphone and all data we input through it was divulged. Our attack study shows all critical data that a user inputs will be compromised if the IME app is malicious. The popularity of third-party markets aggravates this problem, especially considering that 5% to 13% of apps are repackaged in a number of third-party markets [48].

3 OVERVIEW

The goal of I-BOX is to protect users' sensitive input, while still preserving the usability of (curious or malicious) IME apps such that users can still benefit from the rich features. One possible approach might be letting users switch to a trusted IME app when they want to type some sensitive information. While this may work for simple sensitive data like passwords, some users' sensitive input (like addresses and diseases) is scattered in a long conversation. It is cumbersome for users to constantly keep this in mind and do the switch. Another intuitive approach would be to block all network connections during user input, but doing so will negatively affect the user experience. Besides, there are also other channels like third-party content providers and external storages that an IME app may temporally store input data to be leaked later. Therefore, we have to look for new approaches.

Approach overview. As discussed, the key challenges of securely using third-party IME apps are that such apps are usually closed-source and they may do arbitrary processing and transformation of users' input data before sending it out. It is thus hard to model or predict their behavior. Hence, I-BOX instead treats an IME app as a black box and makes it *oblivious* to users' sensitive input data. To achieve this, I-BOX borrows the idea from execution transactions by running an IME app *transactionally*. Consequently, if an IME app touches users' sensitive input data, I-BOX will roll back the IME app's states to make it oblivious to what it has observed so as to address the problem where an IME app stores and transforms users' input data.

I-BOX 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. A clean snapshot of an IME app will be saved before an input transaction starts. For normal input transactions without touching sensitive input data, I-BOX will commit the IME app's state such that the IME app can use these data to improve the user experience. To prevent malicious IME apps from sending private data out during the input transaction, the network connection of the IME app will be restricted when the current transaction is marked as sensitive. When an input session ends and thus the client app has received all user input, I-BOX will abort the input transaction from the view of the IME app, by restoring the IME app's state to a most-recent checkpoint. This makes the IME app oblivious to the sensitive data it observed. Hence, even if the IME app locally saves a user's input to be sent later, the input data will be swiped during restoring.

As input data is provided in a streaming fashion by a user, there is no general way to know the input stream in advance. Because the IME app gets the input data prior to I-BOX, it would be too late to stop an IME app's leaking channels like network connection after it gets the whole input since it may have sent it out or store it locally. Hence, it is generally impossible for an approach not leaking any user input before I-BOX can determine if the current input stream is sensitive or not.

As a result, I-BOX chooses to use a combination of context-based and policy-driven approaches based on the state of the IME app, with the goal of striking a balance between user experience and privacy. For specific input such as passwords, which I-BOX can determine through input context, I-BOX can immediately know they are sensitive and thus constrains IME app's behavior (like blocking networking for the app). For general input, I-BOX uses a state-machine based policy engine to predict whether the current input transaction is sensitive. This is done continuously during the input process, where I-BOX uses the current partial input stream to determine if the next string is sensitive or not.

An architectural overview of I-BOX is presented in Figure 3. I-BOX consists of an isolated user-level policy engine that decides whether I-BOX shall commit or roll back the execution of an IME app's state. The sandbox module is implemented as a kernel module, which saves and restores the states of an IME app as needed.

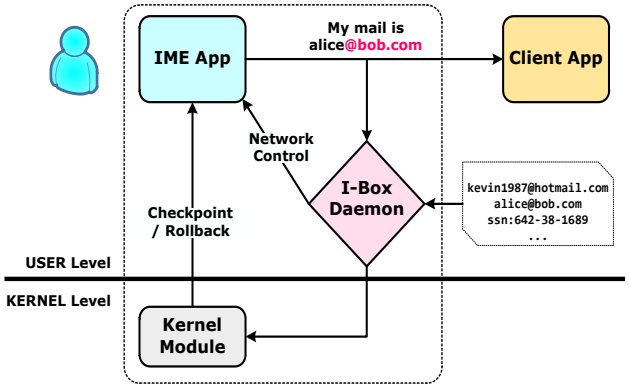


Figure 3: An Architectural Overview of I-BOX

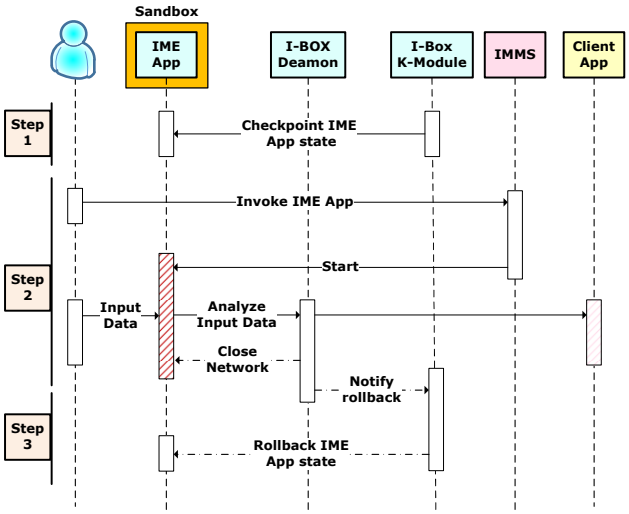


Figure 4: I-BOX work flow.

Challenges. To realize I-BOX, we are facing several challenges. In particular:

- **How to express and enforce security policies?** As users' privacy policies are usually vague, it is critical to efficiently represent users' policies such that there won't be a state explosion problem. This is especially challenging to handle for non-Latin languages as they usually require an additional layer of translation to represent them. Further, once the policies are represented, it should also be relatively easy to check the current input against the policies, which is critical to the latency of the checking.

- **How to efficiently perform the checkpoint and rollback?** As checkpoint and rollback are triggered during input, lengthy checkpoint and rollback may extend the latency of users' input. However, traditional checkpoint and rollback usually require either expensive copying of applications' states, or heavyweight recording of applications' execution. For example, prior checkpointing on server platforms takes around 600ms without copying files [28].

- **How to ensure consistency upon rollback?** By considering the user's input process as a transaction, I-BOX can ignore the implementation details of different IME apps and take them as normal processes from the kernel's viewpoint. However, there also intensive cross-layer and cross-component interactions between an IME app and the rest of the environment, like the Dalvik VM, the application framework and the client app. Further, the IME app is essentially multi-threaded. Hence, consistently checkpointing and rolling back an IME app's states while preserving the states of other components is another key technical challenge for I-BOX.

Threat model and assumptions. As third-party IME apps have the incentive to collect and send out users' data and some IME apps are even malicious (repackaged or even faked), I-BOX considers all third-party IME apps as untrusted. However, I-BOX trusts the underlying smartphone OS, including the OS kernel, system services and any process with root or system privileges. Also, we assume the user's smartphone has not been rooted such that the untrusted IME app cannot break the default security isolation between different apps, especially for system and user-level apps.

I-BOX relies on input contexts and a user's policy to distinguish private data from normal input data. It is possible that I-BOX may leak sensitive user input if the policy is incomplete or inaccurate, or the user's intent has changed after specifying the policy. Further, depending on the state machine, I-BOX may leak a prefix of some sensitive input.

I-BOX also trusts the end user and rely on her as a witness to prevent a malicious IME app from tampering with the user's input during typing. This should be easy as she can tell the difference between what she typed and what she observed from the input screen.

We consider the client app that uses the services from an IME app as trusted. While a rogue or malicious client app may also steal users' sensitive input, a malicious IME app causes more security impact than a malicious client app as it leaks all user input to all client apps (including system apps) in contrast to only input to a specific (third-party) client app. How to protect third-party client apps is out of the scope of this work and many prior efforts have intensively studied solutions to prevent information leakage from apps [24, 51, 34].

4 DESIGN AND IMPLEMENTATION

The work flow of how I-BOX works is illustrated in Figure 4. Specifically, I-BOX intercepts a user's input data by placing hooks into Android Input Method Service (IMS) and detects the sensitive data from the input stream based on the policy engine. I-BOX uses both context-based and prefix-matching policies (§4.1) and enforces them using transactional execution (§4.2) to protect sensitive data such as passwords. Before diving into the details how we design and implement I-BOX, we first use a running example to illustrate how it really works.

A running example. Assuming a sensitive string "IsUsenixSec2015" is being typed by a user through an IME app, I-BOX first makes a checkpoint of the IME app as a clean snapshot before input. If this string is being typed to a password textbox (*context-based policy*), I-BOX immediately knows that the string to type is sensitive and will restrict the IME app's behavior (such as stopping network connections). Otherwise, I-BOX intercepts the characters and runs the analysis through the policy engine. After getting the characters 'I', 's', 'U', and 's', I-BOX predicts that the user may be typing the sensitive string "IsUsenixSec2015" and I-BOX restricts the IME app's behavior immediately to prevent it from sending further keystrokes out (*prefix-matching policy*). Afterwards, the IME app continues to accept input from users' typing and I-BOX monitors the file operations of the IME app to record the files that may log the input data. After the user finishes typing, I-BOX confirms that a sensitive string was typed into the IME app and restores the states of the IME app with the checkpoint to clean the sensitive string out.

4.1 Policy Engine

The policy engine of I-BOX separates sensitive input from normal input such that different policies can be applied to different types of data. I-BOX uses both context-based and prefix matching strategies to derive policies, with the first strategy having higher priority.

Context-based policy. We first provide an automated approach to deriving which input would be sensitive based on the type of the input and execution context of an app. Specifically, Android uses text fields to help the user type text into client apps. Text fields can have different input types, such as numbers, dates, passwords, or email addresses. In fact, the type information of text fields in the client app has been used to help an IME app to optimize its layout for frequently used characters. I-BOX also leverages the type information of the text fields to decide whether the input is sensitive or not, and passwords and email addresses are by default sensitive. In addition, based on the user defined per-client app policy (e.g., an IME app is providing services to a banking application), I-BOX will automatically treat all the input consumed by a sensitive app according to context [44] as sensitive.

Prefix-matching policy. For general input streams, I-BOX leverages prefix matching to distinguish which input stream is sensitive or not. One challenge for defining policies for I-BOX is that IME apps may need to handle multiple languages, including both Latin languages and non-Latin languages. For non-Latin languages, I-BOX can only get the text in the target languages after an IME app has translated the keystrokes for the corresponding text. Hence, it is not viable to simply use keystrokes to represent the current input. To address this problem, I-BOX instead uses the UTF-8 (8-bit Unicode Transformation Format) of the translated keystrokes to represent current keystrokes as well as those in the policy engine.

As there may eventually be multiple data instances that should be considered as sensitive, I-BOX uses a trie-like structure to maintain which data should be considered as sensitive. A trie-like structure is very space-efficient for data with a common prefix and is very efficient for look-up. I-BOX maintains a global trie structure to represent the global policy. I-BOX may also provide an application-specific trie structure if an end user demands more strict policy. During a query, I-BOX queries the global and application-specific trie structures in parallel but prefers application-specific policies over the global one.

While much of the sensitive data like contacts and cookies can be automatically translated to the trie structure as the default policies, I-BOX also allows end users to use regular-expressions when they manually specify the policy. For example, user may define "abc*" to indicate any word starting with "abc" as sensitive input. Associated with the regular expression, there is also an *acceptable disclosure rate* (ADR), which defines how many characters can be exposed in an input stream. The larger the ADR, the more information may be leaked but the more chances are allowed for cloud assistance. Using regular expression is easy for experienced users to specify sensitive input, as it does not require them to fully remember all such sensitive data and thus matches users' ambiguous and incomplete memory. This also avoids asking users to input full secrets to I-BOX. Alternatively, average users may also specify full secret names (i.e., a special case of regular expression) to I-BOX.

I-BOX provides a simple script to add such regular-expressions to the trie-like structure and report any conflicts if they occur. For example, for a sensitive string of 15 characters (such as 'IsUsenixSec2015') and an ADR of 0.2, I-BOX will restrict an IME app's behavior when

the first three characters ('IsU') are typed. I-BOX runs the trie-structure as a state machine to predict the input stream by matching the typed characters with the trie structures. Since any substring in the input data may be sensitive, I-BOX needs to check all of them. To speed up this process, I-BOX searches all possible substrings when a new character is typed. Intermediate states are maintained so that only new characters need to be handled instead of new substrings constructed by the character.

Note that currently I-BOX directly searches over the plain text of the policy file and relies on the Android permission system to protect it for simplicity. This can be further enhanced by encrypting the policy file and using regular expressions to search over the encrypted file, which was shown to have small runtime and space overhead [36].

Prefix-substitution attacks. At first glance, the prefix-matching policy used by I-BOX would appear to be vulnerable to a prefix-substitution attack by a malicious IME App. Specifically, a malicious IME app might first replace the prefix of a typed string with a non-sensitive one so that I-BOX wouldn't recognize this prefix and thus no oblivious sandbox would be applied for this input session. Fortunately, we note that users, the ultimate *witness*, would immediately notice this by observing the difference between what they typed and what was displayed on the screen.

Note that as I-BOX monitors all keystrokes sent from IME apps to user apps, I-BOX will adjust the state machine accordingly for any cursor movement and special characters like deletion. This can detect the case where a malicious IME app stealthily moves the cursor to deceive I-BOX on the input sensitivity.

Overall, I-BOX requires users' awareness of what she types from what she observes to detect malicious behavior from an IME app. If a user does not pay enough attention to the input process, a malicious IME app may still have the chance to fool I-BOX about the sensitivity of the input streams.

4.2 Enabling Transactional Execution

To enable transactional execution of an IME app, I-BOX needs to provide a checkpoint and rollback mechanism. The key challenges here lie in how to provide *low-latency* and ensure *consistency*, which are made especially difficult by Android's unique design. For example, Android uses a Dalvik virtual machine (VM) to run the Java code of the IME app, which interacts intensively with the application framework. Further, the native code of an IME app also interacts with the Dalvik VM through the Java Native Interface (JNI). Finally, Android intensively uses Binder, a complex IPC mechanism for communication among isolated apps. Such hybrid execution and complex communication make it hard to efficiently and consistently checkpoint the states of an IME app.

I-BOX addresses the above challenges by leveraging a set of *quiescent points*. A quiescent point is a point such that all threads of an application have stopped execution and there are no pending states and requests to be processed. Doing checkpointing at quiescent points frees I-BOX from handling a number of subtle states like residual states in stack or other communication peers. Further, it also requires less states to be checkpointed. Finally, when I-BOX rolls back the states of an IME app, the states can be restored consistently without having to deal with some subtle residual states in other apps.

In the following, we describe in greater detail how we choose the quiescent points (§4.2.1), how I-BOX performs the checkpoint and restore of the local states of an IME app (§4.2.2), and how I-BOX handles interactions of an IME app with others through IPCs (§4.2.3).

4.2.1 Quiescent Points

Our key observation is that an IME app is essentially an event-driven app that provides services to the client app. Consequently, it shall be usually in a quiescent point when a user is not typing, as no event will be delivered to the IME app at that time. At this state, the IME app's states are stable and consistent. Thus, I-BOX can be relaxed from handling a lot of complex and subtle local states. To achieve this, I-BOX first checks if an IME app is in a quiescent point by checking the process and thread states (sleeping or not) and the IPC states. The checking result is very likely to be true for most cases. Even if the IME app refuses to cooperate with I-BOX and keeps itself busy, I-BOX can first wait a short time and then enforce a quiescent point by blocking new requests and then forcing the IME app to sleep to do the checkpoint. Here, a non-cooperative IME app could also be a sign of being malicious. However, we never encountered this case as the IME apps we tested always conform to Android IME architecture. Even if so, I-BOX may always roll back the IME app to a clean state checkpointed early.

4.2.2 Checkpointing and Restoring Local States

Since data typed by a user can be stored into any place of the IME app in any form, it requires that all process states restore in order to wipe out any sensitive data. The traditional way of doing checkpoints is copying all related process states into storages, which is very heavyweight and would incur long latency. As the main purpose of I-BOX's checkpoint is to either rollback or discard later, I-BOX chooses a lightweight approach to checkpointing, which creates a shadow process and then tracks all later changes by using copy-on-write (COW) features provided by Linux.

Saving and restoring file states. As typical IME apps usually only modify a small amount of files during one input transaction, I-BOX currently records and copies such files during checkpointing and restores them during rollback for simplicity. Another option is using a COW file system like Btrfs or ext3cow to avoid copying. This requires replacing Android's file system with one with a COW feature, which will be our future work.

Android provides several options to save persistent application data. Based on the position where the data is stored, we can divide these options into two categories: internal and external storage. Every Android app will be assigned with a private directory in the internal storage to store files and data. By default, data saved to the internal storage are private to an app and other apps cannot access them (nor can the user). I-BOX just copies all files in the IME app's private directory and then restores the files modified during the input transaction upon rollback. Since there are usually only a small number of files in the internal storage for an IME app and the modified ones are even less, the time cost is negligible.

For external storage, any Android apps with proper permissions (e.g., `android.permission.WRITE_EXTERNAL_STORAGE`) can access the whole external storage. It would be very lengthy if I-BOX scanned the whole external storage to find the modified files. Hence, I-BOX records all the files modified by the IME app during the input transaction and then restores them as needed. Specifically, once I-BOX detects the IME app tries to write some data into a file, it duplicates the file for subsequent restoring.

Note that as the checkpointed files are created by I-BOX, which runs as a system process, the files are with system privilege and thus cannot be read/written by the IME app itself. This ensures that an IME app cannot first save sensitive key logs into such files and later read them out. Actually, I-BOX also removes the checkpointed files after rolling back an IME app.

Saving and restoring memory states. Memory states include the IME app process's data in memory and process-related metadata maintained by the OS kernel (i.e., Linux). Linux uses a lot of data structures to manage a process and maintain its state, such as `task_struct`, `thread_info` and others. I-BOX relies on a kernel module to save and restore such data structures. Specifically, this module maintains a shadow process in the kernel to store the data of each running IME app. The shadow process duplicates the process states of the original IME app by copying the metadata of the IME app into its own `task_struct` but with some modifications for consistency. For example, it has its own kernel stack and redirects the stack pointer in the `task_struct` to its own one, although the content on the stack is the same as in the original IME app. For independent states like process ID or kernel stack, I-BOX just copies the data into a buffer and writes them back later. As for other states connected with other processes or other events like a pipe or waitqueue, I-BOX needs to record the states and the relationships so that it can recover it correctly later. Besides this, I-BOX also needs to save the process memory. Instead of really copying the memory pages, I-BOX simply creates a shadow page table that shares the memory with the target IME app process and marks the page table of the target process as COW. This omits lots of unnecessary page copying since most pages will not be modified during the input transaction and it just needs to switch the page table root to restore the memory, which is very fast. This helps reduce the stop-time of each IME app process when I-BOX tries to do checkpoint and restore.

Multi-thread rollback. Most Android applications run in the Dalvik virtual machine and have multiple threads for different purposes. Besides the main thread for UI and the core logic of the IME app, there are about another 10 threads for garbage collection, event handling, Binder IPC, and so on. To roll back the process states of an IME app correctly, I-BOX needs to deal with such threads properly. Linux assigns `task_struct` to a thread just like a process to maintain its state and groups all threads belonging to one process together through a list. So I-BOX saves each thread with a separated shadow process and groups these processes together through a list to maintain their parent-child relationships just like the original one. The sharing resources between threads will be duplicated too. For example, I-BOX will save the pipe states between two threads and restore it later.

4.2.3 Handling IPCs

One major challenge I-BOX faces in checkpoint and rollback is how to deal with the IPC states of an IME app process. An IPC involves multiple processes or even multiple machines, but I-BOX can only control one end in the communication. One potential problem is that the other side of an IPC may wait for a reply that will never be sent, since the IME app process has forgotten this request after rollback. Another serious problem is that the client app may communicate with an inactive IPC that has been erased from the IME app process due to rollback.

As a result, I-BOX needs to find proper timing to do checkpoint and rollback such that the consistency of an IPC is not violated. Proper timing requires several conditions. First, there should not be any data in transmission between two processes; otherwise it will lead to a corrupted request with incorrect semantics. Second, there should be no pending IPC requests. This means an IME

app shall wait for all replies before doing checkpoint and ensure that no request is pending to the process before rollback. Fortunately, it is not hard for I-BOX to find suitable timing because I-BOX only does checkpoint and rollback when a user does not input. In most cases, the IME app processes should be sleeping at that point. If not, we can safely enforce it without disturbing other client apps since the user is not typing.

Inter-threads IPC. Linux provides a set of IPC mechanisms such as pipe, socket, and shared memory. Android inherits such mechanisms but only uses them as a method for communication between threads within a single process. Hence, I-BOX can control both ends of these inter-threads IPC, which avoids the inconsistency issues due to unilateral actions. For example, the two communicating parties of a pipe in a single process have a pair of pipe `fd`; the OS kernel allocates a buffer for them to pass the message. To restore the pipe correctly, I-BOX just keeps a record of current pipe status and its buffered data, then restores it as needed. There is no restriction on the timing for checkpoint and rollback. Other IPCs within the same process are done similarly to this.

Android Binder. Android heavily uses its own IPC mechanism: *Binder*, which helps the Android permission system to provide access control to Android services and resources. By mapping kernel memory into user space, Binder IPC only requires one data copy for one transmission, i.e., from the sender's user space to the kernel buffer of the Binder driver. Then the receiver can directly read the data from its read-only user space mapping, which is more performance-friendly. There are two issues I-BOX needs to take care for a consistent restore of the Binder. More specifically:

- **Reference counting for Binder proxies**. An Android app uses Binder proxies (e.g., BBinder, BpBinder) as the reference to remote processes instead of simple file descriptors. The Binder driver in the kernel needs to manage the reference counter for such proxies so that it can know whether a binder instance is useless or not. I-BOX needs to track and record modifications to references to Binder proxies so that it can keep the consistency of the reference counters.
- **Conversation between the Binder request and response**. I-BOX also needs to keep the conversation between the Binder transaction request and response. As an Android service provider, an IME app process will accept a Binder transaction request from the client app and it will send back the transaction response after disposing the request. To achieve this, I-BOX tracks the transaction request and response to find a right timing when all requests have been handled. It is not hard to find such a point because usually I-BOX tries to do checkpoint or rollback when IME is idle without new requests.

Content Provider. An IME app may also interact with both third-party and system content providers. For example, our analysis with TouchPal IME app reveals that this app accesses third-party content providers like `content://com.tencent.mm.sdk.plugin.provider/sharedpref` and `content://com.facebook.katana.provider.AttributionIdProvider`; our analysis with Guobi IME app shows that this app accesses `content://com.iflytek.speechcloud.providers.LocalResourceProvider` and `content://com.tencent.mm.sdk.plugin.provider/sharedpref`. TouchPal accesses the system content provider like `content://sms/inbox` and both TouchPal and Guobi access `content://telephony/carriers/preferapn`. In Android, all requests to content providers are issued through the Binder mechanism, we rely on the Binder mechanism to detect a quiescent point. Fortunately, we note that accesses to content providers are request-oriented and thus connection-less. Thus, there is no request on-the-fly and thus I-BOX can checkpoint such states accordingly.

Network. Different from Binder, the network driver does not expose any semantic information to an upper layer's connections. Hence, it seems hard to maintain the consistency of request and response between an IME app process and its cloud-based server. Fortunately, there are two observations that help relax the strict consistency requirement. First, network connections between an IME app and the cloud-based server, like fetching the words by sending the keystrokes, synchronizing the user's library, and downloading news or advertisements, are usually stateless and non-transactional; a redo operation does not cause any consistency issues. Second, network connections during input transactions are mostly short-time synchronized requests that are finished when input is done; hence they will not be affected by rollback.

Lessons Learned. While it is generally hard to checkpoint a complex app like an IME app, the event-driven nature of I-BOX greatly helps simplify the design and implementation of I-BOX. By leveraging a quiescence-point based approach and conduct checkpointing at the time at which an IME app likely to be quiescent (e.g., before an input session start), I-BOX enjoys both less implementation complexity and runtime overhead.

4.3 Restricting IME Apps' Behavior

When I-BOX detects a sensitive input session, it needs to restrict an IME app's behavior such that no sensitive data

should be leaked during this process. A malicious IME app may leverage various means to store and transform the data during this process. For example, it may directly send input data to the network, or store the input data to a content provider to be restored and sent out later. To this end, I-BOX needs to restrict an IME app's behavior to stop such channels for a sensitive input stream.

I-BOX constrains an IME app from using network and accesses to content provider and services during a sensitive input session. Specifically, during a sensitive input session, I-BOX only grants an IME app with read accesses (like query) to such content providers and services. This is done by interposing the *binder_transaction* and acts according to the access types from the transaction code (i.e., query, insert, update or delete).

One potential issue would be that the IME app may not function correctly without such accesses. Fortunately, most Android apps (including IME apps) are designed to work gracefully with different permissions, due to the fact that the user may grant different permissions and an IME app may work without network accesses. As a result, it is non-intrusive to dynamically deprive the IME app from certain accesses as evidenced by prior research on dynamic permissions on Android [32]. After a rollback, as all residual states inside an IME app have been cleaned, any pending actions like insertion or deletion will not cleared as if they never happen. Thus, there won't be any confusions to the content provider and services.

5 EVALUATION

We have implemented I-BOX based on Android 4.2.2 and Linux kernel OMAP 3.0.72. It consists of two main parts: i) a user-level modification of the Android application framework to insert the I-BOX policy engine and network control module; ii) a kernel module to handle checkpoints and rollback of IME apps.

Experimental Setup. All of our experiments were performed on a Samsung Galaxy Nexus smartphone with a 1.2 GHz TI OMAP4460 CPU, a 1GB memory and 16GB internal storage. We evaluate I-BOX using 11 popular IME apps to measure the performance overhead of I-BOX. The 11 IME apps (as shown in the first column of Table 1) are ranked among the highest in popularity in a large third-party market[3]. Many of these IME apps have been installed more than millions of times (Figure 1). In our testing, we set the security policies to include all contacts in the phone and all commonly used accounts and passwords. This forms a trie containing around 400 words.

[3]http://www.wandoujia.com/

5.1 Performance Evaluation

The time overhead of I-BOX comes from three parts: (1) time to find the quiescent points; (2) time to perform memory checkpoint and rollback, and (3) time to perform file save and restore. To measure the performance overhead, we asked a volunteer with an average typing speed of about 100 characters per minute to enter a 10 word paragraph in an SMS app using the tested IME apps. We did not use an automation tool like an Android Monkey as it cannot handle the complex UI interface of these IME apps.

Latency. As shown in Table 1, the time to find a quiescent point is very small (less than 14ms). This confirms our observation that it is very easy and fast to find or force a quiescent point to do checkpoint and rollback on an IME app. The time of saving and restoring an IME app's memory state is also very small (less than 29ms) since I-BOX does not really copy the whole memory but just mark them as COW. Based on the files touched by the IME app process during the typing, I-BOX needs to restore a few files to prevent the IME app from concealing the secret inside files. Hence, the time for file save and restore is a little bit lengthy (60 ms), which can further be improved by using a copy-on-write file system. In total, the maximum total time to do a checkpoint (including finding a quiescent point) is less than 103ms (14ms + 29ms + 60ms). In contrast, the world record of texting is typing a complicated 25 word message (159 characters) in 25.94 seconds [5], which corresponding to 163 ms/character and 1.0376 second/word. Hence, the time to do checkpointing is very small compared to user typing. As the time to search the trie is negligible, we didn't report it here.

Power. To measure the power overhead incurred by I-BOX, we used the TouchPal IME to input an article and its non-Latin translation to a text-note app called Catch and count its power status. The total input process spans 30 minutes for both unmodified Android and I-BOX-capable Android. We found that in both cases the power dropped from 100% to 99%, whose differences were indistinguishable. This is probably because the IME app is not power-hungry and the additional power consumed by I-BOX was evened by the reduced network transmissions, which is thus hard to be distinguished without a highly accurate power meter. In our future, we plan to further characterize the power consumption using an accurate power meter.

5.2 Security Evaluation

Here, we evaluate whether I-BOX indeed has mitigated the leakage of a user's sensitive keystrokes. We still use the IME apps in our performance testing, along with a

IME app	Quiescent C (ms)	Quiescent R (ms)	Memory C (ms)	Memory R (μs)	File C (ms)	File R (ms)
Sogou	13.3	14.1	22.8	91	30	30
Baidu	8.2	11.1	22.6	275	40	40
QQ	12	11.8	24.3	31	60	30
Pinyin	11.8	12	20.8	122	10	10
Vee	5.9	10.3	0.022	61	20	20
Guobi	7.4	9.5	25.5	61	10	10
Octopus	11.4	11	28.9	245	30	20
iFlytek	4.6	9.7	13.9	92	10	10
Slideit	13.2	15.2	13.5	152	20	30
Jinshou	3.1	6.5	28	91	60	50
TouchPal	7.8	13.3	22.1	183	30	30
Baidu*	3.3	10.9	9	61	30	40
Average	8.4	11.3	22.5	140	28.3	26.7

Table 1: Time overhead for finding a quiescent point, doing checkpoint (C) and rollback (R).

repackaged malicious IME app (described in §2.3), to evaluate its effectiveness. According to the accessibility of these IME apps, we conducted three sets of experiments to determine effectiveness: black-box testing, gray-box testing, and white-box testing.

5.2.1 *Black-box Testing*

Since most of the IME apps use proprietary unknown protocols with unknown encryptions, we cannot directly trace the network packets to confirm our effectiveness. Therefore, we take a black-box approach to approximating our result. That is, instead of inspecting the packet contents, we inspect the packet differences sent by the IME-apps with I-BOX and without I-BOX, within an identical experiment setup and time window.

In particular, we ran all these apps using a two-minute time window, and we typed around 30 non-Latin words with "aa@usenix.org" as the sensitive word and then observed the packet differences using the Wireshark tool. Usually, these IME apps will send some packages out when a user types something that triggers the cloud input function. Interestingly, we found 6 out of the 11 tested apps have a different number of packages, as shown in Table 2. With I-BOX being enabled, there are less packages to be sent out compared to normal ones. This is because I-BOX controls the network of the target IME app when it detects sensitive input data and prevents the target IME app from leaking the data out.

While such side-channel based black-box testing cannot fully confirm that we have prevented all leaks, we believe it is highly likely that I-BOX has stopped them, even for the other 5 apps that we did not observe package differences for. (It is highly likely that these IME apps have buffered the input with the intent to send the data out later. However, our oblivious sandboxing mechanism will clear the buffered sensitive data).

IME app	w/o I-BOX	w/ I-BOX
Baidu	17	6
Sogou	44	30
QQ	37	20
Octopus	32	16
TouchPal	70	28
Baidu*	30	18

Table 2: #packages observed for the testing apps.

```
0000  45 00 01 6e 89 53 40 00  40 06 0c 4b 0a 00 00 02   E..n.S@. @..K....
0010  2a 79 6f 71 ed f1 22 ba  14 22 a5 f1 4d 0a f3 65   *yoq.."." ."..M..e
0020  50 18 00 d5 38 4e 00 00  47 45 54 20 2f 73 65 61   P...8N.. GET /sea
0030  72 63 68 3f 6b 65 79 63  6f 64 65 3d 35 30 25 32   rch?keyc ode=50%2
0040  43 35 36 25 32 43 35 35  25 32 43 35 32 25 32 43   C56%2C55 %2C52%2C
0050  35 32 25 32 43 35 32 25  32 43 35 31 25 32 43 35   52%2C52% 2C51%2C5
0060  34 25 32 43 35 32 25 32  43 35 30 25 32 43 35 34   4%2C52%2 C50%2C54
0070  25 32 43 35 33 26 75 73  65 72 69 64 3d 66 66 61   %2C53&us erid=ffa
0080  35 66 62 64 35 2d 32 35  38 33 2d 34 36 35 32 2d   5fbd5-25 83-4652-
0090  62 37 33 63 2d 65 62 35  34 34 32 65 34 61 32 66   b73c-eb5 442e4a2f
00a0  36 26 68 69 73 74 6f 72  79 3d 73 73 6e 25 32 30   6&histor y=ssn%20
00b0  69 73 25 32 30 36 39 37  36 34 32 35 37 36 20 48   is%20697 642576 H
00c0  54 54 50 2f 31 2e 31 0d  0a 43 6f 6f 6b 69 65 3a   TTP/1.1. .Cookie:
00d0  20 61 75 74 68 5f 74 6f  6b 65 6e 3d 66 66 61 35    auth_to ken=ffa5
00e0  66 62 64 35 2d 32 35 38  33 2d 34 36 35 32 2d 62   fbd5-258 3-4652-b
00f0  37 33 63 2d 65 62 35 34  34 32 65 34 61 32 66 36   73c-eb54 42e4a2f6
0100  0d 0a 48 6f 73 74 3a 20  7a 68 2d 63 6e 2e 69 6d   ..
```

Figure 5: Hexdump of the traced Touchpal package. The leaked SSN is highlighted.

5.2.2 *Gray-box Testing*

Among these 11 IME apps, we are able to observe the packet payload of TouchPal (as in discussed in §2.3) because it uses a plain-text protocol. Therefore, we conducted gray-box testing to confirm I-BOX indeed mitigated the privacy leakage. In this experiment, we open a client "SMS" app to send a short message to one friend with a social security number (SSN), which is private and sensitive by default. The text to send is a mixture of both Latin and non-Latin languages, as well as the number. Cloud input functionality will be triggered in this case.

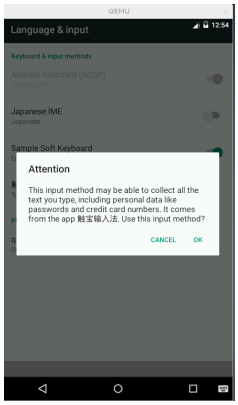
Interestingly, without I-BOX's protection, we found that Touchpal uploaded not only the keycodes the user typed as arguments of cloud input, but also the text message before the current input cursor that includes the sensitive social security number to the cloud through an HTTP POST method. We intercepted this packet using a man-in-the-middle attack. Part of the packet is displayed in Figure 5. However, with I-BOX's protection, we found that I-BOX successfully detected the critical number and shutdown its network to stop the leakage of data, and we did not observe any network trace.

We also studied the privacy warnings generated by Android on which data an IME may collect. Figure 6 shows that Android generates privacy warnings for two popular IME apps, Sogou and TouchPal, indicating that they may collect users' passwords, credit card number, etc. This further confirms our conclusion that they collect users' privacy data.

Apps	Without I-BOX	With I-BOX
SMS (phone number)	6204562244	62045
SMS (message)	Let's meet tomorrow noon at room 302	Let's meet tomorrow noon at room 302
Instagram (account)	thisisfortest@gmail.com	thisisf
Instagram (password)	fakepassword	
Facebook (account)	thisisfortest@gmail.com	thisisf
Facebook (password)	dontbelieveit	
Alipay	nomoney@yahoo.com	nomo
Gmail	tosomeone@hotmail.com	tosom
Google Play	Ingress	Ingress
browser	How much is this PS3?	How much is this PS3?

Table 3: Evaluation result w/ repackaged Baidu IME using different client apps.

(a) Sogou IME App (in Chinese)



(b) TouchPal IME App (in English)

Figure 6: Privacy Warning by Android for two popular IME apps. The left is shown in Chinese and the right is shown in English; the essential meanings are the same.

5.2.3 *White-box Testing*

As discussed in §2.3, we repackaged a very popular Baidu IME app to log all of the user input data and send them out to a malicious server we controlled. Hence, this repackaged IME app is essentially a keylogger. We were able to perform white-box testing by inspecting the packet payloads and confirming them with the source code of our malicious payload. We installed this IME app on our test phone and then used this phone to enter some user-defined private sensitive data with different client apps ranging from SMS, Facebook, and Gmail, etc. Table 3 shows the data we collected at the server side with and without I-BOX's protection.

From this table we can clearly observe that without I-BOX, the malicious IME app will steal all the data that a user enters. Consequently, all sensitive data has been leaked out; with I-BOX, it automatically blocks the network connection so that the server cannot receive any complete sensitive information. For instance, for passwords, the malicious server cannot receive anything as shown in the Instagram and Facebook case. As I-BOX shuts down the malicious IME app's network when it finds character sequences that have matched part of the sensitive phrase in our security policy, the server side can only receive the parts of the typed characters. For example, when a user tries to type her Facebook account *thisisfortest@gmail.com*, the server side can only receive a part of it, i.e. *thisisf*[4]. While partial sensitive input is still being leaked, we believe it is still hard for attackers to guess the original message.

5.3 Users Experience

One principal goal of I-BOX is to limit the negative influence on an end user's experience as little as possible. To evaluate this, we tested latency by determining how an end user would feel when typing characters on devices protected by I-BOX. For this, we invited a dozen students (6 undergraduate and 6 master students) in our Lab to install I-BOX on their phones, and asked them to use our system and provide us with feedback. By default, I-BOX uses the context-based policy and derives all sensitive data from the contacts and cookies. Two of them also tried to input their girl-friend's names and birth dates into I-BOX.

To our pleasure, none of the users complained of any latency imposed by our system. As shown in Table 4, there is only 0.4 milliseconds (ms) overhead per character imposed by our policy checking. While network shutdown takes about 180 ms, it is not executed per word and is instead triggered only when certain sensitive words are going to be formed. Therefore, the additional overhead added by I-BOX cannot be detected by end users. This is because the typing speed for a normal user is 625ms per character, and the world fast record is 160 ms per character, as shown in Table 4.

One complaint we received so far is that the users now need to manually type their account instead of using the automation features provided by the IME apps. We believe this is worthwhile for better privacy protection. Another complaint is that they need to specify their additional secrets manually; this will motivate us to design better UI interface in our future work.

[4]Note that we regard the sequence after @ as one character because an attacker can guess the rest by the first character most of the time.

Policy Checking	0.4ms/char
Network Shutdown	180ms
Checkpoint/Restore	103ms
Guinness World Records of fastest texter	160ms/char
normal user speed	625ms/char

Table 4: Statistics regarding the usage latency of I-BOX.

6 DISCUSSIONS AND LIMITATIONS

While I-BOX has made a first step to mitigate keystroke leakage against untrusted IME apps, there are still a number of limitations in its design and implementation.

Side-channel attacks It has been viable to use side channels to infer some keystroke information [9, 4]. I-BOX currently cannot prevent such side channel attacks. However, such threats are usually less severe than those of malicious IME apps, which can accurately observe all user input. We leave it as our future work to address issues related to the side-channel leakages.

Colluding malware As I-BOX currently only runs an IME app inside in a sandbox transactionally, it is still possible that an IME app could collude with another malware to leak information (i.e., the colluding attack [8]). For example, an IME app could first save the user input in a local file, and inform a colluding malware to read the file when the transaction has not been rolled back and then divulge the input. This essentially violates the policies of I-BOX. However, it is challenging for sandboxing to reliably prevent this, as studied by TxBox [25].

Security of I-BOX Any new security tools may bring new security implications as they usually touch security-sensitive data and I-BOX is of no exception. As I-BOX can essentially touch all users' sensitive data, it is essentially a key logger as well. Yet, I-BOX is much simpler than close-sourced proprietary IME apps (1,700 LOCs vs. hundreds of thousands LOCs). Regarding whether to trust I-BOX or other IME apps, third-party agents need to only audit the code of I-BOX instead of using gray-box based approaches to auditing the behavior of dozens of third-party IME apps. Meanwhile, I-BOX is completely a local service and will not send any private data out of the phone.

Permission Attacks As I-BOX's security is based on Android permission systems, it cannot defend against attacks against the permissions like component hijacking attacks and confused deputy attacks [23]. We consider this out of the scope of this paper; actually there have been a number of prior systems that statically and dynamically detect and prevent such attacks (e.g., [12, 43]). Actually, Android has significantly improved its permission systems since version 4.2 [3].

Voice input Currently we limit input data protection to handwriting input and keystroke input and do not consider voice input as it does not have keystrokes. Yet, users usually use dedicated system services like Apple Siri, Google Now and Microsoft voice recognition. How to handle voice input and preserve its privacy is very challenging and will be our future work.

Beyond Mobile IME Apps Note that the approach of I-BOX does not necessarily only apply to mobile platforms; Similar techniques can also be applied to desktops, which suffer from a similar dilemma between privacy and usability. We may provide a similar oblivious sandbox for each IME app, which should be straightforward as Android actually runs atop Linux. We leave this as our future work. Besides, other applications that requires a tradeoff between privacy and usability may use execution transaction like I-BOX.

7 RELATED WORK

Privacy leakage detection in mobile devices. Recently, there have been significant efforts on the detection of privacy leakage in mobile devices. Early attempts include TaintDroid [16, 17] and PiOS [15], and recent efforts include such as Woodpecker [22], AndroidLeaks [20], ContentScope [50], and Appprofiler [35]. In particular, TaintDroid [16] uses dynamic taint analysis to track whether sensitive information (e.g., address book) can be leaked through the network. PiOS [15] uses static analysis and focuses on the privacy leakage in iOS apps. Woodpecker [22] leverages an inter-procedural data-flow analysis to inspect whether an untrusted app can obtain unauthorized access to sensitive data. ContentScope [50] detects passive content leak vulnerabilities, by which in-app sensitive data can be leaked.

AndroidLeaks [20] instead uses static analysis to detect data leakage in Android apps. Chan et al. [10] further leverages mobile forensics to correlate user actions with privacy leakages. Appprofiler [35] creates a mapping between high-level API calls and low-level privacy-related behavior, which is then used to provide a high-level profile of App's privacy behavior. Besides, there have also been interests in detecting privacy leakage due to mobile ads [38]. In contrast, I-BOX focuses on preventing leakage of sensitive keystrokes.

Privacy leakage prevention in mobile devices. Other than detecting privacy leakage, there are also a number of systems that prevent private data from being leaked. By extending TaintDroid [16], AppFence [24] prevents applications from accessing sensitive information using data shadowing, and it also blocks outgoing communications tainted by sensitive data. While I-BOX and AppFence both block network communications when sensitive data is to be leaked, there are substantial differences: AppFence uses shadowing to provide an illusion to the app such that it can continue performing its taint tracking, whereas I-BOX does not use any illusion nor any instruction-level taint tracking, due to the per-

vasive existence of native code. Meanwhile, AppFence does not encounter the challenges we faced such as consistent rollback, and it only simply blocks the network communication, whereas I-BOX still has to keep the connection and allow other data to be transferred.

TISSA [51] tames information stealing apps to stop possible privacy leakage. SpanDex [14] further uses symbolic execution to quantify and limit the implicit flows through a sandbox, to prevent an untrusted application from leaking passwords. Through automatic repackaging of Android apps, Aurasium [43] attaches sandboxing and policy enforcement atop existing apps, to stop malicious behaviors such as attempts to retrieve users' sensitive information. Unlike Aurasium that adds a sandbox to an app, πBox [30] shifts the sandboxing protection of private data from the app level to the system level, and offers a platform for privacy-preserving apps. However πBox trusts a few app vendors to protect users' privacy data, while I-BOX treats the vendor of IME apps as untrusted, due to their incentives to collect users' input. TinMan [42] instead completely offload passwords-like secret to a remote cloud, but only handles a class of special secrets that are not necessary to be displayed in mobile devices. ScreenPass [31] leverages a trusted software keyboard to input and tag passwords and uses taint tracking to ensure that a password is only used within a specific domain. In contrast, while I-BOX also uses a trusted software keyboard for password input, it focuses more on preventing a malicious IME from leaking sensitive data (not only passwords).

Checkpoint and restore. I-BOX employs a checkpoint and restore mechanism to prevent privacy leakage. Such a mechanism has been built for transactional memory [6], execution transactions [37], as well as whole-system transactions [33]. Retro [26] leverages selective re-execution for intrusion recovery. Storage Capsules [7] also use checkpoint and restore to wipe off residual data after an application has viewed data in a desktop. I-BOX is an instance of a system transaction but designed specially for untrusted IME apps.

Sandboxing. There have been a large number of efforts in building sandboxes to execute untrusted programs, web applications, and native code. These tools were built using a variety of approaches such as kernel-based systems [19], user-level approaches [27], system call interpositions [21], or binary code translation [18], and recompilation [45].

A sandbox that also contains transactions is the TxBox [25], a tool built atop TxOS [33] for speculative execution and automatic recovery. While I-BOX and TxBox share the similarity of using transactions to build a sandbox, there are still significant differences: the goal of TxBox is to confine the execution of native x86 programs atop Linux kernel, whereas I-BOX is to confine the IME apps atop Android OS. Consequently, I-BOX faces additional challenges including resolving IPC bindings. Further, using quiescent points in I-BOX significantly simplifies the design and implementation.

8 CONCLUSION

This paper made a first systematic study on the (in)security of third-party (trusted or untrusted) IME apps, and revealed that these apps tend to leak users' sensitive input (due to their incentives of improving user's experience). To enjoy the rich-experiences offered by such apps while mitigating information leakages, this paper described I-BOX as a first step towards this direction. In light of the opaque nature of an IME app, I-BOX leverages the idea of transactions to run an IME app to make it oblivious to users' sensitive input. Experiments showed that I-BOX is efficient, incurs little impact on users' experiences and successfully thwarted the leakage of sensitive user input.

ACKNOWLEDGMENTS

We thank our shepherd William Enck and the anonymous reviewers for their insightful comments, Xiaojuan Li and Yutao Liu for helping prepare the final version. This work is supported in part by the Program for New Century Excellent Talents in University, Ministry of Education of China (No. ZXZY037003), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), and the Singapore NRF (CREATE E2S2).

REFERENCES

[1] Free Chinese-made software poses security risk. http://www.japantimes.co.jp/news/2013/12/26/national/chinese-made-computer-input-system-banned-in-government-agencies/#.U21w5_aPUS0.

[2] smali-An assembler/disassembler for Android's dex format. https://code.google.com/p/smali/.

[3] Security enhancements in jelly bean. http://android-developers.blogspot.jp/2013/02/security-enhancements-in-jelly-bean.html, 2013.

[4] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith. Practicality of accelerometer side channels on smartphones. In *ACSAC*, 2012.

[5] BBC News. Salford woman makes bid for fastest text title. http://news.bbc.co.uk/local/manchester/hi/people_and_places/newsid_8939000/8939790.stm, 2010.

[6] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *CCS*, pages 223–234, 2008.

[7] K. Borders, E. Vander Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *Usenix Security*, 2009.

[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.

[9] L. Cai and H. Chen. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *HotSec*, 2011.

[10] J. J. K. Chan, K. W. Tan, L. Jiang, and R. K. Balan. The case for mobile forensics of private data leaks: Towards large-scale user-oriented privacy protection. In *APSYS*, 2013.

[11] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Oakland*, pages 191–206, 2010.

[12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, pages 239–252. ACM, 2011.

[13] China IT Research Center. Third-part IMEs usage stats in China for 2014 Q1. http://www.cnit-research.com/content/201405/303.html, 2014.

[14] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. Spandex: Secure password tracking for android. In *USENIX Security*, 2014.

[15] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[16] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[17] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS*, 32(2):5, 2014.

[18] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX ATC*, 2008.

[19] T. Fraser, L. Badger, and M. Feldman. Hardening cots software with generic software wrappers. In *Oakland*, pages 2–16, 1999.

[20] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Trust*, 2012.

[21] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *USENIX Security*, 1996.

[22] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[23] N. Hardy. The confused deputy:(or why capabilities might have been invented). *SIGOPS Oper. Sys. Review*, 22(4):36–38, 1988.

[24] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *CCS*, 2011.

[25] S. Jana, D. E. Porter, and V. Shmatikov. Txbox: Building secure, efficient sandboxes with system transactions. In *Oakland*, 2011.

[26] T. Kim, X. Wang, N. Zeldovich, M. Kaashoek, et al. Intrusion recovery using selective re-execution. In *OSDI*, 2010.

[27] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX ATC*, pages 139–144, 2013.

[28] O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *USENIX ATC*, pages 323–336, 2007.

[29] W. S. Labs. Fake input method editor(ime) trojan. http://community.websense.com/blogs/securitylabs/archive/2010/07/05/trojan-using-input-method-inject-technology.aspx.

[30] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. πbox: a platform for privacy-preserving apps. In *NSDI*, 2013.

[31] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. P. Cox. Screenpass: Secure password entry on touchscreen devices. In *MobiSys*, pages 291–304, 2013.

[32] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, pages 328–332, 2010.

[33] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009.

[34] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *ACM conference on Data and application security and privacy*, 2013.

[35] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *ACM conference on Data and application security and privacy*, pages 221–232. ACM, 2013.

[36] M. A. Salehi, T. Caldwell, A. Fernandez, E. Mickiewicz, E. W. Rozier, S. Zonouz, and D. Redberg. Reseed: Regular expression search over encrypted data in the cloud. In *CCGrid*, 2014.

[37] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using rescue points to navigate software recovery. In *Oakland*, 2007.

[38] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, 2012.

[39] K. Subramanyam, C. E. Frank, and D. F. Galli. Keyloggers: The overlooked threat to computer security. http://www.keylogger.org/articles/kishore-subramanyam/keyloggers-the-overlooked-threat-to-computer-security-7.html.

[40] TechSpot News. Google fired employees for breaching user privacy. `http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html`, 2010.

[41] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *HPCA*, 2013.

[42] Y. Xia, Y. Liu, C. Tan, M. Ma, H. Guan, B. Zang, and H. Chen. Tinman: eliminating confidential mobile data exposure with security oriented offloading. In *EuroSys*, 2015.

[43] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security*, 2012.

[44] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *ICSE*, 2015.

[45] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, Jan. 2010.

[46] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

[47] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.

[48] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.

[49] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Oakland*, 2012.

[50] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *NDSS*, 2013.

[51] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Conference on Trust and Trustworthy Computing*, 2011.