Laurent Simon*, Wenduan Xu, and Ross Anderson

Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards

Abstract: We present a new side-channel attack against soft keyboards that support gesture typing on Android smartphones. An application without any special permissions can observe the number and timing of the screen hardware interrupts and system-wide software interrupts generated during user input, and analyze this information to make inferences about the text being entered by the user. System-wide information is usually considered less sensitive than app-specific information, but we provide concrete evidence that this may be mistaken. Our attack applies to all Android versions, including Android M where the SELinux policy is tightened.

We present a novel application of a recurrent neural network as our classifier to infer text. We evaluate our attack against the "Google Keyboard" on Nexus 5 phones and use a real-world chat corpus in all our experiments. Our evaluation considers two scenarios. First, we demonstrate that we can correctly detect a set of pre-defined "sentences of interest" (with at least 6 words) with 70% recall and 60% precision. Second, we identify the authors of a set of anonymous messages posted on a messaging board. We find that even if the messages contain the same number of words, we correctly re-identify the author more than 97% of the time for a set of up to 35 sentences.

Our study demonstrates a new way in which system-wide resources can be a threat to user privacy. We investigate the effect of rate limiting as a countermeasure but find that determining a proper rate is error-prone and fails in subtle cases. We conclude that real-time interrupt information should be made inaccessible, perhaps via a tighter SELinux policy in the next Android version.

Keywords: mobile, smartphone, android, side channel, interrupt, typing, gesture, gesture typing, SwiftKey, Google keyboard, keyboard, procfs, virtual file system, virtual file, artifical neural network, neural network, reccurent neural network, RNN, machine learning, ML

DOI 10.1515/popets-2016-0020

Received 2015-11-30; revised 2016-03-01; accepted 2016-03-02.

1 Introduction

We users expect a certain level of isolation between mobile apps. For example, we do not expect sensitive information we enter in a messaging app to be read by an apparently benign and permissionless weather app. In this paper, we show that this basic assumption does not hold on soft keyboards that support "gesture typing". This new feature can thus compromise users' privacy.

Gesture typing was invented to improve usability on smartphones with small touch screens. In this mode, you swipe your finger from one character to another rather than tapping each key individually (Section 2). This feature is enabled by default on Samsung and Nexus devices.

Our attack exploits supposedly harmless information exposed by the operating system to every process on the device, namely the system-wide screen hardware interrupt counter and the system-wide software interrupt (a.k.a. context switch) counter. Intuitively, when a user interacts with the screen, such as by touching it or moving a finger in contact with it, the Android kernel receives a hardware interrupt from the interrupt controller, which it can act on to retrieve the current finger location. The number of hardware interrupts leaks information about what a user is typing (Section 2). Now a soft keyboard app must track a user's finger position on the screen in order to infer the word entered, so it must retrieve information from kernel-land into user-land, which in turn requires context switches. The number of context switches (a.k.a. software interrupts)

***Corresponding Author: Laurent Simon:** University of Cambridge, E-mail: lmrs2@cam.ac.uk
Wenduan Xu: University of Cambridge, E-mail: wx217@cam.ac.uk
Ross Anderson: University of Cambridge, E-mail: rja14@cam.ac.uk

leaks information about what the user is typing (Section 2), even when other processes are running – our test phones have 200 running processes on average and 60 apps installed (Section 3).

Our attack monitors the system-wide interrupt counters and uses supervised machine learning to infer text entered by users. We borrow techniques from the Natural Language Processing (NLP) community; we use a sequence model based on a Recurrent Neural Network (RNN) (Section 2.3). We train each user individually, and evaluate the attack in two different scenarios:

1. **Detection of pre-defined sentences:** Given a set of sentences of interest, we ask if we can detect when a user enters them. This could be used by curious advertising libraries embedded in benign apps to infer personal information entered e.g. in messaging apps. For example, an ad library could detect a search term such as "how to lose weight" into a search engine or messaging app. We are able to correctly detect sentences containing at least 5 words 60% of the time with 55% accuracy (Section 3).
2. **User identification:** Given a set of sentences and users, we ask if we can identify which users typed which sentences. This could be used to deanonymize users of "anonymous" messaging apps such as YikYak. Even if sentences have the same number of words, we correctly re-identify their author 97% of the time (Section 3).

To mitigate this attack, we investigate rate limiting, but find it is more prudent to prohibit access to interrupt timing data entirely (Section 4).

In summary, our contributions are as follows:

– We present, design, and evaluate a new side-channel attack against soft keyboards that support gesture typing. These keyboards have been downloaded hundreds of millions of times from Google Play, and they come pre-installed in Samsung and Nexus devices.
– We highlight the limitation of the current SELinux policy in all Android versions, including the latest stock Android M and customized versions used in the Samsung KNOX security container. This allows a permissionless benign app installed on an Android smartphone to breach a user's privacy.
– We propose practical enhancements to the OS platform. After highlighting the imitations of rate limiting, we suggest prohibiting access to interrupt timing data – as well as other global statistical resources – altogether in the next Android version.
– On the scientific front, this is the first work to apply a Recurrent Neural Network (RNN) to a side-channel problem. By using an RNN, we are able to model sentences of arbitrary length naturally and capture long-term dependencies between words within a sentence.

2 Background and Threat Model

2.1 Android Soft keyboards & Gesture Typing

The Android OS lets users install "keyboard apps"[1] to replace the default soft keyboard. The Android OS allows only one keyboard app to be enabled at any time, and this is configurable by a user (we refer to the currently-enabled keyboard app as simply "the keyboard app" in what follows). When an app requires user input (e.g. through displaying an *EditText* Java object), the Android OS launches the keyboard app, which runs in a different process under a different user ID. A user effectively enters text in the keyboard app, which in turn sends it back to the caller app via IPC (APIs are standard and defined by the Android framework). A keyboard app can be used to provide new features, such as encryption [1] or novel input methods – which are the focus of this paper.

Over the years, keyboards with a "gesture typing" mode have emerged to ease user input on devices with small touch screens. At the time of writing, two gesture-typing keyboard apps are common: Swiftkey (50M-100M downloads) is the default keyboard on Samsung devices, used by both "untrusted" apps and apps running within the Samsung enterprise KNOX container, while The Google Keyboard app (100-500M downloads) now comes pre-installed in newer Android devices.

Gesture typing is a mode whereby a user slides her finger from one letter to another without lifting it off the screen. Fig. 1 shows the "path" that a finger would typically follow to enter the word "hello" in the keyboard (red trace). First, she positions her finger on the letter "h", then drags it to the letter "e", 'l' and then 'o', at which point she lifts her finger off the screen. Each subsequence ($\overrightarrow{he}$, $\overrightarrow{el}$, $\overrightarrow{lo}$) can be represented as a vector. When her finger transitions from one subsection

1 Technically, these keyboard apps are built on top of the Android Input Method Editor (IME) API.

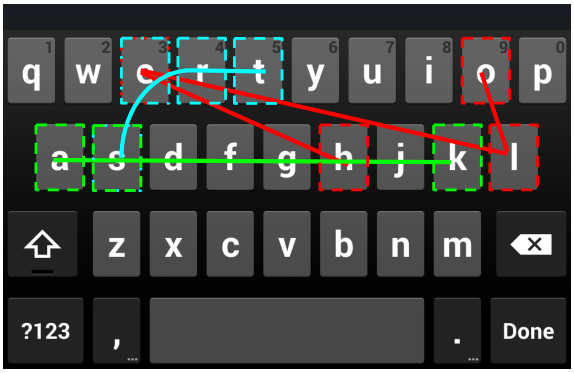

Fig. 1. Path of finger during input of word "hello" (red), "ask" (green) and "très" (light blue).

(e.g. $\overrightarrow{he}$) to the next (e.g. $\overrightarrow{el}$), it often changes direction. Fig. 1 also shows an exception to this – the word "ask" (green trace).

When a user lifts her finger off the screen, the keyboard app interprets it as the end of a word and the "space" character is automatically added to the text. This process is repeated for each word in a sentence. The keyword app keeps track of finger position and infers the most likely word the user wanted to enter. This is fairly accurate in practice. The keyboard app is however limited by the dictionary of words it knows; it never outputs misspelled words, unknown abbreviations or slang words. If a user really wants to enter words that are not recognized by the app (e.g. "lol"), she must add them to the "personal dictionary" section of the phone Settings.

2.2 Android & procfs

The Android OS is built on top of Linux. Its security model is based on the concept of *application sandboxes*. Prior to Android 4.3, application sandboxes were implemented on top of Linux discretionary access control (DAC). On installation, an Android app was given a unique user ID (UID) and ran with the privileges of that user every time it was started. The application-layer permission model relied on this application sandbox. A permission (e.g. "Camera") was generally mapped to a dedicated Linux group (e.g. the "permission" group). Permissions had to be declared by app developers in the AndroidManifest.xml file, and were used to restrict access to system resources at run time. To these mechanisms, Android 4.3 adds the use of Mandatory Access Control (MAC) through SELinux. In practice, the SELinux policy is not as tight as one might expect, as we shall see shortly.

From Linux, Android inherits the *proc filesystem (procfs)*, a virtual filesystem that provides aggregated information about the system as well as detailed information about processes. Android also adds new entries within the procfs. The procfs information can help app developers during troubleshooting, and also provide useful information for which there is no Android API. Process-specific information is generally accessible under */proc/[PID]/** and */proc/pid_stat/[UID]/**, where *PID* is the process ID and UID the unique user ID. The security implications of process-specific information have been demonstrated in various papers [2–4] (Section 6). For example, Zhou *et al.* [4] show how traffic volume information gleaned through the file */proc/uid_stat/[UID]/tcp_snd* and */proc/uid_stat/[UID]/tcp_snd* can be used to fingerprint Twitter app traffic and identify a Twitter user. Such attacks worked before Android M because the SELinux security policy was too loose, i.e. certain process-specific files remained readable by any app on a device.

In Android M, however, the SELinux security policy was tightened up to fix this, so that an app can no longer access another process's specific files in procfs. This appeared to stop one app attacking another by relying on process-specific information. But we decided to study the details more carefully. For example, what are the implications of exposing the file */proc/interrupts* which contains real-time interrupt counters received from peripheral? What are the security implications of exposing the file */proc/stat* that contains an aggregated software interrupt (a.k.a. context switch) counter? As we shall see, they open up side channels with real security and privacy implications.

2.3 Attack Overview

Our threat model is a non-malicious but curious app running on the victim's device. This app does not require special permissions besides internet access (to send gleaned data to remote attackers) which, from Android M onwards, is automatically granted and non-revocable. This app does not actively attempt to break out of the sandbox; instead it observes and monitors publicly available "events" from the system while a user enters text in a victim app. Specifically, these "events" are the variations of (*1*) the system-wide screen interrupt counter and (*2*) the system-wide context-switch counter,

```
# Samsung Galaxy S Plus with Swiftkey keyboard
$ cat /proc/interrupts
[...]
247:       7489      msmgpio  qt602240-ts

# Samsung Galaxy S3 with Google keyboard
$ cat /proc/interrupts
[...]
387:      31695          0          0  s5p_gpioint  melfas-ts

# Nexus 5 with Google keyboard
$ cat /proc/interrupts
[...]
362:       4016    msmgpio  s3350
```

Fig. 2. Interrupt of interest in the file /proc/interrupts. The counter is highlighted in red and underlined.

accessible through the files */proc/interrupts* and */proc/stat* respectively.

For each unique word entered by a user, we observe in the system a series of events that can be used as a "fingerprint" to recognize that word. The challenge is that these events contain noisy data. So we use supervised machine learning to create a word fingerprint. The fingerprint is constructed from training data and is used to infer sentences entered later in victim apps. In certain attack scenarios, training data are not even required (Section 3.4). In the general case, however, we need a fingerprint, and we use both the screen interrupt counter and the system-wide context switch (a.k.a. software interrupt) counter as described in the following subsections.

Screen Interrupt Counter. This is available through the world-readable file */proc/interrupts*. Fig. 2 shows the relevant line containing the screen interrupt counter for different phones and keyboards. Fig. 3 (top) shows variations of the screen interrupt counter while a user types the word "hello" on a Nexus 5. The first section (denoted as (1)) corresponds to a user positioning her finger on the letter "h" and dragging it to the location of the letter "e": the interrupt counter increases linearly with the number of CPU cycles. When the number of CPU cycles reaches 6.5, the interrupt counter stops increasing and remains constant for a short period of time. This corresponds to the user's finger transitioning from subsection $\overrightarrow{he}$ to $\overrightarrow{el}$. This transition generally involves the finger (*1*) slowing down, (*2*) reaching a zero speed, and finally (*3*) re-accelerating to reach the next letter. While the finger is idle (zero-speed), the screen need not report any changes to the OS, so the OS no longer receives screen interrupts. This explains the plateau between each subsection (Fig. 3, top).

Fig. 3 (middle) depicts zero-speed positions of the finger as inferred by an "ideal" processing routine.

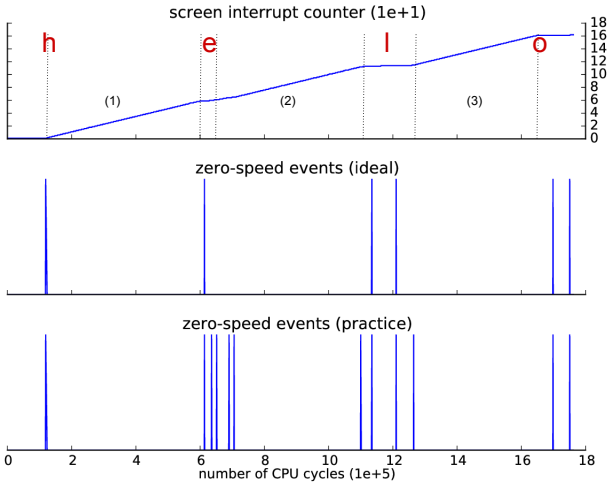

Fig. 3. Screen interrupt counter (top) for word "hello" on Nexus 5; the "ideal" zero-speed events of the user's finger we would ideally want to infer (middle); the zero-speed events detected in practice (bottom). We assume that the user drags her finger towards the letter "l" once only, hence the single "l". The keyboard app automatically infers the second "l".

These zero-speed events are indicative of the word entered so we use their positions as a feature for word fingerprinting. In practice, zero-speed events often correspond to a change of direction by the user's finger (e.g. to transition from subsequence $\overrightarrow{he}$ to $\overrightarrow{el}$). Sometimes, however, no change of direction is needed. This is illustrated in Fig. 1 (green trace) when entering the word "ask" – $\langle\overrightarrow{as},\overrightarrow{sk}\rangle = \|\overrightarrow{as}\|\|\overrightarrow{sk}\|$ (i.e. $\cos(\theta) = 1$). Certain users still voluntarily slow down their finger around the letter "s", which also creates an observable zero-speed event. Note that the absence of zero-speed events can also, by itself, be indicative of a specific word.

In practice, we may either miss zero-speed events or detect false positive ones. For example, the path of the finger may be a curve rather than a sequence of straight-line vectors. This is often the case if the angle θ between two consecutive subsequences is small, as illustrated in Fig.1 (light blue trace; $\overrightarrow{tre}$ to $\overrightarrow{es}$ correspond to French word "très" which means "very". For this reason, some changes of direction (and their corresponding zero-speed events) may not be reliably observable. Therefore, in practice, we observe a *probability distribution* of zero-speed events (Fig. 3, bottom), with different zero-speed events giving different amounts of information about words. Fig. 3 (bottom) illustrates the zero-speed events that our detection routine would typically detect in practice.

Global Context Switch Counter. From here on, we use the terms "context switch" and "software inter-

```
$ cat /proc/stat
[...]
ctxt 1781433
```

Fig. 4. Software interrupt in the file /proc/stat.

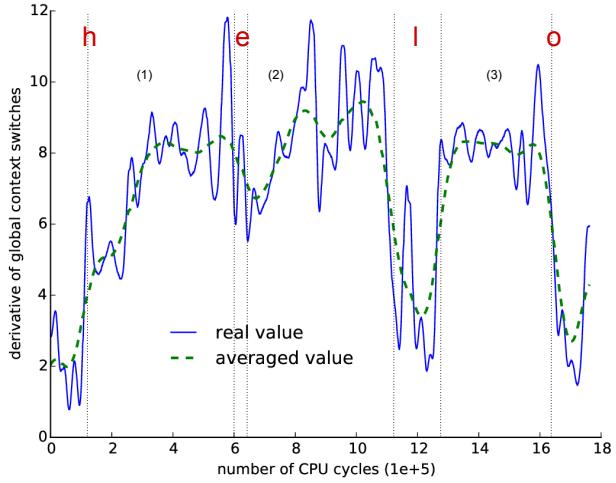

Fig. 5. Speed of the software interrupt counter during input of the word "hello" by a user. We assume that the user drags her finger towards the letter "l" once only, hence the single "l". The keyboard app automatically infers the second "l".

rupt" interchangeably. The software interrupt counter is accessible through the file */proc/stat*. The relevant line is shown in Fig. 4. Unlike the screen's interrupt counter, the line is the same on all devices as it is hardware-independent. We found that its first derivative (i.e. its speed) provides information about text entered in the keyboard. Before computing the derivative of the counter, we first pass it through a Savitzky-Golay smoothing filter [5]. The context switch counter speed corresponding to Fig. 3 is shown in Fig. 5 (word "hello"). During subsection (1) ($\overrightarrow{he}$), the user's finger starts idle at letter "h" (x-axis around 1). Its average speed then increases until it reaches a local maximum and remains roughly constant on the interval $[4, 6]$. Finally, its speed decreases to a local minimum when the finger reaches letter "e" at around 6.5. The same pattern repeats on intervals $[7, 12]$ and $[12, 17]$ corresponding to subsections $\overrightarrow{el}$ and $\overrightarrow{lo}$ respectively.

These patterns are rather intuitive. As it starts idle, the finger must first accelerate. Half-way through a subsection (say, $\overrightarrow{he}$), its speed starts to decrease until it reaches a local minimum. At this point, a new subsection starts and the pattern repeats itself. The variation of the global switch counter is caused by the keyboard app context-switching into kernel-land to retrieve the finger's current location. The speed of the

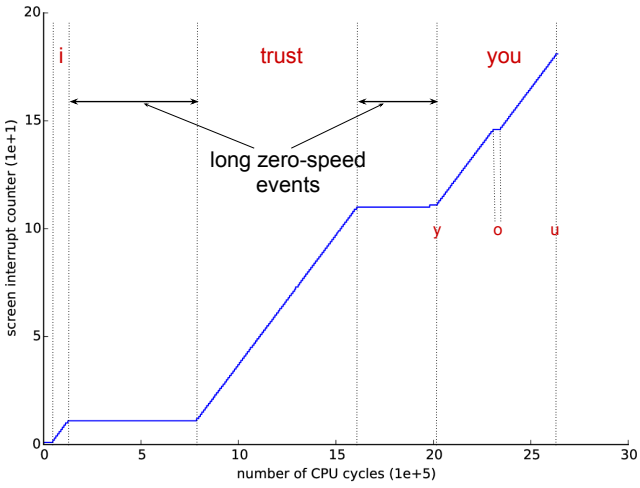

Fig. 6. Screen's hardware interrupt counter during input of the sentence "I trust you" by a user. Long zero-speed events are used to detect words.

finger is correlated with the speed at which the context switch counter varies, as illustrated in Fig. 5. Therefore this counter carries some information about the entered text, and we use it as an additional feature for word fingerprinting.

It is important to realize that monitoring (i.e. reading) the context switch counter may affect the measurements, since it involves invoking the syscall *read()* which requires a context switch. However, this turns out to have little effect on our attack for the following three reasons:

1. We read the file at almost constant intervals, so the number of context switches we generate is almost constant over time. Since we use the first derivative of the context switch counter, and the derivative of a constant function is zero, the measurement effect is to a first approximation removed;
2. Even if the interval between consecutive reads is not exactly constant, the smoothing filter we use further mitigates any artefacts generated by monitoring;
3. We use the same monitoring routine during the training and attack phase, so any residual artefacts we may add are taken into account when we create the fingerprints.

Sentence Decomposition into Words. A requirement of our attack is to be able to chop a sentence (i.e. its corresponding series of "events") into its corresponding words. To this end, we re-use the global screen interrupt counter. Recall from Section 2.1 that the keyboard app detects the end of a word when a user lifts her finger off the screen. While the finger is raised, no

activity on the screen is reported to the OS. Therefore the screen interrupt counter remains constant.

The zero-speed events induced by raising the finger last a lot longer than those caused by transitions between word subsequences. This is illustrated in Fig. 6 for the sentence "I trust you". We use this heuristic to detect the start and end of words. Through our evaluation (Section 3), we find this works more than 99.5% of the time in practice. This allows us to reliably chop a sentence signal into its constituent word signals. These are then passed through our fingerprinting routine as detailed next.

Supervised Training & Classification.

We first investigated a standard SVM classifier to classify each word signal on its own. An SVM ignores wider contexts in a sentence: it takes as input a single word signal and outputs a predicted word. The results were unsatisfactory: for half the users, word predictions were correct less than 10% of the time. So instead of inferring words in isolation, we now consider all the words in a sentence.

This lends itself to a Recurrent Neural Network (RNN), which can naturally model sequences of arbitrary length and consider long-range contextual information in a sentence beyond a local context-window. As described in the natural language processing literature [6, 7], it can propagate a potentially unbounded history of previous words and use this history for word prediction. An RNN is more general than a Markov chain, or a traditional n-gram language model [8], both of which are limited by simplistic independence assumptions – namely that the current word only depends on a limited number of predecessors. Most importantly, the history of words in a sentence is learned automatically during supervised training with an RNN, without the need to hard-code any indicator features, as we have to do for classifiers such as SVMs.

We use the RNN as a supervised classifier; that is, we train it using labelled examples to minimize classification errors on our training data. But unlike with an SVM, we train the RNN using lists of word signals representing sentences rather then individual word signals. At attack time, we use the trained RNN to classify decomposed word signals from intercepted keyboard swiping. The architecture of our RNN is shown in Fig.7a; it is an Elman recurrent neural network [9] that consists of an input layer x_t, a hidden layer h_t with a recurrent connection to the previous hidden layer h_{t-1} and an output layer y_t.

The input layer is a real-valued vector representing a context window of word signals, with the current word signal at position t in the middle. The hidden layer h_{t-1} keeps a representation of all a sentence's context history up to the current word signal. The current hidden layer h_t is computed using the current input x_t and hidden layer h_{t-1} from the previous position. The output layer represents probability scores of all possible words, with the size of the output layer being equal to the size of the vocabulary set.

To train the RNN, we feed to it all sentence signals in our training data one at a time, where each sentence is represented as a list of decomposed word signals. Moreover, each word signal has a corresponding ground-truth word label from our vocabulary set. The goal of training is to make the RNN as accurate as possible at predicting ground-truth words, according to some loss function that measures classification error on the training data.

Concretely, let $S_i = s_0, s_1, \ldots, s_n$ be a list of word signals for sentence i in the training data, and $W_i = w_1, w_2, \ldots, w_n$ be the ground-truth words for S_i. To train the RNN on (S_i, W_i), it reads all the signals in S_i in a left-to-right manner, and at each position t such that $0 \leq t \leq n$, the input x_t fed into the network is:

$$x_t = [s_{t-\lfloor k/2 \rfloor}; \ldots s_t; \ldots; s_{t+\lfloor k/2 \rfloor}], \tag{1}$$

where the right-hand side is the concatenation of all signals in a size k context window (we use $k = 5$ in all our evaluations). As the RNN moves across the input signals in S_i, it keeps a representation of all previously seen signals in its hidden layer up to the current step t, and it uses the values stored in h_{t-1} plus x_t to make a new prediction. Fig. 7b shows the RNN unfolded over an entire input sequence. Note that, the history stored in the hidden state of the RNN is potentially unbounded and the context windows enhance this history.

The RNN is trained with a cross-entropy objective, so does maximum-likelihood estimation over the training data. We use the backpropgation through time algorithm [10, 11] and stochastic gradient descent to minimize the cross-entropy error:

$$L(\Theta) = -\sum_i \log p_i, \tag{2}$$

where Θ is the parameterization of the network and consists of three matrices that are learned during supervised training[2]. Matrix $\mathbf{U}$ contains weights between the input

2 Note that the matrices are carried over across all predictions, and it is not the case that three new matrices are created for each new prediction. After each prediction, the values in these matrices are updated by backpropagation.

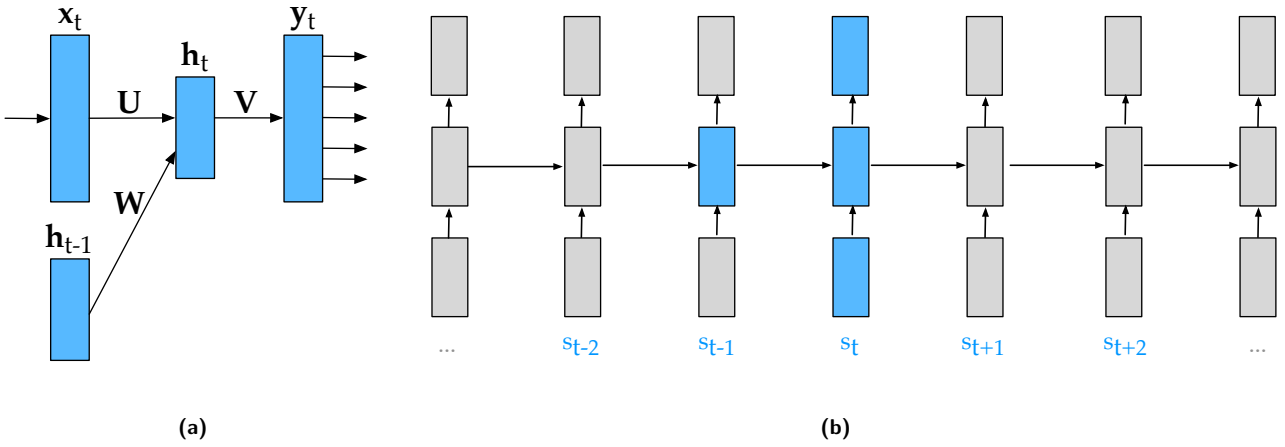

Fig. 7. The architecture of our recurrent neural network. (a) shows the state of the RNN at any given time step. (b) shows the RNN unfolded across the entire input sequence. A context-window size $k = 5$ is used, and the middle of the context-window is s_t; the bottom layer is the input layer, the middle and top layers are the hidden and output layers, respectively.

and hidden layers, $\mathbf{V}$ contains weights between the hidden and output layers, and $\mathbf{W}$ contains weights between the previous hidden layer and the current hidden layer. Minimizing the loss in Eq. 2 maximizes the probabilities of desired output in the training data and minimizes the probabilities of incorrect output.

To make a prediction, the following recurrence[3] is used to compute the hidden layer activations at input position t:

$$h_t = f(x_t\mathbf{U} + h_{t-1}\mathbf{W}), \tag{3}$$

where f is a non-linear activation function; here we use the sigmoid function $f(z) = \frac{1}{1+e^{-z}}$. The output activations are calculated as:

$$y_t = g(h_t\mathbf{V}), \tag{4}$$

where g is the softmax activation function $g(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$ that squeezes raw output activations into a probability distribution. The probability scores at the output layer represent the probability of a word given all previous words, $p(w_t \mid w_{t-1}, w_{t-2}, \ldots, w_0)$, and are used in the attack scenarios that we describe in later sections.

Training Phase. In the general case, our attack requires a preliminary "training phase" during which we build a set of word fingerprints using the RNN. Concretely, as word fingerprint features, we use the following information for each word:

- The length (in CPU cycles) that it spans. This is extracted from the system-wide screen interrupt counter.
- The location of its zero-speed events. This is also extracted from the system-wide screen interrupt counter.
- A stream of discretized values (sampled at regular time intervals) of the first derivative of the system-wide context switch counter.

In certain attack scenarios, the training phase is not even needed (Section 3.4). When it is needed, this phase requires users to enter lists of words in the keyboard app while another app collects the corresponding signals (i.e. the counters). These are used to train the RNN classifier and create the fingerprint set. Obviously, this phase requires knowledge of the words that are entered by users in order to map them to their corresponding signal. Currently, we build the fingerprint set (i.e. a training model) for each user in order to evaluate the efficiency of the attack (Section 3). In practice, the training phase could be performed by apps that receive enough genuine user input. For example, a school might require pupils to use approved apps extensively for everything from online discussion forums to homework submission, and then use the fingerprints generated from them to identify kids who send inappropriate messages via other social apps. And perhaps eventually it could be possible to eliminate the need for per-user training by building the fingerprint set with enough users – such that the resulting fingerprint set works for most users. In this study, however, we focus on evaluating the feasibility of the attack rather than scaling it.

Attack Phase. In the attack phase, a malicious permissionless app records the counters from *procfs* while a user enters text in another victim app on the

3 We assume the input to any layer is a row vector unless otherwise stated.

phone. Note that the attack could also be performed by a "normal" app against a "secure" app running in a KNOX container. Unlike in the training phase, the malicious app can only observe the signals (i.e. the counters) but not the words themselves. Using the fingerprint set at its disposal, it matches the observed signals against this set using the RNN – through the scores ouput by the output layer. More specifically, for a given sentence signal, the RNN outputs, for each word signal in a sentence signal, a probability that the word signal corresponds to a particular dictionary word.

The malicious app must first determine when to start collecting the signals. When it detects that the user interacts with a "screen view" of interest (e.g. the conversation screen of WhatsApp where users enter chat messages), it starts signal collection. The means of detecting the current "screen view" (a.k.a. "Activity" in Android parlance) are not a contribution of our study. They have been explored in previous work [3, 12–14].

User input is indicated by a signal similar to Fig. 6, i.e. with consecutive screen activity periods interleaved with short/long zero-speed events. The initial finger tap to pop the keyboard and the last finger tap corresponding to the "Send" button provide the attacker with further cues to find the start and end of target input. These tap events show up as short-lived peaks in the hardware interrupt trace. None of the chat apps we looked at automatically open the virtual keyboard (Viber, WhatsApp, Line, YikYak etc.). Therefore the initial tap is visible for all these apps. We suspect these apps do not open the virtual keyboard automatically for usability reasons, as it covers a large part of the screen. When returning to an open conversation, users tend to scroll through the latest chat messages received, so opening the keyboard automatically would significantly affect usability.

3 Evaluation

3.1 Methodology

The Corpus. We use the NPS Internet Chatroom Conversations corpus (Release 1.0) [15] available through the NLTK framework [16]. It consists of around 10000 English sentences gathered from age-specific chat rooms of various online chat services in October and November 2006. Within the corpus, we restrict ourselves to the most common 200 words, to which we refer as the "dictionary" from here on. The choice of 200 words strikes a balance between testing our techniques and the burden we put on study participants; by entering dictionary words 15min every day, it took no less than three weeks to collect these samples for each participant.

Participants. We recruited participants through word-of-mouth among acquaintances. We went for this option so we could meet them regularly if need be. When improvements to the software were suggested by participants, we could patch our software and deploy it rapidly. We had 8 participants in our study, three females and five males. Two of them used the gesture feature on their own phone. Their age was between 25 and 40 years old. Three of them have a Computer Science degree, while the others have backgrounds in Psychology, Criminology, Biology, Electronics, and Telecommunications. All have at least a Bachelor's degree, and five have a PhD. Five are native English speakers. These demographics are not representative of the general population. However, our study is radically different from behavioural studies where demographics play a significant role. In this study, we are only interested in simple characteristics, such as finger speed, when people enter words in gesture-based keyboards. This is a pilot study and in any case we believe the results will generalize (as we will discuss later).

Before the experiment, we asked participants if they used gesture typing on their phone. Those who did were told they could start immediately. The rest were asked to familiarize themselves with it and only start once they felt at ease with it. That typically took a few hours or days. We did this because we wanted to assess our attack on people who actually know how to use the feature. If we tested beginners who drag their finger slowly from one letter to another, it could create a bias in our favour, and artificially improve the efficiency of our attack: recall from Section 2.3 that the slower a user's finger, the easier it is to detect "zero-speed" events reliably to build a word fingerprint. After this preliminary requirement, we assumed that participants' typing characteristics did not vary significantly over the experiment period. So we did not retrain users over the course of the experiment. If changes in typing characteristics were a concern, one could update the model with the most recent data. The task that participants completed is described next.

Data Collection. We built a proof-of-concept (PoC) victim app and malicious app to run side-by-side on the Android platform. These apps run in different processes under different UIDs. Therefore they belong to different sandboxes, as would be the case in practice

(Section 2.2). We gave a Nexus 5 (OS $\geq$ 4.4) to the participants we recruited for the study.

Each participant enters lists of dictionary words in the victim app while the malicious app runs in the background and collects signals (i.e. the counters) from the files */proc/interrupts* and */proc/stat*. In the list of words entered by participants, each dictionary word appears 20 times, resulting in 4000 ($20 * 200$) word samples. On average, these samples represent 40MB per user when zip-compressed. We discuss how a curious app could upload this data stealthily to a remote server in Section 5. It takes three weeks for each participant to complete the data-collection task. The phones given to participants have about 60 apps installed on them, and an average of 200 processes running (as reported by the *ps* command). WiFi is enabled at all times. During the course of the experiment, participants witnessed the Android OS downloading updates; news apps regularly pushing articles; and games showing notifications. Our malicious app only monitors the PoC victim app, for ethical reasons.

Fingerprint Creation & Testing. These steps are run on a desktop in our evaluation (we discuss the feasibility of doing them on phones in Section 5). Signals collected from participants correspond to lists of words, so first, we chop each signal into its constituent word signals, using the heuristics presented in Section 2.3. This works over 99.5% of the time in practice. Once we have individual word signals, we randomly pick 85% of them as the training set, i.e. to create the corresponding word fingerprint. The other 15% are used as the testing set, that is, as unknown input we attempt to predict. For both the training and testing sets, we combine word signals to construct sentences in the corpus. These sentence signals are then used as input to the RNN (either for training or prediction). A trained model needs between 1.1 and 1.3MB worth of data when compressed, and up to 3MB without compression.

3.2 Word Prediction

We first want to understand how well words are inferred within a sentence. In this scenario, the RNN takes as input unknown sentence signals from users (i.e. from the testing set), and ranks each dictionary word in order of likelihood for each constituent word signals. The word that appears in the first position is the most likely word entered by the user given the signal, while the one that appears last is the least. For evaluation, we use all the word signals in our testing set. Fig. 8 shows the position

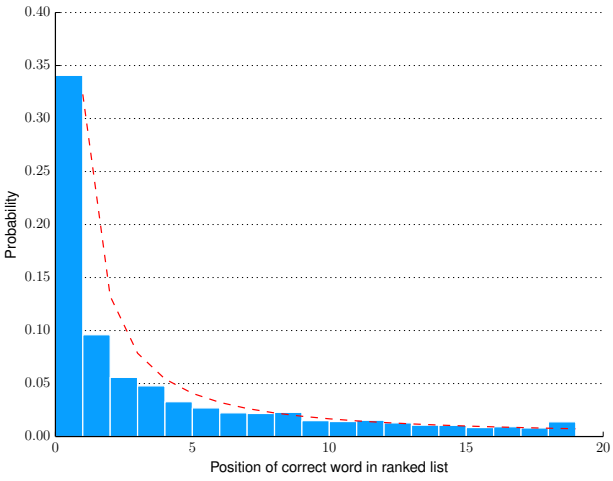

Fig. 8. Distribution of the position of correct word guess.

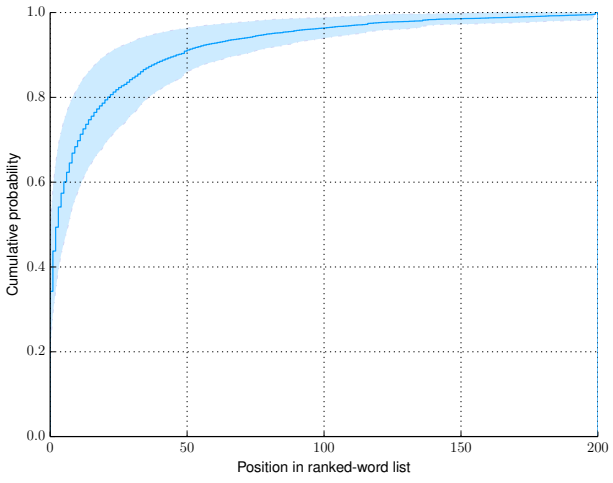

Fig. 9. Cumulative distribution of the position of correct word guess.

of the correct word in the ranked list. About 34% of the time, the correct word appears in first position (first bin of the histogram); this is ≈ 68 times better than a random guess ($p_{random\_guess} = \frac{1}{200} = 0.5\%$). The correct word appears in second position 9% of the time (second bin of histogram), etc. Fig. 9 shows the cumulative distribution of the position of the correct word in the ranked list. About 80% of the time, the correct word appears in the first 22 positions. Of course, there were some variations among participants, but the results did not indicate a correlation between users who had previously used swipe keyboards, and those who had not.

3.3 Detection of Sentences of Interest

Given a set of pre-defined sentences of interest (e.g. "I'm pregnant" or "I want to lose weight"), we ask if we can efficiently detect if a user enters them. A practical attack could be a school looking for pupils who sent messages bullying other students; parents trying to monitor the topic of discussion of their kids on social media; or just a curious app peeking at text entered by a user in the Google search bar. For the evaluation, we randomly select a set of sentences from our chat corpus. A sentence of interest ("SoI" from herein) need not span an entire sentence though; it may only be a subset of a longer sentence. For example, for the SoI "I'm pregnant", we want to detect it within longer sentences such as "I think I'm pregnant too". Our detection routine outputs a match if, for each word in an SoI, the word appears in the first N positions in the ranked list output by the RNN. For the evaluation, we vary the parameter N.

Fig. 10 shows the True-Positive-Rate (TPR) vs. False-Positive-Rate (FPR), a.k.a. the ROC curve for SoIs. For SoIs containing at least 4 words, we correctly detect them 50% of the time with a False Positive Rate (FPR) close to 0. The results are similar for SoIs containing at least 5 and 6 words. An important metric missing from the ROC plot is the precision of our matching algorithm, that is, *when we output a match, how often are we correct?* Fig. 11 answers this question. For SoIs containing at least 4 words, we correctly detect them 50% of the time ($TPR = 0.5$); and when we output a match we are correct 35% of the time ($PPV = 0.35$). This corresponds to a False Positive Rate (FPR) near 0 on the ROC curve of Fig. 10. For SoIs containing at least 5 words, the results improve: we correctly detect them 60% of the time ($TPR = 0.6$); when we output a match, we are correct 55% ($PPV = 0.55$). This corresponds to a False Positive Rate below 0.5% on the ROC curve. Intuitively, as the number of words in a sentence increases, we have more information to distinguish sentences. Therefore, for sentences with at least 6 words, results further improve to a $TPR = 0.7$ and $PPV = 0.6$ corresponding to a $FPR \leq 0.05$ on the ROC curve.

3.4 De-Anonymization of Users

Given a list of known sentences entered by a set of users, we ask if we can efficiently map each sentence to the user that entered it. As per our threat model outlined in Section 2.3, we assume that each user has our curious app running on their device. A practical attack scenario could be to identify users of "anonymous" messaging board apps such as YikYak messenger[4], which has more than 1M downloads on Google Play. Such apps let users write "anonymous" posts on a messaging board. A school could try to find which pupil posted an inappropriate message. YikYak messaging boards are arranged by location: posts are visible to all users in the vicinity of the sender. Posts are anonymous in the sense that they do not contain a name, pseudonym or location data that would link them to their author (Fig. 12). Posts do contain a time, but this is not very precise. During

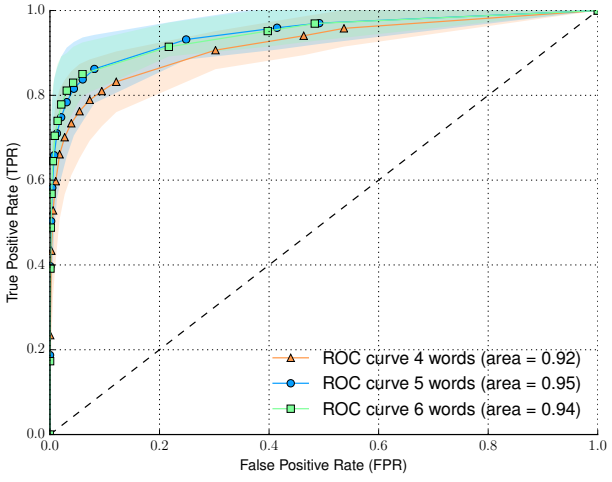

Fig. 10. TPR-FPR curve (ROC) of known sentence detection. The shadow area represents the standard deviation for sentences containing at least 4 words.

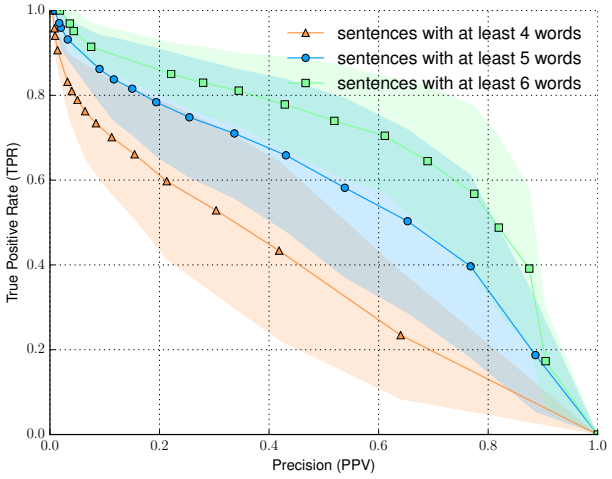

Fig. 11. Precision-Recall curve of known sentence detection.

4 http://www.yikyakapp.com/

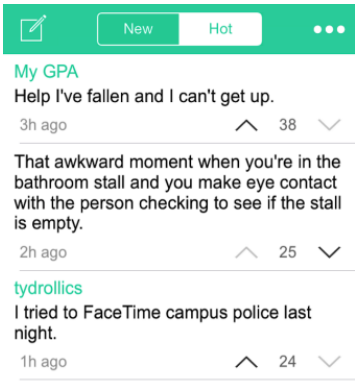

Fig. 12. YikYak messaging board example.

a time period T, U users may post anonymously to the board. Assuming these U users are infected by a curious app, we ask if it can identify which user entered which post. We first used the YikYak app ourselves to see how many messages were posted on average over time. We found that within one minute, no more than a dozen messages were posted in our area.

Let N be the number of sentences posted during a time period T on an anonymous messaging board. We denote a sentence as $Sen_{i,1\leq i\leq N}$. On each user's device, the curious app observes a signal $Sig_{i,1\leq i\leq N}$ as described in Section 2.3. Since there are N posts, there are exactly N signals that correspond to them. Note that certain users could be the author of multiple posts, that is, the number of users $U \leq N$.

3.4.1 Sentences of different lengths

If sentences on the messaging board each contain a different number of words, it becomes straightforward to map them to their corresponding signal Sig_i by simply counting the number of words in each Sig_i. As detailed in Section 2.3, we detect the number of words contained in a signal by counting "long" zero-speed events in the signal. Once we have the number of words entered by each user, we just map these to the length of sentences on the messaging board. No matter how many words each sentence contains, so long as each of them contains a different number of words, we can identify their author virtually all the time. The only condition is that we manage to properly count the number of words in a sentence, and our experiment reveals this works over 99.5% of the time in practice. Interestingly, in this attack, we neither need the user training phase nor the fingerprint. To make the task challenging, we study the case where all sentences have the same length.

$$\begin{pmatrix} sc_{1,1} & sc_{1,2} & \cdots & sc_{1,N-1} & sc_{1,N} \\ & & \cdots & & \\ & & \cdots & & \\ \cdots & \cdots & sc_{i,j} & \cdots & \cdots \\ & & \cdots & & \\ & & \cdots & & \\ sc_{N,1} & sc_{N,2} & \cdots & sc_{N,N-1} & sc_{N,N} \end{pmatrix}$$

Fig. 13. Score matrix.

3.4.2 Sentences with same length

This is the worst-case scenario for the attacker. Let L be the number of words in all sentences posted on the messaging board. The first step of our re-identification routine is to compute, for every signal Sig_i and every sentence Sen_i, a score that represents the likelihood that the signal Sig_i corresponds to Sen_i. Recall from Section 2.3 that our fingerprint routine outputs, for any of the L word signals $Sig_i[k], 1 \leq k \leq L$ in a sentence Sig_i and a dictionary word DW, the probability that $Sig_i[k]$ corresponds to DW. We define the score $score_{i,j}$ for sentence Sen_i and signal Sig_j as:

$$score_{i,j} \stackrel{\text{def}}{=} \sum_{k=1}^{L} log(proba(Sig_i[k] == Sen_j[k])), \quad (5)$$

where $Sen_i[k]$ is the k^{th} word of Sen_i, $Sig_j[k]$ is the k^{th} word-signal of Sig_j, and $proba(\mathsf{WS} == \mathsf{DW})$ is the probability that word-signal WS corresponds to dictionary word DW, as output by the RNN.

We then build a square "score matrix" where each row i represents a signal, each column j represents a sentence, and each element in the matrix is the score $score_{i,j}$ ($sc_{i,j}$) as illustrated in Fig. 13.

Sentence Prediction. We first evaluate how well we can infer the correct sentence given a sentence signal. Specifically, for each signal Sig_i (i.e. for each row i in the score matrix), we rank each sentence score ($sc_{i,j,1\leq j\leq N}$) in increasing order. That is, the first sentence in the ranked list corresponds to the sentence with the highest score, and the last with the lowest score. Fig. 14 shows the position of the correct sentence in the ranked list for a set of $N = 35$ sentences. Recall that in practice, about a dozen messages are posted every minute. We nevertheless raise the bar to up to $N = 35$ messages for our evaluation. For sentences containing 3 words, the correct sentence appears in first position about 63% of the time. This increases to 77% and 86% for sentences containing 4 and 5 words respectively.

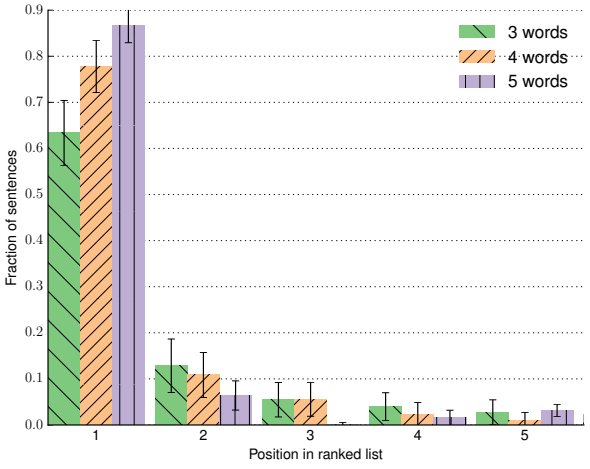

Fig. 14. Position of correct sentence in ranked list (35 sentences).

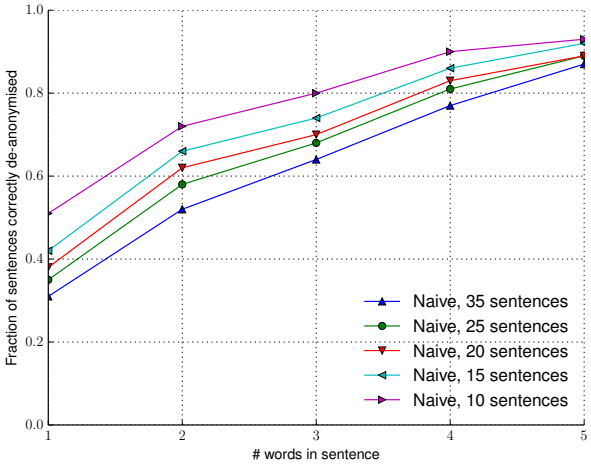

Fig. 15. De-anonymization of sentences using the "naive" method.

Naive Re-Identification Algorithm. Given a signal Sig_i and its corresponding list of scores $sc_{i,j,1\leq j\leq N}$, a simple solution to de-anonymize users is to select the top score in the list. We call this solution the naive solution. In our experimental setup, we randomly select N sentences each containing L words. Then we run the naive algorithm. We repeat this 200 times and average the results. These are presented in Fig. 15. For example, for sentences containing 5 words and for a set of 35 sentences, we correctly re-identify their author 86% of the time. This is consistent with the results of Fig. 14. In order to increase readability, and since the mean error between individual runs was always below 10%, we omit error bars. As the number of sentences in the set decreases, the results improve: for a set of 10 sentences each containing 5 words, we reach 92% de-anonymization. Intuitively, the more words a sentence contains, the more information we have about it. Therefore, as the number of words increases in a sentence, the re-identification improves (Fig. 15). We next show how to improve these results significantly.

Optimal Re-Identification Algorithm. Naive re-identification is not optimal, so here we describe a better method. Our goal is to maximize the sum of the scores when selecting sentences corresponding to signals. Looking back at the score matrix (Fig. 13), this means our goal is to select a set of optimal $scores_{i,j}$. Since each sentence corresponds to a single signal, each row and column must have exactly one score selected; and the sum of the selected scores must be optimal.

Practically speaking, this means there are $N!$ possible assignments to test. For $N = 20$ sentences, this means more than $10^{18} \approx 2^{60}$ candidates; and for $N = 35$ sentences, this means $10^{40} \approx 2^{130}$ candidates. This is greater than the strength of a 1024-bit RSA key (2^{80}). Fortunately, our problem is equivalent to the so-called "assignment problem" [17] for which there exist solutions that run linearly in the size of the input, i.e. in $\mathcal{O}(N)$. More specifically, we use the *Munkres* algorithm [17] that runs in $\mathcal{O}(N)$ with $\mathcal{O}(N^2)$ space requirements.

As in the naive method, we randomly select N sentences each containing L words and run the *Munkres* algorithm. We repeat this 200 times and average the results. These are presented in Fig. 16. For a set of 35 sentences each containing 4 words, we correctly guess the author of more than 92% of the signals (it is around 77% with the naive method). Regardless of the set size, we correctly guess the author of more than 97% of the signals when sentences contain 5 words.

4 Countermeasures

App-level. At the app level, we are limited. An app cannot disable gesture typing from the default keyboard app. However Android allows arbitrary apps to include their own custom keyboard layout/code through the *KeyboardView* API. This way an app could provide its own keyboard without the gesture typing feature. However, re-implementing a keyboard can be tedious, and removing gesture typing could greatly inconvenience users who have grown accustomed to it.

Zhang *et al.*[18] suggest killing apps that may be collecting side-channel information in the background

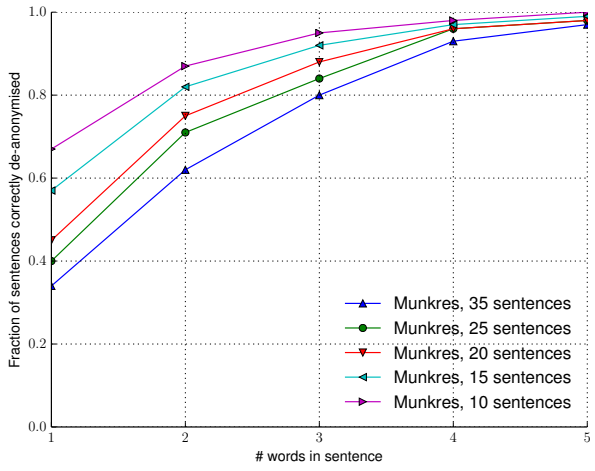

Fig. 16. De-anonymization of sentences using "Munkres algorithm".

while the foreground app performs sensitive tasks. This provides some level of protection without changing the OS or the apps being protected. However it comes with several caveats. First, their approach relies on the assumption that a malicious app must monitor resources at high frequency to be successful. But Michalevsky *et al.* [19] show that high sampling is not always necessary: by sampling the power consumption once a second, they can infer the route driven by a user (we discuss subsampling in the case of our attack later). Second, their techniques only protect foreground apps, not background processes. Third, they rely on monitoring app-specific procfs files; these are no longer accessible in Android M.

We conclude that app-level countermeasures are fragile and limited, so we investigate OS-level countermeasures next.

OS-level. OS-level countermeasures are more reliable since the OS can enforce a global policy that an app cannot. On Android, there have been inconsistencies between what resources are available through the framework APIs vs. those available through virtual files. The framework APIs enforce the permission model but the same is not always true for virtual files – certain permissions can be bypassed. For example, the virtual file */proc/net/arp* exposes the BSSID (i.e. the MAC address) of the WiFi Access Point a phone is currently connected to. This allows a curious app on the phone to find the location of a user's phone without requiring location permissions [4]. There are other pieces of information available through app-specific and global files in the virtual file system *procfs* (as well as */sys*). These represent the main source of leaks and inconsistencies that break the permission model. Therefore, we advocate restoring consistency, that is, we advocate prohibiting access to any virtual files (except those "owned" by the requesting app), perhaps through a stricter SELinux policy. Recall from Section 2.2 that the SELinux policy still allows access to certain global virtual files as of Android M. Of course, denying access to global virtual files could break apps that rely on them. In practice, we think this should affect only a very small number of apps, if any, as global virtual files exposed by *procfs* only provide admin-like information for troubleshooting, rather than relevant information for mobile apps. A trade-off could be to allow users to toggle this feature on and off for certain apps through an additional option within the "Developer" menu in phone Settings.

When we responsibly disclosed this work to Google, it became clear that they worried that protecting global procfs entries could break some utility apps. So might it be possible to have the OS rate-limit virtual file access, rather than prohibit it entirely? We study this for both attack scenarios in the following sections.

Rate-Limiting to Protect SoIs. Recall from Section 3.3 that in this scenario, an attacker has a predefined set of sentences of interest (SoIs), and wants to detect when these are entered by a user. Our current attack relies on the ability of an attacker to chop the sentence signal into its constituent word signals. For this, we used the zero-speed events extracted from the screen's interrupt counter (Section 2.3). If all word signals look "enough" like the words of an SoI, we output a match. Therefore one way to defend against our current implementation is to make it infeasible for an attacker to correctly infer the number of words entered by a user. Note however that this may not thwart more advanced attacks that build a fingerprint based on the entire sentence signal rather that its constituent word signals.

Fig. 17 shows the effect that subsampling (i.e. rate limiting) has on our detection routine. With a reduction of the sampling rate by 2 ("2 subs"), we correctly detect the number of words in an 8-word sentence about 80% of the time only. This drops down to 10% for a 10-fold reduction, which corresponds to about $10ms$ on a Nexus 5. Therefore a 10-15ms rate-limiting policy appears to already provide good security.

However, looking back at our data, we found that we could improve our original word-splitting routine to use the software interrupt rather than the screen hardware interrupt. Fig. 18 illustrates the effect that subsampling has on the first derivative of the software interrupt counter. The top signal shows the original signal corresponding to a 4-word sentence. Even with a 500-fold reduction of the sampling rate, the number of words

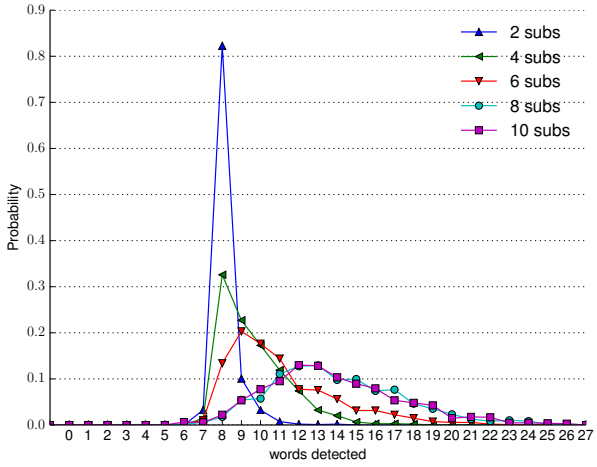

Fig. 17. Distribution of the number of words detected using the screen interrupt counter, for sentences containing 8 words.

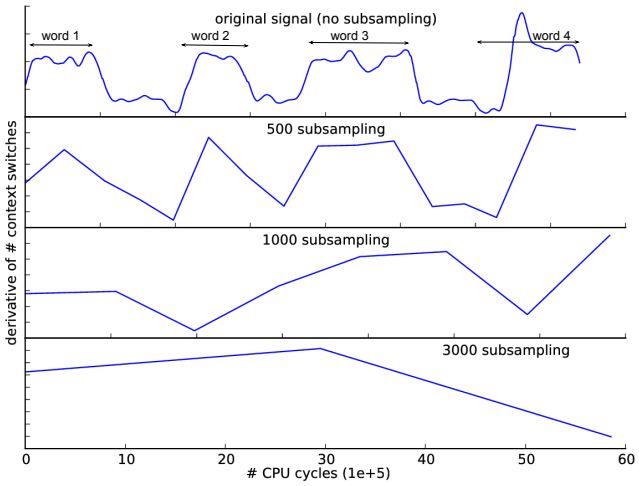

Fig. 18. Effect of subsampling on the first derivative of the number of software interrupts measured by a malicious app.

is till clearly visible. With a 1000-fold reduction, the detection becomes unreliable, and appears impossible with a 3000-fold reduction. Fig. 19 shows the number of words detected by our new routine subjected to subsampling (for all samples collected from users). Even with a 200-fold sampling rate reduction, more than 60% of the time we correctly detect the number of words (8). As we further reduce the sampling rate, the number of words detected moves towards zero. But even with a 3000-fold reduction of the sampling rate, we detect the presence of one word (50% of the time) rather than no word at all. A rate limit of 1.4-3s ("1400 subs"-"3000 subs" in Fig. 19) thwarts our attack on SoI detection, since with such sampling rates we never correctly detect the number of words in a sentence. Of course, this only defeats our current implementation, and it would be more prudent to prohibit access to virtual files entirely as suggested earlier.

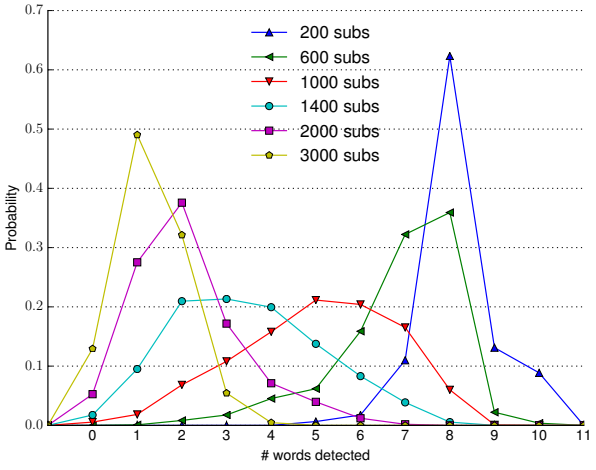

Fig. 19. Distribution of the number of words detected using the software interrupt counter, for sentences containing 8 words.

Rate-Limiting to Protect User Anonymity. In this scenario, we have a list of sentences posted on an "anonymous" messaging board by a set of users. We try to determine which user entered which sentence. We have extensively studied the re-identification of users when sentences contain the same number of words, as this is the worst-case scenario for an attacker (Section 3.4). If we apply the 1.4s rate limiting policy as for SoI detection, an attacker can no longer detect the number of words reliably, so this seems to thwart re-identification attacks on sentences with the same number of words. Recall that without the right number of words, we cannot extract where words start and end in the signal stream, as a result of which we cannot extract the features necessary for fingerprint. The 1.4s rate, however, is still not enough if sentences contain a different number of words. As we mentioned in Section 3, in this case an attacker need not train on users, but only detect the number of words. Let us consider 2 signals Sig_1 and Sig_2 corresponding to 2 sentences Sen_1 and Sen_2 containing $L_1 = 3$ and $L_2 = 8$ words respectively. Even if an attacker cannot reliably infer the number of words in each $Sign_i$, she may be able to re-identify users solely based on the estimated number of words detected. Fig. 20 shows the number of words detected for sentences containing 3 and 8 words respectively, when subjected to subsampling. For example, for a 1000-fold reduction of sampling rate, if an attacker detects 5 words, she is sure the signal corresponds to Sen_2 containing 8 words, since our routine never detects more

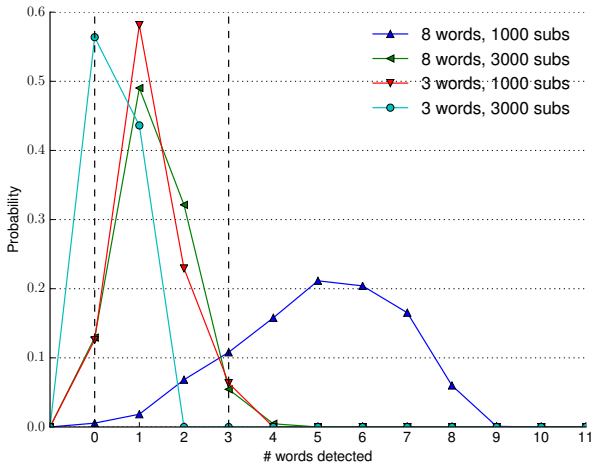

Fig. 20. Distribution of the number of words detected using the software interrupt counter, for sentences containing 8 and 3 words.

than 3 words for a signal corresponding to a 3-word sentence. So let us consider the following re-identification heuristics:

$$Sig_i = \begin{cases} Sen_1, & \text{if } N_{detectedWord} \leq N_{cutt-off}, \\ Sen_2, & \text{otherwise.} \end{cases} \quad (6)$$

where $N_{cutt-off} = 3$ and $N_{cutt-off} = 0$ for 1000-fold reduction and 3000-fold reduction of sampling rate respectively. This allows an attacker to correctly re-identify users 84% and 43% of the time respectively. In other words, even a rate limit of 3s is not enough to thwart re-identification attacks that are based on the number of words and, in effect, on phrase length. We conclude that finding the right rate that thwarts all attacks – including those still unknown to us – is non-trivial. It is more prudent to simply prohibit access to virtual files as we first suggested.

5 Discussion

The dictionary set we used was limited by experimental constraints (Section 3). In terms of dictionary size, the practical limiting factor was the burden put on study participants. We were unable to test our techniques for larger set sizes. This is an area that will no doubt attract more work in the future.

We also attempted to recover arbitrary words. However, this turned out to be challenging. We investigated language models in combination with the RNN, but did not achieve satisfying results. One major issue was the lack of a large chat dataset to create a language model. We think our corpus was too small. We thought of using other sources of chat data such as Twitter, but were put off by the restrictions on the use of the data. In fact, Twitter now blocks large-scale downloads entirely to prevent data mining.

For now, we have demonstrated that aggregated interrupt counters consitute a threat. We are confident that our results could be improved in two ways. First, as noted above, a larger chat corpus would help an attacker develop good language models. These are typically trained over millions of entries, while our corpus had only a few thousand. It would be very valuable to understand the full implications of such attacks under ideal conditions.

Second, we believe fine tuning the classifier parameters could also improve our results. More data from users could help to take advantage of deep learning capabilities of the RNN. Again, artifical neural networks excel at understanding complex data relationship through millions of samples, but we trained ours with less than 20 occurrences of each word for each user. The restricted number of participants made it difficult to assess the feasibility of creating a "master" model that works for most users to avoid the per-user training phase. Realisticallly, this may require hundreds or even thousands of users. It was out of scope of this paper, since we focused on piloting the attack, rather than scaling it. Consequently, with our current implementation, predictions made through a model trained on one user simply do not work for another. But the history of HCI suggests that, with enough users, we can move from user-dependent recognition to user-independent operation.

As well as scaling the attack across users, it may be worth while trying to scale the attack across devices. We demonstrated it on the Nexus 5, but we are confident it will generalize; we have looked at various phone models, and their counter streams all show similar properties during user input. However, different phones have different hardware and software characteristics. So could we build a model by training users on one phone, and predict text entered on a different one? This is another topic for further research.

As detailed in Section 3, our current implementation collects data on phones, but data pre-processing, fingerprint creation and prediction are performed offline on a desktop. But the processing power of current smartphone is not a barrier; we could do all the data processing locally. Model creation on a standard desktop takes a few minutes at most with code written in python. We

believe the same computation would be feasible on a smartphone if we reimplemented it in C, albeit with a slight performance hit. The prediction phase is a lot faster and takes at most a few seconds on our desktop. Software that can collect data and build a model locally opens up the prospect that a malware writer could collect data at scale and use it for the methodological improvements described here, including building a better language model and developing user-independent recognition techniques.

Stealthy malware must blend in its environment in terms of energy consumption, network activity, and data storage. Energy consumption during data collection would be negligeable because the curious app monitors input only when needed. There is no busy loop constantly executing to drain the battery. Where energy consumption could become an issue is during model training if this was done on the phone. To be stealthy, malware could do this in steps, rather than all at once, e.g. by spreading the computation over several days, or doing it only when the phone is plugged in to a power socket. In terms of storage, models take a bit over 1MB at rest (i.e. compressed) and about 3MB when used (uncompressed). Data collected for the training phase is about 40MB when compressed (Section 3). Although this is not small, neither is it big enough to make users suspicious. In the case where data processing is offloaded to a server, network activity should also be camouflaged, e.g. by uploading data over several days, or only when in wifi.

We use phones with around 60 apps for our evaluation. The apps generate noise for the counters we monitor. Our results show that under normal conditions, this noise is negligible. We also investigated our attack under heavy load while the browser was downloading a 100MB file. The download spanned the entire user input. Under this condition, our attack did not work. So our current implementation is very sensitive to ambient noise. But this does not mean that all such attacks will be. Even if the side channel turns out to be inherently sensitive to high loads, users generally only interact with one app at a time on smartphones. So maybe most of the time the system is under low load and the attack remains feasible, or maybe the curious app can just discard data when heavy load is detected.

Another interesting question is how we could combine sensor-based side channels with our attack. This is something we have not attempted yet. Another kind of attack that may become possible through monitoring interrupt counters is inference of what users type on normal keyboards. Attacks based on keystroke dynamics would benefit from the interrupt-based side channel described in this paper. The screen's hardware interrupt counter may also be used to infer other user activities on the screen. It could also have been used, for example, by PIN Skimmer [20] – where the authors used the microphone to detect user taps.

In short, the attack we have presented in this paper is really just an early prototype, and can probably be improved and extended in all sorts of interesting ways. Rather than waiting for this to happen, it may be prudent to tackle the problem now.

6 Related Work

Side channels have been widely studied for many years. Power-analysis side channels, introduced by Kocher [21], recover keys by monitoring power consumption during cryptographic operations. Cryptographic keys can also be recovered through cache-timing attacks if the code path or data access is data-dependent, e.g. unprotected RSA modular exponentiation [22] or AES table lookups [23, 24]. Acoustic and EM side channels have also been used to recover what people type on physical keyboards [25, 26] and data being printed [27].

Mobile sensors and peripherals also carry their share of novel side channel risks. Mäntyjärvi *et al.* [28] use the accelerometer to identify users through their gait. Michalevsky *et al.* [29] infer a person's gender by recovering spoken words using smartphone accelerometers. Sarfraz *et al.* [30] infer a user's location through their smartphone accelerometer. Michalevsky *et al.* [19] also infer a person's location by monitoring the power consumption of their smartphone. TapLogger [31], TouchLogger [32], TapPrint [33], and Aviv *et al.* [34] infer a PIN entered on a smartphone by monitoring phone motion inferred from real-time accelerometer data. Simon and Anderson [20] work out the PIN using the phone camera to infer device motion during user input. Dey *et al.* [35] fingerprint devices based on their accelerometer characteristics. Marquardt *et al.* [36] show how a smartphone app can abuse accelerometer readings to infer text entered on a nearby keyboard.

Another category of side-channels are those based on protocols. Cache [37] statistically fingerprints 802.11 implementations through their duration field. Nmap [38] sends a series of network packets to a machine and infers its operating system based on its network stack behaviour.

Another category of side-channels are those based on radio. Perta *et al.* [39] abuse the different radio states of cellular phones to infer the phone IP address; the time it takes a phone to reply to incoming traffic depends on its level of radio activity. Brik *et al.* [40] identify the source network card of an IEEE 802.11 frame through passive radio-frequency analysis.

Another category of side-channels are those based on traffic. Stöber *et al.* [41] identify smartphones based on their traffic. Conto *et al.* [42] infer a smartphone user's activity based on their internet traffic. More generally, internet packet length and timing characteristics are also used for webpage fingerprinting [43] and to identify protocols such as Tor [44]. Traffic analysis can also be used to infer voice content in encrypted VoIP traffic [45, 46].

At the API level, Zhou *et al.* [4] infer the location of a user's smartphone through its software voice assistant. Specifically, they exploit the mutually exclusive access control of the audio API to infer the length of spoken words, and from these deduce the directions taken by a user.

The last category of side channels are those based on virtual filesystems such as *procfs*. Zhang and Wang [2] demonstrate how an app's stack pointer exposed by *procfs* can be used to fingerprint keystroke events and infer a user's password. Jana and Shmatikov [47] infer information contained in a web page by monitoring the memory footprint of the web browser while it loads the page. More recently, Zhou *et al.* [4] show how traffic information gleaned through virtual files can be used to fingerprint the Twitter app traffic and identify a Twitter user.

To the best of our knowledge, our work is the first that uses real-time hardware and software interrupts to infer what users actually type on their phone.

7 Conclusion and Future Work

We present a novel attack against Android soft keyboards that support gesture typing. By monitoring the number of screen hardware interrupts and aggregated software interrupts during user input, we show that it is possible to breach a user's privacy, both by identifying some sentences a target user types and by identifying which user typed some incriminating sentence.

To the best of our knowledge, our work is the first to exploit global information exposed by *procfs*. This falsifies the general belief that non-app-specific information exposed through virtual files is harmless. Our attack works on the latest Android version where app-specific virtual files are inaccessible.

We investigate the efficiency of rate limiting as a countermeasure. We find that determining a proper rate limit is nontrivial and fails in subtle use cases. Therefore we advocate removing access to global virtual files in the next Android version.

Future work could improve the performance of our attack by using a larger chat corpus to build better language models, and by using more user data to train our recogniser better and to develop a user-independent recogniser. It could also study whether our techniques can be applied to normal (non-swipe) keyboards, e.g. using keystroke dynamics. Another area worth investigating is the extent to which we can infer a user's activity through the side channel presented in this paper. Finally, more work is required to assess the security and privacy implications of global virtual files under /proc, /sys, etc.

8 Acknowledgements

We thank the anonymous reviewers for their valuable suggestions and comments. We also thank Dongting Yu for his valuable suggestions regarding the deanonymization of users. This work was partially supported by the Samsung Electronics Research Institute (SERI), Thales, and the Carnegie Trust for the Universities of Scotland.

References

[1] A. T. Ozcan, C. Gemicioglu, K. Onarlioglu, M. Weissbacher, C. Mulliner, W. Robertson, and E. Kirda, "BabelCrypt: The Universal Encryption Layer for Mobile Messaging Applications," in *Financial Cryptography and Data Security (FC)*, 01 2015.

[2] K. Zhang and X. Wang, "Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems," in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, (Berkeley, CA, USA), pp. 17–32, USENIX Association, 2009.

[3] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: UI state inference and novel android attacks," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pp. 1037–1052, 2014.

[4] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, lo-

cation, disease and more: Inferring your secrets from android public resources," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), pp. 1017–1028, ACM, 2013.

[5] A. Savitzky and M. J. E. Golay, "Smoothing and Differentiation of Data by Simplified Least Squares Procedures.," *Anal. Chem.*, vol. 36, pp. 1627–1639, July 1964.

[6] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model.," in *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pp. 1045–1048, 2010.

[7] T. Mikolov, S. Kombrink, L. Burget, J. H. Černockỳ, and S. Khudanpur, "Extensions of recurrent neural network language model," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pp. 5528–5531, IEEE, 2011.

[8] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[9] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[11] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[12] "Android apps in sheep's clothing." http://www.modzero.ch/modlog/archives/2015/04/01/android_apps_in_sheeps_clothing/index.html.

[13] "Currentapp.java." https://gist.github.com/jaredrummler/07a3f723e96ec06fb761.

[14] "Activitymanager." https://developer.android.com/reference/android/app/ActivityManager.html#getRunningTasks%28int%29.

[15] E. N. Forsythand and C. H. Martell, "Lexical and discourse analysis of online chat dialog," in *Proceedings of the International Conference on Semantic Computing*, ICSC '07, (Washington, DC, USA), pp. 19–26, IEEE Computer Society, 2007.

[16] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st ed., 2009.

[17] J. Munkres, "Algorithms for the assignment and transportation problems," 1957.

[18] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on android," 2015.

[19] Y. Michalevsky, G. Nakibly, A. Schulman, and D. Boneh, "Powerspy: Location tracking using mobile device power analysis," *arXiv preprint arXiv:1502.03182*, 2015.

[20] L. Simon and R. Anderson, "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 '13, (New York, NY, USA), pp. 67–78, ACM, 2013.

[21] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, (London, UK, UK), pp. 388–397, Springer-Verlag, 1999.

[22] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, (London, UK, UK), pp. 104–113, Springer-Verlag, 1996.

[23] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, (Berlin, Heidelberg), pp. 1–20, Springer-Verlag, 2006.

[24] D. J. Bernstein, "Cache-timing attacks on aes," tech. rep., 2005.

[25] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards.," in *USENIX security symposium*, pp. 1–16, 2009.

[26] L. Zhuang, F. Zhou, and J. D. Tygar, "Keyboard acoustic emanations revisited," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 3, 2009.

[27] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers.," in *USENIX Security Symposium*, pp. 307–322, 2010.

[28] J. Mäntyjärvi, M. Lindholm, E. Vildjiounaite, S. marja Mäkelä, and H. Ailisto, "Identifying users of portable devices from gait pattern with accelerometers," in *in IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2005.

[29] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 1053–1067, USENIX Association, 2014.

[30] S. Nawaz and C. Mascolo, "Mining users' significant driving routes with low-power sensors," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, (New York, NY, USA), pp. 236–250, ACM, 2014.

[31] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pp. 113–124, ACM, 2012.

[32] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion.," in *HotSec*, 2011.

[33] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: your finger taps have fingerprints," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pp. 323–336, ACM, 2012.

[34] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith, "Practicality of accelerometer side channels on smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 41–50, ACM, 2012.

[35] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi, "Accelprint: Imperfections of accelerometers make smartphones trackable," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[36] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(sp) iphone: decoding vibrations from nearby keyboards using

mobile phone accelerometers," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 551–562, ACM, 2011.

[37] J. Cache, "Fingerprinting 802.11 Implementations via Statistical Analysis of the Duration Field," tech. rep., 2006.

[38] "Nmap security scanner." https://nmap.org/. Accessed: 2015-07-31.

[39] V. C. Perta, M. V. Barbera, and A. Mei, "Exploiting delay patterns for user ips identification in cellular networks," in *Privacy Enhancing Technologies*, pp. 224–243, Springer, 2014.

[40] V. Brik, S. Banerjee, M. Gruteser, and S. Oh, "Wireless device identification with radiometric signatures," in *Proceedings of the 14th ACM international conference on Mobile computing and networking*, pp. 116–127, ACM, 2008.

[41] T. Stöber, M. Frank, J. Schmitt, and I. Martinovic, "Who do you sync you are?: smartphone fingerprinting via application behaviour," in *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pp. 7–12, ACM, 2013.

[42] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Can't you hear me knocking: Identification of user actions on android apps via traffic analysis," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pp. 297–304, ACM, 2015.

[43] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 191–206, IEEE, 2010.

[44] S. Khattak, L. Simon, and S. J. Murdoch, "Systemization of pluggable transports for censorship resistance," *arXiv preprint arXiv:1412.7448*, 2014.

[45] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson, "Spot me if you can: Uncovering spoken phrases in encrypted voip conversations," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 35–49, IEEE, 2008.

[46] A. M. White, A. R. Matthews, K. Z. Snow, and F. Monrose, "Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 3–18, IEEE, 2011.

[47] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 143–157, IEEE, 2012.