TapLogger: Inferring User Inputs On Smartphone Touchscreens Using On-board Motion Sensors

Zhi Xu
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, USA
zux103@cse.psu.edu

Kun Bai
IBM T.J. Watson Research Center
Hawthorne, NY, USA
kunbai@us.ibm.com

Sencun Zhu
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, USA
szhu@cse.psu.edu

ABSTRACT

Today's smartphones are shipped with various embedded motion sensors, such as the accelerometer, gyroscope, and orientation sensors. These motion sensors are useful in supporting the mobile UI innovation and motion-based commands. However, they also bring potential risks of leaking user's private information as they allow third party applications to monitor the motion changes of smartphones.

In this paper, we study the feasibility of inferring a user's tap inputs to a smartphone with its integrated motion sensors. Specifically, we utilize an installed trojan application to stealthily monitor the movement and gesture changes of a smartphone using its on-board motion sensors. When the user is interacting with the trojan application, it learns the motion change patterns of tap events. Later, when the user is performing sensitive inputs, such as entering passwords on the touchscreen, the trojan application applies the learnt pattern to infer the occurrence of tap events on the touchscreen as well as the tapped positions on the touchscreen.

For demonstration, we present the design and implementation of *TapLogger*, a trojan application for the Android platform, which stealthily logs the password of screen lock and the numbers entered during a phone call (e.g., credit card and PIN numbers). Statistical results are presented to show the feasibility of such inferences and attacks.

Categories and Subject Descriptors

C.2.m [**Computer-Communication Networks**]: Miscellaneous; C.2.0 [**General**]: Security and protection

General Terms

Security, Design, Experimentation

Keywords

Smartphone, Trojan, Motion Sensor, Accelerometer Sensor, Orientation Sensor, User Inputs Logger, Android

1. INTRODUCTION

Today's smartphones are equipped with various embedded motion sensors, such as the accelerometer, gyroscope, and orientation sensors [40]. These motion sensors are used to gauge the smartphone's status of motion, such as orientation, acceleration, and direction. They are useful in supporting the innovative mobile UI [2], context awareness [31, 19], and motion-based commands(e.g., shuffling songs [1]). Probably due to the assumption that data collected by motion sensors is not sensitive, so far third party applications are allowed to access the readings of embedded accelerometer and orientation sensors without any security permission requirements on all Android [9], iOS [23], and Blackberry [4].

As the use of motion sensors in mobile device has become more widespread, serious concerns have been raised about the potential risks of a user's private information from being leaked through an installed third party application which explores these motion sensors. Recently, Marquardt et al. [28] presented a spying application *(sp)iPhone* which utilizes the vibrations sensed by a smartphone to infer the user inputs on a nearby keyboard. Cai and Chen [5] proposed a motion-based side channel for inferring keystrokes on the smartphone touchscreen. Such security and privacy exposures threaten to stifle the adoption and acceptance of these applications and even the smartphone platforms [39].

In this paper, we explore the feasibility of inferring a user's inputs on the smartphone touchscreen using sensor data collected from motion sensors. Our work is based on the observed correlations between the tap events and the motion change of smartphone. First, during a tap event, the acceleration of smartphone will change caused by the force from finger on the touchscreen. The change of acceleration follows certain patterns that can help the installed trojan application to detect the occurrences of tap events.

Second, tapping different positions on the touchscreen will cause small, but discernable changes of guesture of smartphone by the sensors. For example, tapping on the left side of touchscreen may cause the smartphone to turn left while tapping on the right side may cause the smartphone to turn right. By observing the gesture changes during a tap event, the attacker may roughly infer the tapped position on the touchscreen. The inferred position may not be precise. However, if the attacker knows the context of tap events and the layout of current view on the touchscreen, he may be able to infer the user's inputs (e.g., the pressed number button) with the inferred tap position.

We begin this paper by providing a technical background on this developing threat, and then describe in detail the key contributions of our work, where:

- We show the unique patterns of tap events in terms of changes of acceleration of smartphone. With statistical approaches, such patterns may help the installed trojan application to detect the occurrence of tap events.
- We show the correlation between the tap position and the gesture change during one tap event. With knowledge about the layout of view displayed on touchscreen, we show the feasibility of inferring user inputs with observed readings from the orientation sensor during tap events.
- We present the design and implementation of *TapLogger*, a trojan application that utilizes observed sensor readings to stealthily log the user inputs on touchscreen. For demonstration, we present two types of attacks on Android: stealing the password of screen lock and logging the PIN number entered during a phone conversation.

2. TECHNICAL BACKGROUND

2.1 User Inputs on Touchscreen

The touchscreen is the primary user interface of a touch-capable smartphone, and it presents a target of opportunity for potential hackers and information thieves. When a user taps on the touchscreen, the display and its supporting hardware and firmware will report the coordinates of tap events to the operating system of the smartphone. The coordinates of a tap event together with knowledge of the application view currently displayed on the touchscreen determine the corresponding user input. For example, a tap event with coordinates within the boundary of a button displayed on the touchscreen stands for a tap action on this button. As the layout of many views are public and uniform, such as the layout of screen lock view shown in Figure 6, the coordinates of tap events become extremely sensitive.

To protect user inputs from being eavesdropped, there are rigorous restrictions on the receiver who is allowed to receive the tap events and their coordinate information. For example, on the Android platform, only the *view* that is focused and currently being displayed on the touchscreen will be allowed to receive the coordinate information [10]. Therefore, a third party application running in background (e.g., a *service*) can never receive the coordinate information when the user is tapping the touchscreen to unlock the smartphone with passwords or dial a number during a call.

2.2 Motion Sensors

Motion sensors embedded in smartphones offer the hacker a side channel into a user's mobile device, from which he/she may steal private data. Specifically, we refer to the on-board accelerometer and orientation sensors, which are two of the most commonly equipped motion sensors on commodity smartphones. Up to the most recent version of major platforms, e.g., iOS [23], Android [9], and Blackberry [4], accessing these two sensors requires no security permissions, and enables considerable accesses by a third party application to the underlying resources in the mobile device. As Android is the most popular and most exploited smartphone platform, we adopt the specification and terminology defined on the Android platform in this paper.

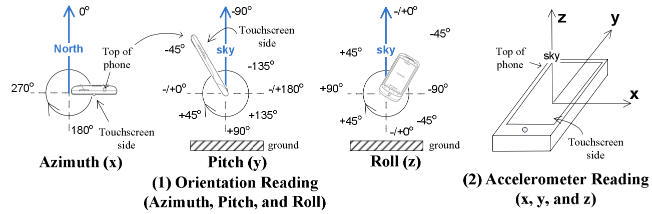


Figure 1: An introduction to accelerometer and orientation readings on the Android platform

2.2.1 *Accelerometer Sensor*

The accelerometer sensor monitors the current acceleration of a smartphone along three axes: left-right(i.e., lateral or x-axis), forward-backward (i.e., longitudinal or y-axis), and up-down (i.e., vertical or z-axis) [29, 9]. The returned readings of the accelerometer sensor are the rates of change of velocity with time along these three axes in m/s^2.

In Figure 1, we illustrate the mapping of axes in relation to a smartphone that is placed horizontally. As described in this figure, the reading of x-axis acceleration would be positive if the smartphone is moving toward the right side; the reading of y-axis acceleration would be positive if the smartphone is moving in the direction of the top of smartphone; the reading of z-axis acceleration would be positive if the smartphone is being lifted. For instance, when placing the smartphone statically and horizontally on a surface as shown in Figure 1, the reading of x and y-axis will be zero and the reading of z-axis will be the earth gravity (i.e., $9.8m/s^2$) because the surface is supporting the smartphone. When the smartphone is dropping freely, the readings of all axes will be zero.

To facilitate our discussion, we represent the current acceleration A of a smartphone by a vector $\vec{A} = < A_x, A_y, A_z >$ $=A_x\hat{x}+A_y\hat{y}+A_z\hat{z}$, where $\hat{x}$,$\hat{y}$, and $\hat{z}$ are the unit vectors in the directions of positive x, y, and z axes, respectively.

2.2.2 *Orientation Sensor*

The orientation sensor tracks the changes of orientation and gesture of smartphone along three dimensions: *Azimuth* (x-axis), *Pitch* (y-axis), and *Roll* (z-axis) [29, 9]. The returned readings of the orientation sensor are the current gesture of the smartphone in these three dimensions. Each reading is measured by degrees. As illustrated in Figure 1, the reading of x-axis is 0^o when the smartphone is facing north and the reading changes between 0^o and 360^o; the reading of y-axis is 0^o when the touchscreen side faces to the sky and the reading changes to $-/+180^o$ when facedown; the reading of z-axis represents the sideways tilt between -90^o and $+90^o$.

2.2.3 *Hardware Specifications*

Hardware specifications, such as the sensor sample rate, are important to sensing applications. Different smartphone platforms have different hardware specifications. The brief specifications of three selected smartphone platforms are listed in Table 1. As the orientation sample rate of Motorola Atrix is too low, in this paper, we presents experimental results on HTC Aria and Google Nexus (One).

2.3 Tap-To-Open/Active Interactive Pattern

Many interactive patterns exist for using touchscreens. In this paper, our threat model is based on the "*Tap To Open/Active*" pattern [34] which is almost the simplest but

Table 1: Specifications of selected smartphones

Model	Touch Screen Size (pixels)	Acc. Sample Rate	Ori. Sample Rate	Android
Aria	$W(320)\times H(480)$	$\sim 50Hz$	$\sim 50Hz$	v2.3
Nexus	$W(480)\times H(800)$	$\sim 25Hz$	$\sim 25Hz$	v2.3
Atrix	$W(540)\times H(960)$	$\sim 95Hz$	$\sim 8Hz$	v2.3

most used interactive pattern. With this pattern, a user taps in a specific position of the touchscreen to trigger a feature or to respond, similar to clicking with a mouse. With advanced hardware supports, some latest smartphones may support multi-touch touchscreen and more complex patterns, such as "*Drag To Move*" and "*Slide To Scroll*". However, in this work, our focus is the "*Tap To Open/Active*" pattern because of its extreme popularity as well as simplicity.

The typical scenario we consider here is that *a user holds the smartphone with her left hand and taps the touchscreen using the forefinger of her right hand*. Consider a tap event on the touchscreen that is face up. The motions of smartphone during one tap event can be briefly described as the three consecutive phases: *Action_Down*→ *Hold*→ *Action_Up*: when the user presses down on the touchscreen (i.e., in the *Action_Down* phase), the smartphone will be forced to move downward. When the *Action_Down* is over; the smartphone will stop (i.e., be in the *Hold* phase); Then, the user lifts his finger and the hand holding the smartphone will respond by lifting the smartphone back to its original position (i.e., in the *Action_Up* phase). The whole tapping action is usually performed in a short time period. According to our measurements, a swift tap event is about $50 \sim 70$ ms and an ordinary tap event is about $180 \sim 220$ ms.

In our discussion, we name the time duration of a tap event as a $SigWin$, representing the time duration when the finger is in touch with the touchscreen (i.e., between *Action_Down* and *Action_Up*).

3. ATTACK OVERVIEW

3.1 Assumptions

On the Android platform, a third party application must explicitly and statically declare the permissions for sensitive resources needed, such as networking and audio recording permissions. As motion sensors are considered as insensitive resource, *TapLogger* does not require any security permission to access the accelerometer and orientation sensors. However, $TapLogging$ requires the *networking* permission to send inferred user inputs to a remote attacker, and the *Phone Status* permission for context awareness. Both the *networking* and *Phone Status* permission are very common ones, which are also required by most of popular apps, such as *Angry Bird* [26], *Facebook* [14], and *Kindle* [25]. Therefore, *TapLogger* would have very little chance to draw attention with respect to security permissions.

3.2 Attack Goals

The target in our work is the meaningful user inputs, e.g., the button that is pressed during a tap event, instead of the precise coordinate of a tap event. Inferring the precise coordinate is practically infeasible due to interference factors, such as background noise. However, a button usually covers a zone in the screen and all user's tap events within this zone corresponding to the same user input. Thus, with knowledge about the layout of a target view, e.g., the one in Figure 6, the attacker can infer the user inputs (e.g., the button that is pressed) instead the precise coordinate of a tap event.

3.3 Attack Workflow

TapLogger works in two modes: *Training Mode* and *Logging Mode*. It switches to the training mode when the user is interacting with the *HostApp*, and switches to the logging mode when the user is performing sensitive inputs. The context related information can be retrieved from the Android operating system. Briefly, we explain the workflow as shown in Figure 2:

Training Mode: In the training mode, when the user is interacting with the *HostApp*, *TapLogger* can legally receive the coordinates of tap events on the touchscreen, and apply the coordinates with the readings of accelerometer and orientation sensors collected during tap events to generate a user's interaction pattern.

Thus, for one tap event detected in the training mode, the *HostApp* records related information including (1) the coordinate (on the screen); (2) the timestamps when it starts and when it ends; and (3) the accelerometer and orientation sensor readings between the start and the end time. The time duration between the start and the end consists a $SigWin$. The number of collected readings in the $SigWin$ depends on the time duration of tap events as well as the sensor sample rate. In addition, we also collect several readings before and after $SigWin$, denoted as $PreWin$ and $AfterWin$, respectively.

Logging Mode: In the logging mode, the $SensorListener$ runs in background and keeps stealthily monitoring the readings of accelerometer and orientation sensors. When the user is performing sensitive inputs, the acquired pattern can be applied to infer the tapped area (i.e., zone) on the touchscreen based on sensor readings collected by the *SensorListener* service. Then, with the knowledge about the layout of a targeted view, *TapLogger* can then infer the corresponding user inputs. To reduce battery consumption and false positives, the $SensorListener$ service only works in certain sensitive contexts.

3.4 Challenges

To make the proposed attack feasible, several challenges exist, which will be addressed one by one in the following sections.

- How to detect the tap events in the logging mode? An approach is needed for $SensorListener$ to identify the tap events in the logging mode by observing the readings of accelerometer and orientation sensors only in the background.
- How to infer the user inputs on the touchscreen based on observed sensor readings? A tap position classifier is needed to help the attacker to infer the tap positions and their implicated user inputs.
- How to design and implement *TapLogger* that is able to perform the proposed attack stealthily and efficiently? The trojan application should require as few permissions and workload as possible to launch the attack.

4. TAP EVENT DETECTION

In this section, we first introduce the unique change pattern of acceleration readings during tap events, and then show how to build the user's pattern in the training mode and how to apply the pattern in the logging mode.

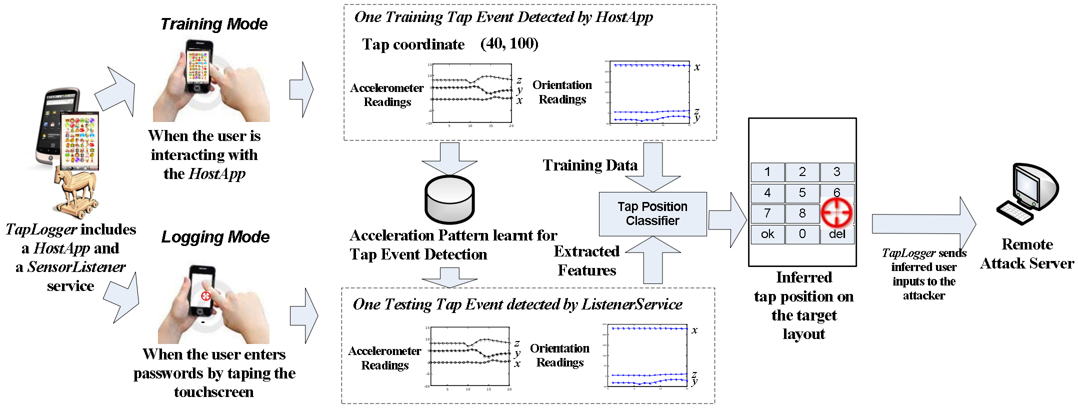


Figure 2: The attack overview

4.1 Observed Pattern of Tap Events

Accelerometers on smartphones have been widely exploited by context-aware applications to infer the current contexts of smartphones, such as transportation mode inference (i.e., inferring if the user is walking or taking vehicles [32, 38]) and activity recognition (i.e., inferring if the user is walking, sitting, or running [36, 37]).

Differently, in our application, our goal is not in context inference but in detecting the occurrences of tap events. These tap events are comparatively small fragments in the sequence of collected readings, thus existing context inference approaches do not work in our case.

To detect the tap events, we utilize the unique change pattern of external force on the smartphone during tap events. To measure the change of the external force F, we use the term $SqSum$ to denote the 2-norm of acceleration vector, i.e., $SqSum{=}|A|^2 = A_x^2 + A_y^2 + A_z^2$. Obviously, $SqSum$ represents the force on the smartphone and is directly proportional to $|F|^2$. When the smartphone is held statically in one hand, the value of $SqSum$ is equal to the square of gravity, i.e., $G^2 \approx 9.8^2$. When the smartphone is dropping freely, the value of $SqSum$ is 0.

In Figure 3, we show the measured $SqSum$ readings when the user is performing different activities. As shown in this comparison, when the user is tapping buttons shown in the touchscreen, the fluctuation of wave is much smaller. This is reasonable because the user will subconsciously stabilize the smartphone when he taps buttons on the touchscreen.

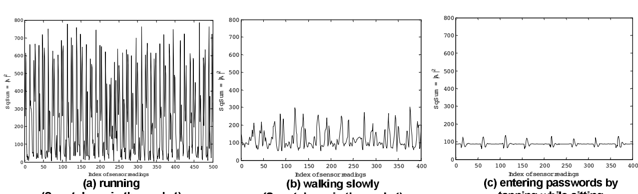


Figure 3: Acceleration readings in different contexts

Admittedly, the tap event detection will be difficult when the background noise is great. Some background noise may also generate acceleration patterns similar to tap event patterns. To lower the interference caused by background noise, *TapLogger* collects data for both training and logging attacks only when the background noise is small. A simple approach to measure the background noise is to calculate the standard deviation of the latest $SqSum$ readings. *TapLogger* collects data when the standard deviation is small. Other activity recognition approaches [36] can also be applied to prevent collecting data in noisy environments. Further, *TapLogger* only starts the *ListenerService* when in sensitive contexts, e.g., during a phone conversation. In this way, we greatly avoid false triggering.

4.2 Proposed Statistic Approach

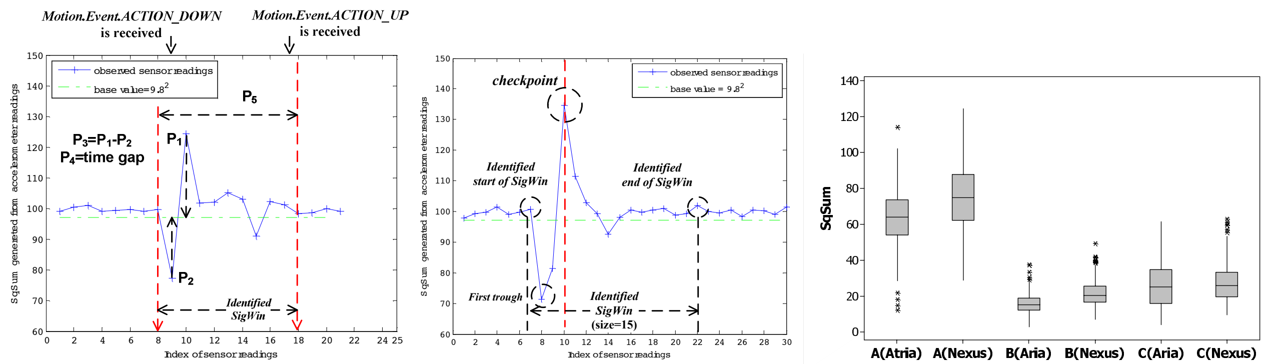
Based on the observed pattern of tap events, we propose a statistic approach for tap event detection by monitoring the change of force on the smartphone. Specifically, we describe the pattern of tap events by a set of statistic features in terms of $SqSum$ readings. For each feature, *TapLogger* learns its pattern interval through the training data collected by the *HostApp* in the training mode. These learnt pattern intervals are user specific representing a user's pattern of tap events. When *TapLogger* is in the logging mode, the *SensorListener* service monitors the acceleration of smartphone by keeping a monitoring window for a sequence of the latest $SqSum$ readings. If the features extracted from the readings of the monitoring window fall in the learnt pattern intervals, the *SensorListener* will consider a tap event having been detected. The $SigWin$ of a detected tap event is within the monitoring window.

4.2.1 Pattern Learning in Training Mode

The unique $\searrow\nearrow$ pattern of $SqSum$ in the *Action_Down* phase is the key to tap event detection in our case. As shown in the enlarged figure of a tap event in Figure 4(a), the $SqSum$ readings will first go down and then jump up dramatically corresponding to the first *Action_Down* phase in the *Tap-To-Open/Active* interactive pattern. The rest part of tap events represents the motion of smartphone when the user lifts up his finger from touchscreen. The jitter represents the case when the hand holding the smartphone is trying to stabilize the smartphone after the tap event.

To catch the unique patten of tap events, we first identify the start and the end of $SigWin$ by the timestamps of received events *Motion.Event.ACTION_DOWN* and *Motion.Event.ACTION_UP*, respectively. Then, we extract five features from the readings of $SqSum$ within the identified $SigWin$:

- P_1: The peak reading of $\searrow\nearrow$ at the end of *Action_Down* minus the $base$, i.e., $SqSum_{peak} - base$;
- P_2: The trough reading of $\searrow\nearrow$ of *Action_Down* minus the $base$, i.e., $SqSum_{trough} - base$;
- P_3: Difference between the peak and the trough readings, i.e., $SqSum_{peak}$-$SqSum_{trough}$;
- P_4: Time gap between peak and trough, i.e., $Time_{peak}$-$Time_{trough}$;
- P_5: The standard deviation of the entire $SigWin$, i.e., $Std(SigWin)$;



(a) Extracting features from a tap event in the training mode (b) Evaluating a tap event in the logging mode (c) The distribution of P_3 of different user on different smartphones

Figure 4: The proposed statistic approach for tap event detection

We briefly explain these measures as follows: $base = G^2$ denotes the $SqSum$ value when the smartphone is placed statically with no external force on it, P_1 measures the magnitude of force when pressing down, P_2 measures the magnitude of force from the hand holding the smartphone when reacting to the tap event, P_3 and P_4 measure the change rate, and P_5 measures the fluctuation. Each tap event is described by these five features in the tap event detection.

With all tap events detected in the training mode, *TapLogger* learns the distributions of these features for the current user on the current smartphone. According to our measurements in experiments, the distributions of these five measures form bell-shaped curves that are close to normal distributions. Thus, with the measurements in our training dataset, we take the range between the *Lower Extreme* and the *Upper Extreme* [33] of each measurement, represented by $[L, U]$, to describe the pattern interval of this measurement. Specifically, $L = Q_1 - 1.5 \times IQR$ and $U = Q_3 + 1.5 \times IQR$, where Q_1 is the *lower quartile*, Q_3 is the *upper quartile*, and $IQR = Q_3 - Q_1$ is the *interquartile range*.

In this way, the pattern for tap event detection can be presented by five sets of boundary values, i.e., P_S={I_1,I_2,I_3,I_4,I_5}, where $I_i = \{L_i, U_i\}$. Different people have different tapping patterns.

4.2.2 Tap Event Detection in Logging Mode

When *TapLogger* is in the logging mode, the *SensorListener* service detects tap events by observing the readings of accelerometer. Services running in background (e.g., *SensorListner*) are not allowed to receive touch event information from the touchscreen. Specifically, in the logging mode, the *SensorListner* service keeps a *MonitorWin*, which contains the last K observed *SqSum* values of acceleration. In our experiments, *TapLogger* calculates the average size of $SigWin$ in the training dataset, and sets K twice the average size. The *SqSum* readings in the *MonitorWin* are ordered by their generation times. We present an example of $MonitorWin$ in Figure 4(b).

Whenever a new *SqSum* reading is generated, the *SensorListner* service will first update the *MonitorWin* by adding the new *SqSum* reading, and then check the *SqSum* reading at a fixed index position named *checkpoint*, e.g., the 10th reading in the *MonitorWin* shown in Figure 4(b). The purpose is to check whether the current *SqSum* reading at the *checkpoint* is the peak point of the first ↘↗ of a tap event.

Basically, assume that the current *checkpoint* is the peak point of a tap event. The *SensorListner* will try to identify the corresponding trough point in the *MonitorWin*, extract features $P_1 \ldots P_5$, and check if the extracted features are within the intervals described in the learnt tap event pattern. The checking scheme is illustrated in Figure 4(b). The detailed checking algorithm is presented in Appendix A.

Table 2: Experimental results of tap event detection

User	Model	Precision	Recall	F-Measure
User A	*Aria*	93.6%	91.8%	92.74%
User B	*Aria*	76.3%	90.0%	82.6%
User C	*Aria*	70.4%	97.4%	81.7%
User A	*Nexus*	99.3%	96.3%	97.8%
User B	*Nexus*	74.67%	95%	83.61%
User C	*Nexus*	83.97%	88.37%	86.12%

4.3 Evaluations

In the evaluation, the tester students were first asked to play the *icon-matching game* for 30 rounds, collecting a training dataset of more than 400 tap events. With the collected training dataset, *TapLogger* extracts the proposed five features (i.e., $P_1 \ldots P_5$) from each tap event, and generates a statistic pattern for each dataset. These statistic patterns (i.e., boundaries of parameters) are user specific as well as device specific.

For example, in Figure 4(c), we present the distribution of the P_3 feature collected from training data of different users (i.e. user A, B, and C) on different smartphones. As shown in this comparison, first of all, most of measured P_3 feature falls in the range between the lower extreme and the upper extreme. Secondly, different users have different patterns. The more convergent the distribution of extracted features, the more consistent the pattern. Thirdly, the tap event patterns of the same user are different on different smartphones due to the difference in the hardware, such as the sensor sample rate and weight.

In the testing phase, the tester students were asked to enter a sequence of about 150 numbers in a testbed with UI the same as the number pad layout as shown in Figure 5. During the testing phase, *TapLogger* stealthily monitors the accelerometer readings in the background and used the corresponding statistic patterns to detect the tap events on the touch screen. The experimental results in Table 2 show that the proposed tap event detection approach can achieve high accuracies in identifying the tap events on the touchscreen.

5. TAP POSITION INFERENCE

The goal of tap position inference is to identify the tapped area on the touchscreen. In this section, we first show the

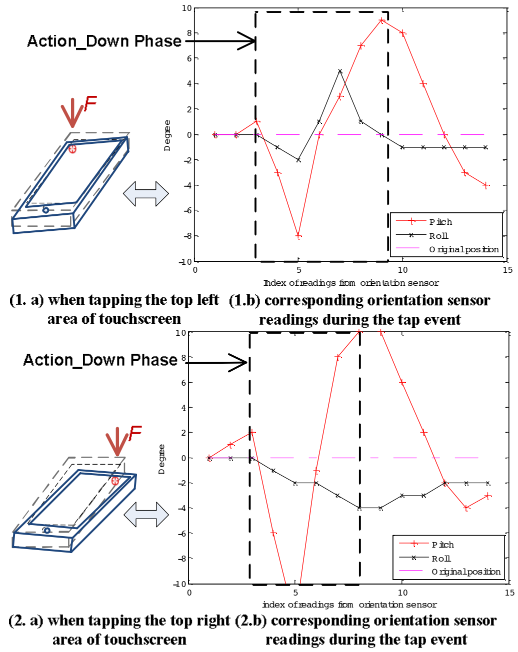


Figure 5: Examples of gesture changes when tapping different areas of the touchscreen

correlation between the position of a tap event and its corresponding gesture changes, measured by the orientation sensor. Then, we present a way of dividing the touchscreen into zones (i.e., tapped area) based on layout of the target view. Finally, with real tap events data collected in our experiments, we show the feasibility of distinguishing tap events in different zones. Limitations and reasons are also presented in the end to show the capacity of inferencing using this correlation.

5.1 Gesture Change Analysis

When a user taps on the touchscreen using his finger, the gesture of smartphone will change caused by the external forces from the finger and the hand holding the smartphone. The way that the gesture changes is related to the position on the touchscreen that has been tapped. Intuitively, when the user taps on the top of touchscreen, the head of smartphone will go down; while the user taps on the right side of touchscreen, the body of smartphone will lean right as well.

TapLogger senses the gesture changes through two types of readings from the orientation sensor: the readings of *Pitch*, measuring turning left or right of smartphone; and the readings of *Roll*, measuring the going down of head or bottom. As shown in Figure 5, when the user taps on the top left of touchscreen, the smartphone will turn left and its head will goes down. Correspondingly, the *Pitch* value goes up because the head goes down, and the *Roll* value goes up because the face of touchscreen turns left. When the tap event is done, the gesture of smartphone will go back to the approximate of its original gesture. For the comparison, we show the gesture change when tapping on top right in Figure 5(2.a). As shown in its corresponding orientation readings, the *Pitch* value goes up because the head goes down, but the *Roll* value now goes down because the face of touchscreen turns right.

Clearly, it would be ideal if one can infer the exact coordinates of tap events based on the observed gesture change of smartphone. However, due to noise and limitation on sensor sample rate, such exact inference is not possible. Instead of inferencing the exact coordinates, *TapLogger* divides the screen into meaningful zones and infers a zone on the screen that contains the exact coordinates.

5.2 Screen Division

We name the view on which the user is performing sensitive inputs as the *Target View*. In a target view, zones stand for meaningful objects, such as buttons. Therefore, if the attacker can correctly identify the pressed zones, he can get the user inputs on the touchscreen with knowledge about the layout of the target view. In Figure 6, we present the layout of two target views: one is the layout of screen lock view on which the user enters the password by tapping buttons; the other is the layout of number pad view on which the user enters his PIN or credit card number when making calls. As we can see in Figure 6, the layout can easily be divided into meaningful zones by individual buttons. The zones we are interested in are buttons used to enter the password or PIN.

To disguise its purpose, the attacker may also carefully design the user interface of *HostApp*. For example, in Figure 6(1), we present the screen layout of a *icon-matching game* (i.e., the *HostApp*). When playing the *icon-matching game*, the user may not realize that he is contributing training data for the target view in Figure 6(2) .

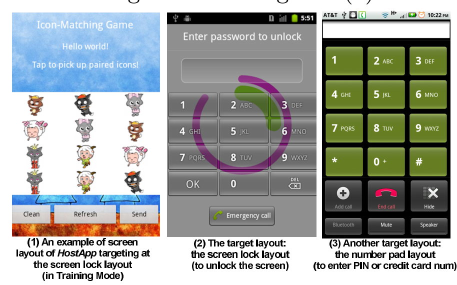


Figure 6: An example of *HostApp*'s screen layout and target screen layouts

5.3 Proposed Inference Approach

With the screen division, we convert the tap position inference problem to a classification problem, in which, each class is a labeled zone in the target view. In the training mode, *TapLogger* uses the collected training data to create a tap position classifier following the user's tapping pattern. While in the logging mode, *TapLogger* uses this trained classifier to infer the corresponding class (i.e., the zone or button). For each detected tap event, the output of classification in the logging mode is an estimated zone that contains its real coordinates.

5.3.1 Classifier Generation in Training Mode

Similar to the proposed tap event detection approach, *TapLogger* keeps monitoring the readings of the embedded orientation sensor. When a tap event is observed by the *HostApp*, *TapLogger* first determines the start and end of *SigWin* by events *Motion.Event.ACTION_DOWN* and *Motion.Event.ACTION_UP*, and then, generates the training data for the classifier from the sequence of orientation sensor readings during this *SigWin*.

The training data of an observed tap event consists of a vector $\{L, F_O\}$, where L is the label of zone containing its associated coordinates, and F_O are the set of features extracted from the orientation sensor readings during its *SigWin*. These extracted features measure the gesture changes, and are generated by changes of the orientation sensor readings in *Roll* and *Pitch* during the *Action_Down* phase. Briefly, we list the extracted features as follows:

- F_1: the change of *Roll* in the first monotonic section;
- F_2: the change of *Roll* in the second monotonic section; if *Roll* in the *Action_Down* phase is monotonic as a whole, F_2 is assigned 0.
- F_3: the change of *Roll* from the start to the end of *Action_Down* phase;
- F_4: the change of *Pitch* in the first monotonic section;
- F_5: the change of *Pitch* in the second monotonic section; if *Pitch* the *Action_Down* phase is monotonic as a whole, F_5 is assigned 0;
- F_6: the change of *Pitch* from the start to the end of *Action_Down* phase;

To explain, Features F_1, F_2 and F_3 help to determine if a tap event is on the left side of touchscreen or on the right side; F_4, F_5, and F_6 help determine if a tap event is on the top of touchscreen or on the bottom.

Note that, as shown in Figure 5, the changes of both Roll and Pitch may not be monotonic in the *Action_Down* phase. According to our observation in experiments, the reason is that, when the finger first contacts the touchscreen, two types of movements of the smartphone take place in parallel. One type of movement is that smartphone may move down as a whole, and the other type is the gesture change of smartphone. Thus, we separate the change of readings into two consecutive sections in which the change of readings is monotonic.

5.3.2 Inference in Logging Mode

In the logging mode, *TapLogger* also keeps a *OriMonitorWin* for the latest orientation sensor readings. When a tap event is detected by the observed *SqSum* readings, its *SigWin* will also be determined. With the start and the end of determined *SigWin*, *TapLogger* retrieves the sequence of orientation readings within the *SigWin* from the *OriMonitorWin*, and extracts features F_O from the retrieved orientation sensor readings. These extracted features are supplied to the classifier so as to infer the label of zone that contains the coordinates of this detected tap event.

5.4 Evaluation

The classifier for tap position inference is both user specific and device specific. To show the effectiveness of our selected features in classification, in Figure 7, we show three experimental results of distinguishing tap events at different positions based on extracted features. Specifically, Figure 7(1) and Figure 7(2) compare the experimental results of the same user on different smartphones, i.e., Aria and Nexus; Figure 7(2) and Figure 7(3) compare the experimental results of different users on the same smartphone. In both experiments, we collect about 60 tap events on each of 12 buttons on the number pad layout.

The experimental results in Figure 7 show that the proposed features $\{F_1, F_2, F_3\}$ (i.e., features measuring the change of *Roll*) can help to distinguish tap events on the left side of the touchscreen from those on the right side. Other features $\{F_4, F_5, F_6\}$ (i.e., features measuring the change of *Pitch*) can help to distinguish tap events on the top of the touchscreen from those on the bottom.

Further, by comparing the results of Figure 7(1) and Figure 7(2), we show the hardware factors affecting the accuracy of tap position inference. First of all, the data points in Figure 7(1) are more convergent because the sample rate of orientation sensor at Aria is about 50 Hz and is much faster than the 25Hz of Nexus. Secondly, despite its lower sample rate, the comparison between Figure 7(1.b) and Figure 7(2.b) shows that it is easier to distinguish buttons in the same column on Nexus. One reason is because the size of Nexus is bigger than the Aria, making the gesture changes in *Pitch* axis more obvious during tap events. Based on the comparison, we see that, besides the user pattern, the inference will be easier on smartphones which are bigger and more sensitive in sensing.

Moreover, Figure 7(2) and Figure 7(3) show the user factors affecting the accuracy of tap position inference. Specifically, Figure 7(2.a) and Figure 7(3.a) show that, although the patterns of different users are different, the tap events by the same user on different side of touchscreen are still distinguishable. However, the distinguishability is different from user to user. Take user B in Figure 7(3) as an example. Figure 7(3.a) shows that it is not difficult to distinguish the tap events of user B in the *Roll* axis. But, in Figure 7(3.b), distinguishing tap events by the *Pitch* axis is difficult. One reason may be that the Arial is small and light, making the gesture change not obvious. Another possible reason is that the way user B holding and tapping the smartphone is different from that of user A.

From the perspective of classification, Figure 7 also shows the difficulty of distinguishing tap events on neighboring buttons. For example, in the target screen lock layout, many tap events on button 9 may be falsely classified as its contiguous buttons, such as button 12 (i.e., the "*Del*" button), 8, and 6. One reason of error is that the number of collected sample readings is not enough to describe the gesture changes because of the limited sample rate. The other reason is that the actual tap position is very close to neighboring nodes, making the inference difficult to distinguish correctly. Therefore, in specific attacks, the attacker may apply additional background knowledge to improve the accuracy of inference. Indeed, in the two attacks shown later, we will use different approaches to improve the inference accuracy.

6. APPLICATIONS ON ANDROID

We have implemented *TapLogger* as an Android application with the attack flow presented in Figure 2. In this section, we show the detailed design of *TapLogger* on the Android platform. Further, two types of attacks with evaluations are also presented to show the effectiveness of *TapLogger* in logging and inferring sensitive user inputs.

6.1 Implementation on Android

TapLogger implements the proposed attack flow presented in Figure 2. The components of *TapLogger* on Android include: a *HostApp* activity, masquerading a benign application (e.g., the *icon-matching game*); a *SensorListener* service, collecting readings from sensors in the background; a *ContextIdentifier* service, starting/stoping the *SensorListener* service based on the context of smartphone; and a *Booter* Broadcast Receiver, starting the *ContextIdentifier* for the context awareness.

6.1.1 Training Mode

The *HostApp* activity starts the *SensorListener* service when it is brought to the foreground. In the training mode, *HostApp* logs coordinate information of tap events and the *SensorListener* service logs the sensor readings during these tap events. Both information are combined at *ContextIden-*

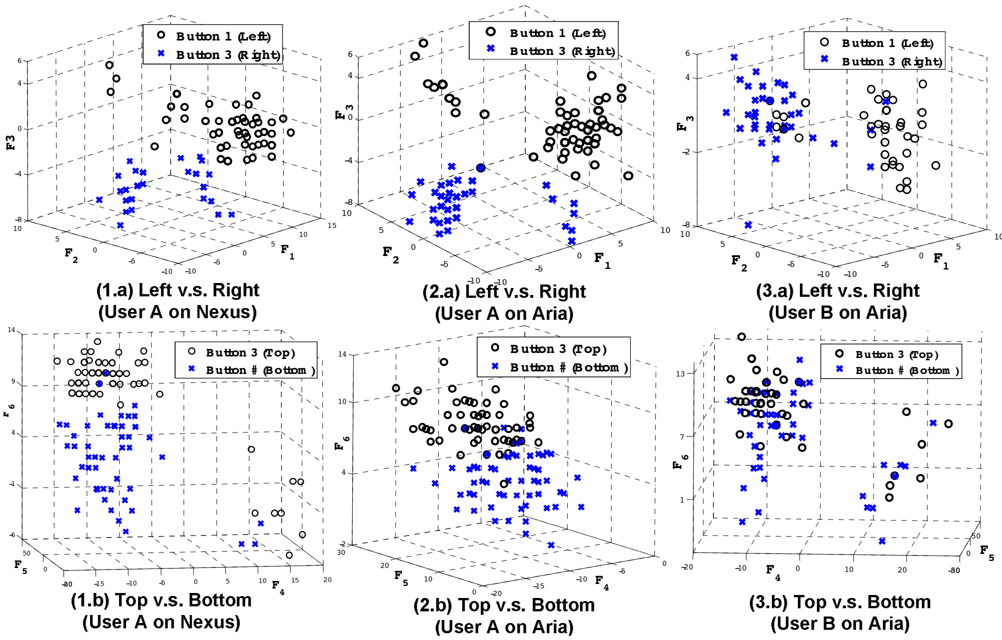


Figure 7: Examples of distinguishing tap events at different positions

tifier and then used as training data for pattern generation in tap event detection and for building the classifier for tap position inference.

6.1.2 Logging Mode

In the logging mode, the *Booter* will be automatically started when receiving the system broadcast (i.e., an Intent) *BOOT_COMPLETED*, indicting that the boot process of smartphone is complete. Thus, even the user has never started *TapLogger* after installation, *TapLogger* can still launch itself when the smartphone is started. Once started, the *Booter* launches the *ContextIdentifier* service which determines the current context of smartphone based on the Intents broadcasted by the Android operating system. By identifying the context, the *ContextIdentifier* services starts the *SensorListener* service only in a sensitive context.

In the end, only the results of inferencing, such as the sequence of labels of pressed buttons, will be sent to a remote attack server. Therefore, the data communication overhead for *TapLogger* is very small.

6.2 Number Pad Logging Attack

6.2.1 Attack Overview

During phone conversations, a user is often asked to enter valuable information on the number pad displayed on the touchscreen. Such valuable information includes the PIN numbers (e.g., requested by IP phone companies), social security numbers (e.g., requested by credit card companies or credit card centers), date of birth (e.g., requested by insurance companies).

In the number logging attack, we apply *TapLogger* to log the user inputs on the number pad during the call. Specifically, in the training mode, the attacker first collects training data by *HostApp* with the layout of number pad (shown in Figure 6), and then builds a classifier for tap position inference. In the logging mode, the *SensorListener* monitors the readings of motion sensors and performs the task of tap event detection and tap position inference. Specifically, the *SensorListener* will be invoked when the phone conversation starts (i.e., upon receiving *CALL_STATE_OFFHOOK* intent) and will be stopped when the phone conversation ends (i.e., upon receiving *CALL_STATE_IDLE* intent).

Real Inputs:		4840	2134	9596	1217	
Inferred labels:	**1st Label**	1043	5131	#869	1244	**0.3125**
	2nd Label	5*76	2567	9296	4511	**0.625**
	3rd Label	4810	4294	653#	7477	**1.0**
	4th Label	20*9	84#2	03#0	285*	**1.0**
						Coverage of true inputs

Figure 8: An example of tap position inference in the number pad logging attack

6.2.2 Tap Position Inference

In the number pad logging attack, we build the classifier by applying k-means clustering to the training data belonging to each label/button. Each label/button will be considered as one class. In the logging mode, given a tap event detected by the *SensorListener* service, we first extract features from the orientation sensor readings, and then measure the distances from this detected tap event to the trained clusters of all labels. Shorter distance means higher similarity to the cluster.

As buttons are close to each other, solely taking the label with shortest distance may cause high inaccuracy. Therefore, in the number logging attack, *TapLogger* may output several top ranked candidate labels for each detected tap event. An example of inference is presented in Figure 8 for demonstration. In this example, the real inputs are 16 numbers entered on the number pad layout shown in Figure 6. For each tap event, *TapLogger* outputs the top 4 candidate labels. The 1st label is the one with the closest distance in the k-means clustering. For example, for the first input “4”, the inferred labels are “1, 5, 4, 2” ranked by measured distances. The number pad layout shows that the mistakenly inferred labels “1”, “4”, and “2” are neighboring buttons around the real input “4”. If the attacker only take the top one or top two labels, the true label “4” will not be covered. As shown in the example, for the whole sequence of inputs, the coverage of true inputs with only the 1st label is 0.3125. The coverage will increase as more top ranked candidate labels are taken into consideration. In this example,

Layout of Number Pad		
1	2	3
4	5	6
7	8	9
*	0	#

Coverage rate with Top 1 ranked label		
0.2759	0.4643	0.5185
0.4138	0.1200	0.3333
0.2069	0.1250	0.2500
0.4348	0.3462	0.8750

Coverage rate with Top 1 & 2 ranked labels		
0.7931	0.75	0.7037
0.6897	0.4400	0.6061
0.4483	0.2917	0.6250
0.6087	0.4615	0.9583

Coverage rate with Top 1 & 2 & 3 ranked labels		
0.9310	0.8214	0.9259
0.8621	0.7200	0.9091
0.6897	0.5833	0.8333
0.6522	0.6154	0.9583

Coverage rate with Top 1 & 2 & 3 & 4 ranked labels		
0.9310	0.9286	0.9259
0.9655	0.8400	0.9394
0.8966	0.6250	1.0
0.8261	0.7692	1.0

Figure 9: Evaluation of number pad logging attack

the coverage of true inputs will increase to 1.0 when all the top 3 candidate labels are considered.

6.2.3 Evaluations

In the evaluation, we randomly generated 20 sequences of tap inputs with length of 16. Before the testing, the user is asked to play the *HostApp* for about 60 rounds to create the training dataset. We measure the coverage on each button in the number pad layout. The experiments results with different number of top ranked labels are listed in Figure 9.

From the experimental results, several observations attract our attentions. First of all, from the observed coverage, it is easier to distinguish the buttons close to the edge (such as button "#") than the inner buttons (such as button "5" and "8"). Secondly, outputting more top ranked candidate labels can greatly increase the coverage of inference. For example, the average coverage with only the top 1 label is about 0.364, and the average coverage with the top 1&2&3&4 labels is about 0.887. Moreover, taking the top four labels would achieve high coverage rates on all buttons. This is reasonable because the mistakenly inferred labels are mostly neighboring labels of the truly tapped button. Based on these observations, the attacker may achieve a high coverage of the true inputs by slightly increasing the search space.

6.3 Password Stealing Attack

6.3.1 Attack Overview

Passwords (i.e., PINs) based screen lock is the most common way to protect the smartphone from being accessed by unauthorized users. Briefly, when the screen of smartphone is turned on, the user will be asked for entering a sequence of passwords (i.e., PIN numbers) to unlock the screen. Such a sequence of passwords usually consists of a sequence of number from 0 to 9. The length of passwords is 4 on iOS and 4 or more on Android platform. Stealing the password of screen lock provides the attacker the access to the victim smartphone as well as some private information carried by the password itself. For example, people would like to use the same passwords in different occasions.

Before the attack, *TapLogger* first uses the *HostApp* with the layout shown in Figure 6 to collect the pattern of tap events and build the classifier for tap position classifier in the training mode.

To log the screen lock passwords, the *ContextIdentifier* starts the *SensorListener* service when the screen is turned on (i.e., upon receiving *ACTION_SCREEN_ON* intent) and stops the *SensorListener* after a certain period of time (e.g., 10 seconds). Because the user will be asked to enter the passwords whenever the screen is turned on. During this period of time, if tap events are detected, the *ContextIdentifier* will use the built-in classifier to infer the tap position and further the related button that has been pressed.

6.3.2 Tap Position Inference

To improve the accuracy of inference, we utilize one observation: suppose that the password of screen lock does not change and the user always enters passwords correctly, the user will always enter the same passwords in every round. With this observation, *TapLogger* divides the passwords into a sequence of individual inputs. Each input corresponds to the tap event at a fixed position of this sequence. *TapLogger* builds an *Inference Distribution Table* for each input in this sequence. For example, in Figure 10, the user enters the same passwords "5, 7, 6, 0" for 32 rounds. In each round, the first input is always the same. Thus, for every tap event detected as the first input, *TapLogger* will add one to its inferred label in the table, meaning that the tap event in this round is inferred as this label. In this way, the inference distribution table counts the number of times that a certain label (i.e., button) was inferred as a user input. The more frequently a label is inferred, the more likely this label is the real user input. Note that, if the sample size is small, the true input may not be top ranked in the distribution. As shown in Figure 10, the top ranked label with input # 4 is button *Del* instead of button 0 (the real input).

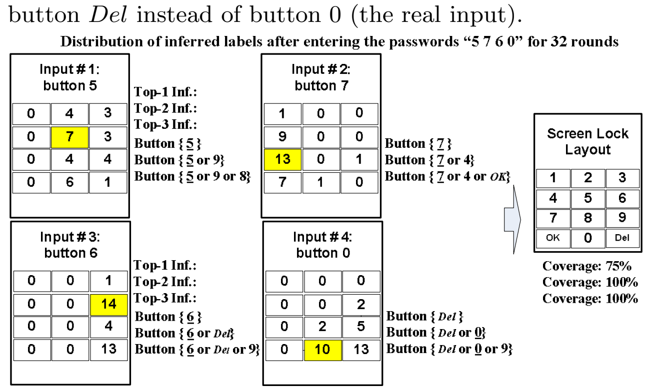


Figure 10: Example of inference distribution table

Thus, *TapLogger* may output several top ranked labels for each input instead of only the toppest one. We name them *Top-1 Inference*, *Top-2 Inference*, and *Top-3 Inference*, representing the number of top ranked label in the inference distribution table. In Figure 10, we show these three types of outputs on the right side. Correspondingly, the *coverage* of an inference is defined as the number of inputs that appear in the inferred labels. In the case of Figure 10, the *Top-1 Coverage* is $3/4 = 75\%$, while the *Top-2 Coverage* and *Top-3 Coverage* are both 100%.

Obviously, *Top-3 Inference* always generates better coverage rate than *Top-1 Inference*. However, it also means a greater search space for identifying the exact true input by the attacker. With the *Top-3 Inference* of a password of length 4, the search space for the attacker to try is $3^4 = 81$, but the search space of a *Top-2 Inference* is only $2^4 = 16$.

6.3.3 Evaluations

Experiments have been done with randomly generated passwords of length 4, 6, and 8. Five passwords are generated for each length. Before the attack, the user is asked to play the *HostApp* for about 60 rounds to build the training dataset. To collect the testing dataset, the user is asked to enter each password 30 rounds with *TapLogger* working

Table 3: The results of screen lock attack

Password Length	4	6	8
Average Top-1 Coverage	0.4	0.266	0.45
Average Top-2 Coverage	0.75	0.6002	0.925
Average Top-3 Coverage	1.0	0.8	0.975

in the logging mode. For tap position inference, we build the classifier using LibSVM [7].

The results of average coverage in our experiments is shown in Table 3. According to the experimental results, with *TapLogger*, the attacker may only need to try the top 3 labels for each input and receive a high probability of hitting the actual password. Note that, in the Table 3, the average coverage with password length 8 is even better than that of length 6. This is because some randomly generated passwords of length 6 contain inner buttons, e.g., Button 5, which causes a low coverage rate.

7. DISCUSSION

7.1 Security Permission Requirements

Note that, as motion sensors are considered as an insensitive resource, *TapLogger* does not require any security permission to access the accelerometer and orientation sensors. In fact, accessing both sensors on other platforms, such as iPhone and Blackberry, do not require security permission either. Due to the space limit, we discuss the feasibility of implementing *TapLogger* on other platforms in Appendix B.

Moreover, as the *ContextIdentifier* service keeps running in the background, the reader may wonder if a curious user may find it suspicious when reviewing the list of running services in *Service Manager*. First of all, we admit that the *ContextIdentifier* service will appear in the list. However, as the smartphone platform and apps are becoming more and more complex, there are usually tens of services, e.g., system's or apps', running in the background. According to our experience on PC, the attacker can easily fool the user by giving the *ContextIdentifier* service a benign name to avoid noticing.

7.2 Overhead Analysis

Computational Overhead: the computational overhead of *TapLogger* on smartphone includes detecting tap events with a $MonitorWin$ of sensor readings and training a classifier for tap position inference. The workload of tap event detection is low because the size of $MonitorWin$ is fixed and limited (e.g.,$<$ 50 readings). Suppose a sensor reading is represented by 4 Bytes. Only 200 Bytes are required in the memory. Also, our tap event detection algorithm relies on heuristic rules and is very lightweight.

For the classifier, both LibSVM and K-means approaches are applied in our attacks. As only six features are applied in the inference, the computational overhead is small. During the experiments, training a classifier with LibSVM with about 800 tap events takes seconds to complete. Thus, the computational overhead is not a problem for *TapLogger*.

Another concern is about the battery consumption caused by continuous sensing with the accelerometer and orientation sensors. According to our measurements on Nexus, the battery can only hold for less than 4 hours if we keep sensing with accelerometer sensors as the highest sample rate. To avoid draining the battery, our *ContextIdentifier* identifies the current context of smartphone and starts the sensing when the touchscreen is on. Therefore, when the touchscreen is off, the sensing will be stopped and *TapLogger* incurs no overhead at all. In this way, we avoid draining the battery by continuous sensing.

Communication Overhead Analysis: *TapLogger* sends inferred sensitive user inputs to a remote attack server with labels of zones of a target view. Thus, the generated traffic is very little. To avoid the user from noticing data uploading, *TapLogger* stealthily uploads in two ways. One way is to send the data when the user is interacting with the *HostApp*. For example, *TapLogger* may upload the collected tap events when the user is uploading his local scores to online game center. The other way is to transmits data in background after the touchscreen is turned off. This approach is more timely and the amount of each data transmission will be smaller. It is suggested uploading tap events collected in the logging mode using this approach.

7.3 Countermeasures

The fundamental problem here is that sensing is unmanaged on existing smartphone platforms. People are still unaware of potential risks of unmanaged sensors on smartphones. To prevent such types of attacks, we see an urgent need for sensing management systems on the existing commodity smartphone platforms. For example, we could modify the privacy mode introduced in [43] to place security restrictions on data access to onboard sensors. Sensors, such as accelerometer and orientation sensors, should all be considered as sensitive to user's privacy and need gaining security permissions to access.

Further, even with permission restrictions on the on-board sensors, the attacker may still be able to gain access to sensor readings indirectly, e.g., through the confused deputy attack [17] or the permission re-delegation attacke [15]. In this case, the defense approaches recently proposed in [15] and [11] could be applied on smartphones.

Third, from the perspective of a user, several approaches can all increase the difficulties of attacks launched by *TapLogger*, such as changing the password frequently, choosing password with numbers difficult to infer, and increasing the length of PIN numbers.

8. RELATED WORK

8.1 Logging Attacks on Smartphones

Several logging attacks have been proposed to get user inputs on smartphones. Compromising the operating system [41] or hijacking the touch event with fake user interface [24] are straightforward, but they are complex and easy to be detected. Luring the user to install malicious input method applications is another approach. However, the functionality of input method applications is restricted and the user will be warned before installing such applications [12]. [3] studies the feasibility of identifying the password pattern of screen lock by examining the smudges left on touchscreen after usage. Besides, [27] and [30] propose shoulder surfing attacks which infer user inputs by observing a user's actions on the touchscreen with a camera.

In this work, we utilize the (relatively) insensitive motion sensor readings to infer the sensitive coordinate information of tap events. As shown in Section 6, no specific security permission is required to launch the attack (except the Networking permission required to send the user inputs back to attack server). Moreover, *TapLogger* is automatic in both the training and logging phases. The attacker does not need to be close to the victim user (as in the shoulder surfing attacks) because *TapLogger* will stealthily perform the logging attack and transfer the inferred inputs back to the remote attack server.

8.2 Attacks Relying on Mobile Sensors

The privacy concerns on sensitive mobile sensor data have been raised for some while [22, 8, 6]. Besides the location tracking attacks [16, 18, 21], recent attacks show that the video camera might be controlled by a malware to stealthily record the surrounding when a user enters a building[42] that requires security clearance, credit card and pin numbers can be stolen by a trojan malware which controls the microphone in a smartphone when a user speaks his credit card number to phone menu systems [35]. All these attacks rely on well-known *sensitive sensors*, such as GPS, microphone, and camera. Accessing these sensitive sensors requires security permissions granted by users. Different to existing works, our attack is based on motion sensors that are usually considered insensitive and can be accessed by the background services with no security permissions. Thus, the proposed attack is stealthier and more difficult to detect.

With motion sensors, [20] presented a proof-of-concept of location inference attack that infers the location changes of a vehicle on a city map basing on the accelerometer sensor measurements collected from the driver's smartphone. [28] introduced a spying application, named *(sp)iPhone*, which utilizes the sensed accelerometer readings on a smartphone to infer the user inputs on a nearby keyboard. The work most similar to *TapLogger* are [5] and [13]. [5] observed the correlation between tap event positions and motion changes of smartphone, and validated the observation via a data collection application. [13] divided the touchscreen into zones and studied the feasibility of inferring tapped zones basing on readings collected from accelerometer. However, the differences between *TapLogger* and [5, 13] are significant. First of all, *TapLogger* proposes a new approach for tap event detection which is not discussed in [5, 13]. Secondly, compared to that in [5], *TapLogger* applies different features extracted from orientation sensor readings in the tap position inference. [13] extracted features from accelerometer sensor readings only. Thirdly, we present the complete design and implementation of a trojan which includes a stealthily training phase as well as two practical attacks. Last but not the least, we further showed how the user and device factors impact on the accuracy of logging attacks.

9. CONCLUSION

While the applications relying on mobile sensing are booming, the security and privacy issues related to such applications are not well understood yet. In this paper, we study the feasibility of inferring user inputs on smartphone touchscreen by monitoring readings collected from on-board motion sensors. Specifically, we first present a tap event detection scheme to discover and utilize the user's pattern with statistical measurements on acceleration, and then present an approach of inferring tap position with observed gesture changes. Further, we propose the detailed design of *TapLogger*, a trojan application implementing the proposed two approaches. We show two feasible attacks based on *TapLogger* and use experimental results to show the feasibility of proposed attacks.

9.1 Acknowledgments

We thank the reviewers for the valuable comments. This work was supported in part by NSF CAREER 0643906. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. Government.

10. REFERENCES

[1] Apple: shuffle songs on iphone, `http://www.apple.com/iphone/features/ipod.html`

[2] Electronic Arts: Need for speed shift on iphone, `http://itunes.apple.com/us/app/need-for-speed-shift/id337641298?mt=8`

[3] Aviv, A.J., Gibson, K., Mossop, E., Blaze, M., Smith, J.M.: Smudge attacks on smartphone touch screens. In: Proceedings of the 4th USENIX conference on Offensive technologies. pp. 1–7. WOOT'10 (2010)

[4] BlackBerry: Ui and navigation - development guide - blackberry java sdk - 7.0 beta

[5] Cai, L., Chen, H.: Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In: Proc. of HotSec'11 (2011)

[6] Cai, L., Machiraju, S., Chen, H.: Defending against sensor-sniffing attacks on mobile phones. In: The First ACM SIGCOMM Workshop on Networking, Systems, Applications on Mobile Handhelds (MobiHeld) (2009)

[7] Chang, C.C., Lin, C.J.: Libsvm: A library for support vector machines. ACM Trans. Intell. Syst. Technol. 2, 27:1–27:27 (May 2011)

[8] Das, T., Mohan, P., Padmanabhan, V.N., Ramjee, R., Sharma, A.: PRISM: platform for remote sensing using smartphones. In: Proceedings of the international conf. on Mobile systems, applications, and services (2010)

[9] Android Developers: SensorEvent specification, `http://developer.android.com/reference/android/hardware/SensorEvent.html`

[10] Developers, A.: Handling UI events, `http://developer.android.com/guide/topics/ui/ui-events.html`

[11] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight provenance for smart phone operating systems. In: Proc. of Usenix Security'11

[12] Android Dveloper: InputMethodManager, `http://developer.android.com/reference/android/view/inputmethod/InputMethodManager.html`

[13] Emmanuel Owusu, Jun Han, S.D.A.P.J.Z.: ACCessory: Keystroke Inference using Accelerometers on Smartphones. In: Procceedings of Workshop on Mobile Computing Systems and Applications (HotMobile) (2012)

[14] Facebook: Facebook on android, `https://market.android.com/details?id=com.facebook.katana&hl=en`

[15] Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: Proc. of Usenix Security'11

[16] Golle, P., Partridge, K.: On the anonymity of home/work location pairs. In: Proceedings of the 7th International Conference on Pervasive Computing. pp. 390–397. Pervasive '09, Springer-Verlag, Berlin, Heidelberg (2009)

[17] Hardy, N.: The confused deputy: (or why capabilities might have been invented). SIGOPS Oper. Syst. Rev. 22, 36–38 (October 1988)

[18] Hoh, B., Gruteser, M., Xiong, H., Alrabady, A.: Enhancing security and privacy in traffic-monitoring systems. IEEE Pervasive Computing 5, 38–46 (October 2006)

[19] FitnessKeeper Inc.: RunKeeper application, `runkeeper.com`

[20] Jun Han, Emmanuel Owusu, T.L.N.A.P.J.Z.: ACComplice: Location Inference using Accelerometers on Smartphones. In: Proceedings of COMSNETS'12 (2012)

[21] Krumm, J.: Inference attacks on location tracks. In: Proceedings of the 5th international conference on Pervasive computing. pp. 127–143. PERVASIVE'07, Springer-Verlag, Berlin, Heidelberg (2007)

[22] Lane, N.D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., Campbell, A.T.: A survey of mobile phone sensing. Comm. Mag. 48, 140–150 (September 2010)

[23] Apple iOS Developer Library: Uiaccelerometer class reference, http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIAccelerometer_Class/Reference/UIAccelerometer.html

[24] Lineberry, A.: Android touch-event hijacking (2010), http://blog.mylookout.com/2010/12/android-touch-event-hijacking/

[25] LLC, A.: Kindle on android, https://market.android.com/details?id=com.amazon.kindle&hl=en

[26] Ltd., R.M.: Angry birds on android, https://market.android.com/details?id=com.rovio.angrybirds

[27] Maggi, F., Volpatto, A., Gasparini, S., Boracchi, G., Zanero, S.: Poster: Fast, automatic iphone shoulder surfing. In: Proc. of the 18th Conference on Computer and Communication Security (CCS) (2011)

[28] Marquardt, P., Verma, A., Carter, H., Traynor, P.: (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. CCS '11, ACM (2011)

[29] Meier, R.: Professional Android 2 Application Development. Wiley Publishing, Inc. (2009)

[30] Raguram, R., White, A.M., Goswami, D., Monrose, F., Frahm, J.M.: ispy: automatic reconstruction of typed input from compromising reflections. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 527–536. CCS '11, ACM (2011)

[31] Ravindranath, L., Newport, C., Balakrishnan, H., Madden, S.: Improving wireless network performance using sensor hints. In: Proceedings of USENIX conference on Networked systems design and implementation (2011)

[32] Reddy, S., Mun, M., Burke, J., Estrin, D., Hansen, M., Srivastava, M.: Using mobile phones to determine transportation modes. ACM Trans. Sen. Netw. 6, 13:1–13:27 (March 2010)

[33] Ross, S.M.: Introduction to Probability and Statistics for Engineers and Scientiests. Academic Press, 2nd edn. (1999)

[34] Saffer, D.: Designing Gestural Interfaces. O'Reilly (2008)

[35] Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundminer: A stealthy and context-aware sound trojan for smartphones. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS) (2011)

[36] Siewiorek, D., Smailagic, A., Furukawa, J., Krause, A., Moraveji, N., Reiger, K., Shaffer, J., Wong, F.L.: Sensay: A context-aware mobile phone. In: Proceedings of the 7th IEEE International Symposium on Wearable Computers. pp. 248–. ISWC '03, IEEE Computer Society (2003)

[37] Takeuchi, S., Tamura, S., Hayamizu, S.: Human action recognition using acceleration information based on hidden markov model. In: Proc of 2009 APSIAPA Annual Summit and Conference (2009)

[38] Thiagarajan, A., Biagioni, J., Gerlich, T., Eriksson, J.: Cooperative transit tracking using smart-phones. In: Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems. pp. 85–98. SenSys '10 (2010)

[39] USA Today: Hello, big brother: Digital sensors are watching us, http://www.usatoday.com/tech/news/2011-01-26-digitalsensors26_CV_N.htm

[40] wikipedia: Comparison of smartphones, http://en.wikipedia.org/wiki/Comparison_of_smartphones

[41] Wikipedia: ios jailbreaking, http://en.wikipedia.org/wiki/IOS_jailbreaking

[42] Xu, N., Zhang, F., Luo, Y., Jia, W., Xuan, D., Teng, J.: Stealthy video capturer: a new video-based spyware in 3g smartphones. In: Proceedings of the second ACM conference on Wireless network security (2009)

[43] Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphoneapplications (on android). In: Proc. of TRUST'11

APPENDIX

A. TAP EVENT DETECTION ALGORITHM

The detailed checking scheme for tap event detection is described in the following Algorithm 1.

input : $P_S = \{I_1, I_2, I_3, I_4, I_5\}$, the learnt pattern of tap events, where $I_i = \{L_i, U_i\}$;
MW_S, the sequence of $SqSum$ readings in the current $MonitorWin$;

output: $Indicator$, that returns $TRUE$ if a tap event is detected;
$StartIndex$ and $EndIndex$, that returns the estimated start and end of the detected tap event in the $MonitorWin$, respectively.

$base \leftarrow G^2$;
if $MW_S(checkpoint) - base \notin I_1$ **then**
    $Indicator \leftarrow FALSE$; Return $Indicator$;
end
$StartIndex \leftarrow$ the first reading that is close enough to $base$ starting from the $checkpoint$ to 0
$trough \leftarrow$ the index of minimum reading between $StartIndex$ and $checkpoint$ in the $MonitorWin$;
if $MW_S(trough) - base \notin I_2$ **then**
    $Indicator \leftarrow FALSE$; Return $Indicator$;
end
if $MW_S(checkpoint) - MW_S(trough) \notin I_3$ **then**
    $Indicator \leftarrow FALSE$; Return $Indicator$;
end
if $checkpoint - trough \notin I_4$ **then**
    $Indicator \leftarrow FALSE$; Return $Indicator$;
end
$EndIndex \leftarrow StartIndex + AveLength$
if $std(NW_S[StartIndex, EndIndex]) \notin I_5$ **then**
    $Indicator \leftarrow FALSE$; Return $Indicator$;
end
Return $Indicator$, $StartIndex$, $EndIndex$;

Algorithm 1: Tap event detection in logging mode

B. ATTACK ON OTHER PLATFORMS

Other smartphones, such as iPhone and BlackBerry, have similar on-board sensors equipped on devices. There are already thousands of iPhone and iPad apps that leverage the accelerometer sensor in gaming, healthcare areas, etc. Similar to the accelerometer usage defined on Android platform, iOS also provides three axes readings: X axis reading for moving left or right, Y axis reading for moving forward or backward, and Z axis reading for moving up or down. We can obtain them by implementing a class specifying the UIAccelerometerDelegate protocol to listen to the accelerometer events. Although iOS 4 does not support true multitasking (except a few services, e.g., streaming etc.,), which means our taplogger cannot reside in background running on iOS devices to keep tracking tap events, it is feasible doing on jailbreaked iOS devices. We do not discuss BlackBerry in this article because touchscreen based BlackBerry devices are rare in the sense of both in BlackBerry family and in smartphone market share. However, our approach will be still valid with careful investigation of the working mechanism of the on-board sensors on BlackBerry. We will address it in our future work.