Slogger: Smashing Motion-based Touchstroke Logging with Transparent System Noise

Prakash Shrestha
University of Alabama at Birmingham
Birmingham, Alabama
prakashs@uab.edu

Manar Mohamed
University of Alabama at Birmingham
Birmingham, Alabama
manar@uab.edu

Nitesh Saxena
University of Alabama at Birmingham
Birmingham, Alabama
saxena@cis.uab.edu

ABSTRACT

Recent research shows that it is possible to infer a user's touchscreen inputs (e.g., passwords) on Android devices based on inertial (motion/position) sensors, currently freely-accessible by any Android app. Given the high accuracies of such touchstroke logging attacks, they are now considered a significant threat to user privacy. Consequently, the security community has started exploring defenses to such side channel attacks, but the suggested solutions are either not effective (e.g., those based on vibrational noise) and/or may significantly undermine system usability (e.g., those based on keyboard layout randomization).

In this paper, we introduce a novel and practical defense to motion-based touchstroke leakage based on *system-generated*, *fully automated* and *user-oblivious* sensory noise. Our defense leverages a recently developed framework, SMASheD, that takes advantage of the Android's ADB functionality and can programmatically inject noise to various inertial sensors. Although SMASheD was originally advertised as a malicious app by its authors, we use it to build a defense mechanism, called *Slogger* ("Smashing the logger"), for defeating sensor-based touchstroke logging attacks. Slogger transparently inserts noisy sensor readings in the background as the user provides sensitive touchscreen input (e.g., password, PIN or credit card info) in order to obfuscate the original sensor readings. It can be installed in the user space without the need to root the device and to change the device's OS or kernel.

Our contributions are three-fold. *First*, we introduce Slogger, identifying a novel, benign use case of SMASheD that can defeat touchstroke logging attacks. *Second*, we design and implement the Slogger app system that can be used to protect sensitive touchscreen input from leaking away. *Third*, we comprehensively evaluate Slogger against state-of-the-art touchstroke detection and inference attacks. Our results show that Slogger can significantly reduce the level of touchstroke leakage to the extent these attacks may become unworkable in practice, without affecting other benign apps. We also show that the leakage can be minimized even when attacks utilize a *fusion* of multiple motion-position sensors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WiSec'16 , July 18-22, 2016, Darmstadt, Germany

© 2016 ACM. ISBN 978-1-4503-4270-4/16/07. . . $15.00

DOI: http://dx.doi.org/10.1145/2939918.2939924

CCS Concepts

•Security and privacy → Malware and its mitigation; Side-channel analysis and countermeasures; Mobile platform security;

Keywords

Side-channel attacks, mobile security, Android

1. INTRODUCTION

Sensors are becoming an inevitable part of mobile and wireless computing. With mobile devices coming readily equipped with multiple, low-cost sensors and mobile OS platforms adding full software support for developing applications using these sensors, there is an enormous growth in the adoption of mobile devices.

Different varieties of sensors are available on the current generation of mobile devices, such as smartphones and tablets, including: user input sensor (touchscreen and hardware buttons), audio-visual sensors (microphone and camera), and inertial or motion-position sensors (e.g., accelerometer, gyroscope and magnetometer). The mobile apps that are based on these sensors have seen a widespread deployment in many domains ranging from entertainment, navigation and transportation (e.g., [12]) to elderly care (e.g., [4, 7]) and safety (e.g., [24]). In addition, mobile device sensors are used to build a wide range of security/privacy applications, including those geared for authentication and authorization (e.g., [9, 10, 19]).

Since mobile sensors provide potentially sensitive information about the host device, the device's user or the device's surroundings, protecting sensor data from abuse by malicious applications becomes paramount. Consequently, most mobile platforms have established a sensor security access control model. Specifically, Android, one of the most popular mobile OSs and the subject of this paper, follows a model where *read access* to many sensitive sensors is very restrictive (e.g., an app can only read its *own* touchscreen input data) or requires special install-time permissions granted by the user (e.g., to access microphone or camera).

However, the read access to most other sensors, including inertial sensors, is not restricted within this model because Android may not consider these sensors as explicitly sensitive. This openness in the Android sensor security architecture to the inertial sensors has given rise to a potentially significant threat of *motion-based side channel attacks*. Especially, an interesting line of recent research [17, 23] has shown that it is possible to infer a user's touchscreen inputs on Android devices (deemed sensitive, and protected by Android's security model), such as passwords, based on these "globally accessible" inertial sensor measurements. The primary intuition behind these attacks is that the movement and positional changes introduced by hitting a key on the device's touchscreen are correlated with the key itself and can thus be exploited to make an

inference of the key. These attacks generally follow a two-step process. The first step is *touchstroke detection*, that is, finding the start and end of the tap event based on the motion sensor readings. The second step is *touchstroke inference*, that is, inferring the pressed key based on the motion sensors readings during the touch event detected in the first step. (These attacks are reviewed in Section 2.)

Given the high accuracies of such touchstroke logging attacks [17, 23], they may now be considered a significant threat to user privacy. For example, as shown in prior research, user's PINs or passwords can be extracted with high success rates. Naturally, the security community has responded by studying defenses to such side channel attacks. However, the suggested solutions are either not effective, such as the one based on vibrational noise created by the mobile phone's vibration motor [17], and/or may significantly compromise the usability of the system, such as the one based on keyboard layout randomization [20, 22]. We analyze the limitations of prior defense mechanisms in Section 3.

In this paper, we introduce a novel and practical defense to motion-based touchstroke leakage based on *system-generated*, *fully automated* and *user-oblivious* sensory noise. Our defense leverages a recently developed framework, SMASheD ("Sniffing and Manipulating Android Sensor Data") [16], that takes advantage of the Android's ADB (Android Debugging Bridge) functionality and can programmatically sniff and inject noise to various inertial sensors. Although SMASheD was originally advertised as a malicious app by its authors [16], we use it to build a defense mechanism, called *Slogger* ("Smashing the logger"), for defeating sensor-based keystroke logging attacks. Slogger transparently inserts noisy sensor readings in the background as the user provides sensitive touchscreen input (e.g., password, PIN or credit card info) in order to obfuscate the original sensor readings. It does so without impacting other benign apps that rely upon original sensor readings.

Our Contributions: We believe that our work brings forth the following contributions to the field of mobile and wireless security:

1. *Defensive Use of a Malicious Tool*: We identify a novel, benign use case of the SMASheD framework that can be used to defeat touchstroke logging attacks. We believe that turning an existing malicious approach into a defensive tool could be valuable to the advancement of science.

2. *Design and Implementation of Slogger*: We provide the design and implementation of the Slogger app system that can be used to protect sensitive touchscreen input from leaking away using the motion-based side channel attacks. Slogger works in the background and is completely transparent to the user and other benign apps. It can be installed in the user space like any other ADB app, without the need to root the device and to change the device's OS or kernel. Our design and implementation details are presented in Section 4.

3. *Recreation of Prior Attacks and Evaluation of Slogger*: We comprehensively evaluate Slogger against state-of-the-art touchstroke detection and inference attacks. To achieve this goal, we first present a re-implementation and evaluation of the prior attacks in independent settings. Our results show that Slogger can significantly reduce the level of touchstroke leakage to the extent these attacks may become completely unworkable in practice. We also demonstrate that Slogger may not cause any negative effects to other benign apps that rely upon sensor measurements while the user provides sensitive input (e.g., the pedometer and fall detection apps). Our re-validation of prior attacks is described in Sections 2.2 and 2.3. Our evaluation of Slogger is presented in Section 5.

4. *Evaluation against Fusion of Sensors*: We show that the touchstroke leakage based on motion sensors can be greatly increased by utilizing a *fusion* of multiple motion sensors. That is, we propose new attacks that achieve much higher accuracies than the existing attacks employing only single sensor (accelerometer alone). Importantly, we further show that Slogger can still minimize this leakage thereby defeating even these powerful attacks. The details of this part of the work are presented in Section 5.3.

2. BACKGROUND

Several research works have been proposed that utilize motion-based side-channel attack to log the user's touchscreen input (e.g., passwords) [17, 23]. The general approach of these motion-based touchstroke logging attacks consists of two steps. The first step is *touchstroke detection*, that is, finding the start and end of the tap event based on the motion sensor readings. The second step is *touchstroke inference*, that is, inferring the pressed key based on the motion sensors readings during the touch event detected in the first step. Most of the recent work focuses on the second step, in which, the attacker extracts the features from the sensor readings and employs one of the standard machine learning techniques to infer the pressed key. Intuitively, if we can block one or both of these two steps, it is possible to defend against such attacks. To show that our proposed defense mechanism – *Slogger* – works well against both touchstroke detection as well as inference attacks, we implemented one of the state-of-the-art touchstroke detection algorithms, as implemented in the TapLogger [23], and one of the state-of-the-art touchstroke inference algorithm, called ACCessory [17].

In this section, we first present the general threat model that most of the motion-based side-channel touchstroke logging attacks have considered. Then, we provide a brief review of TapLogger and ACCessory , and independently re-create and re-validate both attacks. Next, we present SMASheD [16], a framework that has been proposed to inject sensors events for malicious purposes, that we used to build our *Slogger* defense system. Throughout our work, we use Samsung S4 with sampling frequency of 100Hz.

2.1 Threat Model

As mentioned earlier, motion-based side-channel touchstroke logging attacks consist of two steps: *touchstroke detection* and *touchstroke inference*. Each of these steps first needs to learn the patterns (basically extract the features) of the touchstrokes based on the motion sensor measurements. Later, they utilize the learned touchstrokes motion pattern to detect and to infer the touchstrokes. Both steps in motion-based touchstrokes logging attacks have two phases:

1. **Training Phase:** In this phase, the touchstroke logging model acquires the touchstroke information such as the timestamps of the touch pressed and released events, the coordinates of the touchstroke on the screen and the motion sensor measurements during the touchstroke event to learn the motion pattern corresponding to a touchstroke. This model assumes that the adversary fools the user to install and use a malicious application that stealthily collects this information. The malicious application can be, for example, a gaming application, such as *HostApp* used by TapLogger, that requires the user to tap on various positions of the screen. The adversary then utilizes the collected information to extract the features and learn the pattern of the touchstrokes.

2. **Testing Phase:** In this phase of the attack, the malicious application runs in the background, and records sensor measurements stealthily whenever user starts entering the sensitive input. Using the learned knowledge in the training phase, the touchstroke logging model attempts to detect and infer the touchstrokes.

The model presented above is in line with the general threat model that motion-based side-channel touchstroke logging attacks have assumed in their work, such as TapLogger [23], ACCessory [17], TextLogger [18], and TouchLogger [3].

Slogger aims to defend such attacks by injecting noise into the sensors files. Slogger threat model also consider attackers with more capabilities, such as those who try deliberately to remove the injected noise, or trying to infer the keystrokes over multiple rounds of sniffing the sensors data.

2.2 TapLogger: Review and Re-Validation

Xu et al. [23] developed "TapLogger" to infer a user's tap inputs to a smartphone by utilizing the accelerometer and orientation sensors. First, TapLogger learns the motion change patterns of tap events. Later, TapLogger uses the learned pattern to infer the occurrence of tap events and the tapped positions on the touchscreen. TapLogger shows that tap events have a unique pattern in terms of the changes in the accelerometer readings, which can be utilized in detecting the occurrence of taps. This information along with the orientation sensor readings and the screen layout can be utilized to infer the user input.

We re-implemented the tap event detection algorithm, *Tap-detector*, as described in [23]. Tap-detector calculates the square sum of accelerometer readings, $SqSum = x^2 + y^2 + z^2$, which represents the force induced on the smartphone while typing. During the training phase, as the user is tapping on the attacker's trojan app, the start and end of the tapping event can be identified by the timestamps in which Motion.Event.ACTION_DOWN and Motion.Event.ACTION_UP events are received, respectively. Tap-detector first extracts the $SqSum$ corresponding to the tap events. Then, it extracts several features to describe the tap event: the peak and trough of the readings minus base, difference and time gap between the peak and the trough, and the standard deviation of the entire tap event. After the user performs multiple taps, Tap-detector learns the range between the lower and upper extremes of each of the features and utilizes these ranges to detect tapping events later.

In our study, we do not use TapLogger 's tap inference algorithm. This is because TapLogger employs the orientation sensor measurements for touchstroke inference attacks. Orientation sensor is a software-based sensor that derives its data from accelerometer and geomagnetic field sensor which is deprecated starting from Android 2.2 (API Level 8) [1]. Even if Orientation sensor were to be used, Slogger can still defeat it since it can insert noise to both accelerometer and geomagnetic field sensor from where this sensor derives its data.

Validation: To validate our implementation, we trained our implementation of Tap-detector with 1200 taps, and tested it against another 100 taps. We tested our implementation with 5 sets of these 100 taps. These taps were collected in a similar setting as documented in the TapLogger paper [23], i.e., the phone is held by one hand and the key on touchscreen is pressed by index finger of the other hand. Tap-detector was able to get the precision of 88.31%, recall of 84.02% and F-measure of 85.97%, which is very close to the one reported in the TapLogger paper. Given these validation results, we later (in Section 5.1) evaluate our Slogger defense system against Tap-detector.

2.3 ACCessory: Review and Re-Validation

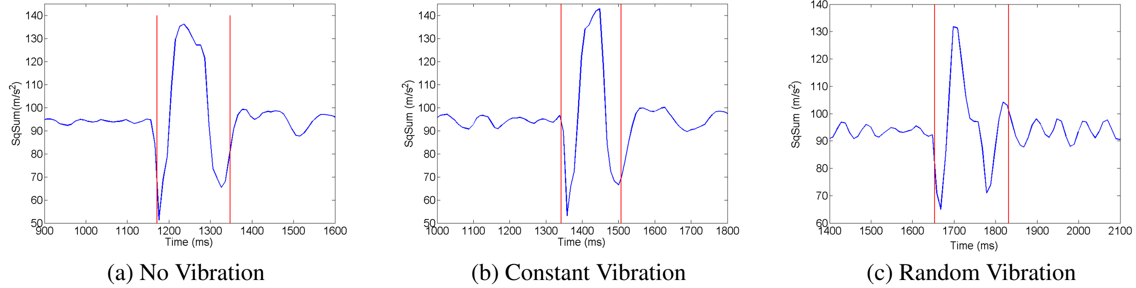
Owusu et al. [17] provided the design and implementation of an Android application, ACCessory, which demonstrates that accelerometer can be used as a side channel attack to infer short sequence of touchstroke on a smartphone soft keyboard, and machine learning techniques can be employed to infer input like password.

ACCessory has two collection modes: area mode and character mode. In the area mode, the screen was divided into regions, and the task was to infer the tapped regions at different granularities level (i.e., 2, 4, or 8). In particular, the goal of the area mode was to evaluate the inference accuracy at varying levels of granularity. The authors determined that splitting the screen into eight regions and classifying individual region keys separately yields the best average key accuracy of approximately 24.5%. The purpose of the character mode of collection was to extend the attack to inferring a sequence of entered text, in contrast to a per-key inference, in order to reconstruct typed passwords.

We re-created the area mode collection of ACCessory to later evaluate the performance of our proposed defense. We designed an Android application for area mode collection similar to the one described in [17] that consists of 10x6 array of buttons that completely cover the entire screen of the smartphone. As the application runs, it starts recording every new updated accelerometer reading. Each record contains accelerometer measurement and timestamp for the accelerometer measurement. The same application also monitors the key-pressed and key-released touch events as they are dispatched by each button and the coordinates of the button pressed on the touchscreen to establish ground truth for the analysis. As in the ACCessory design, we first use the linear interpolation technique to obtain consistent sampling interval throughout the dataset. The average sampling rate of our device's accelerometer is 100 *Hz*. We then compute $SqSum$ and extract the features that describe a tap event in a particular region of the screen. The features used to describe a tap event in a particular region of the screen, as described in ACCessory and used in our implementation, are shown in Table 1. Random Forest algorithm is then used to obtain the per key inference accuracy corresponding to each of the regions granularity.

We note that it was important for us to evaluate our Slogger system against touchstroke inference, even if Slogger could perfectly defeat touchstroke detection. This is because the attacker may employ various strategies other than utilizing the sensor measurements to perform tap detection which may be more accurate. For instance, attacker may surreptitiously monitor the process's shared memory size while the device's keyboard app is being used, and uses it to infer the tap events [18]. It is also possible that the attacker is recording a video of user typing on touch enabled devices, which may be utilized to detect the touchstroke events [21]. To this end, to evaluate our defense against touchstroke inference attacks, given touchstroke detection has been launched successfully, we implemented the area mode collection of ACCessory and tested with our proposed defense mechanism, Slogger.

Validation: To validate our implementation, we consider the setting which is similar to the one used in the ACCessory paper, i.e., the device is held by both hands in the landscape orientation, and thumbs are used to enter the text. Similar to ACCessory, with this setting, we collected data corresponds to about 1200 key presses, where each key receives about 20 presses. Using stratified 10-fold cross-validation, our implementation was able to achieve 90.02% of accuracy for two region splits (i.e., for two halves of the screen) which is in line with the result reported in the ACCessory work. When considering the higher level of granularity, our implementation was able to achieve 68.73% for 4 regions, and 10.93% for



(a) No Vibration (b) Constant Vibration (c) Random Vibration

Figure 1: Negligible effect of vibrational noise on a stream of accelerometer readings while pressing a key on phone's touchscreen.

Table 1: List of features used to describe accelerometer stream values for a tap inference. Dimensional features (D) are computed separately for each dimension (x, y, z) as well as for $SqSum$ of acceleration. Meta features (M) describe the window features of acceleration stream and are calculated only once per feature vector [17].

Feature	Description	D/M
RMS	The Root-Mean-Square Value	D
RMSE	The Root-Mean-Square Error	D
Min	The Minimum Value	D
Max	The Maximum Value	D
AvgDeltas	The average sample-by-sample change	D
NumMax	The number of local peaks	D
NumMin	The number of local crests	D
TTP	The average time from a sample to a peak	D
TTC	The average time from a sample to a crest	D
RCR	The RMS cross rate	D
SMA	The Signal Magnitude Area	D
Total Time	The Total Time of the window	M
Window Size	The number of samples in the window	M

60 regions while ACCessory reported more than 80% for 4 regions and 24.5% for 60 regions. We attribute these differences to use of a different device in our experimental set-up. We used a Samsung Galaxy S4 phone, while ACCessory experiments were done with an HTC ADR6300. These two devices differ in screen sizes: S4 has a larger screen (136.6 mm x 69.8 mm) than HTC ADR6300 (117.5 mm x 58.5 mm). The larger the screen size, the higher will be the motion generated noise while typing, which may have contributed to lower accuracies in our case.

2.4 SMASheD Framework

SMASheD [16] is a framework that can be used for sniffing and manipulating Android sensor data. SMASheD leverages ADB to install a service on an Android device with shell privileges. Specifically, the installed service would have privileges to read from and write to the Android device sensors files (i.e., the files corresponding to position, motion and environmental sensors as well as user input sensors: touch screen and hardware buttons).

SMASheD framework consists of a service, two scripts, and an Android application. The service is responsible to read from and write to the sensors files. The two scripts are used to push the service from a PC to an Android device and run it. This way the service will be granted all the shell privileges. The Android application is an application that monitors the device status. For example, it checks which applications are installed on the device and which applications are running in the foreground, and according to its desired purpose, it sends read or write requests to the SMASheD service.

In [16], Mohamed et al. provided various functionalities that SMASheD can achieve given its capabilities in attacking various sensing-based authentication and authorization applications. That is, the original use case of the SMASheD framework was malicious in nature. In our paper, we utilized the SMASheD framework for a novel benign use case, specifically defending against touchstroke logging attacks. We propose Slogger – a defense based on SMASheD's sensor event injection functionality in an attempt to mitigate well-researched attacks that utilize motion sensors for touchstroke detection and inference. Slogger injects negligible sensor events such that the benign apps are not affected and injects sensor events only when the user is entering sensitive information like a password.

3. LIMITATIONS OF KNOWN DEFENSES

In this section, we discuss and analyze various possible defenses against touchstroke detection and keystroke inference attacks that have been suggested in prior research, and argue for their ineffectiveness.

Vibrational Noise: One of the possible strategies to mitigate the sensor-based touchstroke detection and inference attack, as pointed in [17], is to automatically initiate the phone vibration while the input is being provided to the phone. However, authors did not perform detailed evaluation of this intuitive strategy. To this end, we set out to evaluate the strategy of creating vibrational noise to defend against sensor-based side-channel attack. We considered two different types of vibrational noise: *(1) Constant Vibration*, and *(2) Random Vibration*. In the Constant Vibration mode, the phone is programmed to vibrate constantly over the time with same vibrational intensity as the user provides the input to the phone, whereas in the Random Vibration mode, the phone vibrates with a random pattern, i.e., the phone vibrates for some random duration, pauses for a random duration and then vibrates again, repeating this process while the user provides input to the device.

We evaluated whether the creation of such vibrational noise has significant effects on the motion sensors that may mitigate the keystroke information leakage. We recorded the accelerometer measurements when a key is being pressed, while holding the phone with one hand and pressing with index finger of another hand, in presence of both types of vibrational noise. Figure 1 represents a $SqSum$ plot of accelerometer measurements in presence and absence of the two types of vibrational noise. Specifically, Figure 1(a) represents a scenario where there is touchstroke event without any vibration (i.e., in the absence of the defense). In this case, the $SqSum$ of the stream of accelerometer can be used to detect the touchstroke event and later on detected touchstroke signal can be analyzed to infer the touchstroke. Figure 1(b) and 1(c) represent the scenarios with constant vibration and random vibration, respectively. The figures clearly show that, although both types of vibrations have some effect on the stream of accelerometer measurements corresponding to the touchstroke event, they do not offer significant contribution to hide the touchstroke event from the stream of accelerometer readings. That is, touchstrokes

are still clearly distinguishable and would be subject to inference. This analysis therefore demonstrates that vibrational noises generated by the phone's vibrational motor do not have significant effect on the accelerometer measurements and is ineffective in defending against sensor-based touchstroke logging attack, contrary to what was assumed by the ACCessory authors.

Keyboard Layout Randomization: In the motion-based key inference attack, the goal of the attacker is to learn the key pressed on the screen to learn sensitive information such as PIN, password, or even email content. By statistical analysis of motion sensor measurements, the attacker can determine the position of the touchstroke on the screen. Since the layout of the keyboard or number pad on standard devices is typically public knowledge, the keyboard layout is known to the attacker. Once the attacker determines the position of the touchstroke on the screen, with the knowledge of keyboard layout, he can map touchstroke position with keyboard/number pad layout and find out the actual key pressed on the screen.

If the layout information is kept secret from the attacker, even if the attacker knows the touchstroke position on the screen, it may fail to determine the actual key pressed and thus the information leakage can be eliminated. Song et al. [22] have proposed the idea of randomizing the layout of keyboard to hide the layout information from the attacker so that attacker could not figure out which key has been pressed even if he could find the exact tap position.

Though the idea of randomizing the layout have a sound potential to defend against motion based touchstroke inference attack, it may not be practical. Randomizing the keyboard layout significantly increases the time taken by the user to enter the text, as the user would need to search for the keys in the randomized keyboard layout every time rather than using his knowledge about the keyboard layout to locate the keys. Thus, this approach would significantly compromise the usability of the system [20] and would not be a viable defense the users can deploy.

Motion Shielding: One possible strategy to defeat motion-based touchstroke logging attacks is motion shielding. As proposed in [15], the use of phone cases, such as the one made up of leather or rubber, can minimize the motion generation, which in turn can potentially minimize the information leakage. However, this approach requires the phone cases to be highly shock absorbent and thick. Thin leather/rubber cases may reduce the motion leakage by absorbing certain amount of motion, but there may still be some motion leakage that the high resolution accelerometers can still record, which can be used by an attacker to infer the actual keys. Moreover, many users may not be willing to use such specialized phone cases due to cost or convenience reasons.

Permission Restriction to On-Board Sensors: As suggested in [2, 15, 17], on-board sensors, such as accelerometer and gyroscope, should be considered sensitive to user's privacy and therefore special security permissions must be required to gain access to such sensors. This approach, however, requires users to have a good understanding of the security model, and relies upon the users to read and understand the app's permission dialog while installing the app. This approach requires cognitive effort from the user and may not work in practice as shown by many studies [8, 13].

Sensor Access Control: Another approach as suggested in [2, 15] to mitigate the sensor-based touchstroke logging attack is the modification of mobile devices' operating systems to pause the motion sensors when sensitive input operation is being performed. This approach would make it impossible for an attacker to correlate the sensor data with the keyboard taps. However, there are several applications that run in background all the time (e.g., pedometer applications), and withholding such applications from gaining access to the sensors, or requiring manual shut down before performing any sensitive operation by the user, would greatly reduce the applicability of this approach.

Reduced Sampling Rate: Varying the sensor sampling rate can reduce the accuracy of touchstroke detection as well as touchstroke inference. Higher the sampling rate, the better the tap inference performance because more sensor samples are available to capture each tap's measurements that can model the tap effectively. Increasing the sampling rate improves the inference accuracy [15, 17]. Conversely, reducing the sampling rate can reduce the effect of touchstroke inference attack [14]. However, such approach may have undesirable effect on several legitimate applications running in the background (e.g., pedometer). Furthermore, there exist some sophisticated machine learning techniques that work well even with sampling frequency as low as 20Hz [2].

4. SLOGGER DESIGN AND IMPLEMENTATION

Our defense mechanism Slogger aims to cloak the motion-based touchstroke logging attacks with the use of internal, programmatic noise which is completely transparent to the user. We follow the implementation of the SMASheD framework [16] to realize our Slogger system. We added an initialization phase, in which Slogger learns the range of the sensor values corresponding to the user's typing style. When the user installs Slogger on her device, the user is asked to type on her device in all the settings (i.e., holding a phone in hand, or keeping phone on a surface of table and typing), and the minimum and the maximum values of all the axes for all the position sensors are computed. These values are later used to set the range of the values of the injected noise. This step is performed by the user only once. In our experiment, we used Samsung S4 which has only accelerometer and gyroscope as hardware position sensors. The rest of the position sensors (i.e., gravity, linear acceleration and orientation) are calculated based on the readings of the hardware position sensors.

We implemented the Slogger application such that whenever a user launches the application that we use for our data collection, Slogger sends inject request to the Slogger server, when the user closes the application Slogger sends stop request to Slogger server. The application that we developed for our analysis purpose can be updated such that it sends the inject request whenever the keyboard is running or whenever the user is entering any sensitive data.

Slogger server locates the files corresponding to accelerometer and gyroscope in "/dev/input/" folder. Slogger server has a socket, that keeps listening for requests. When Slogger server gets a start request, it injects random values in both the accelerometer and gyroscope that are in the range between the pre-calculated maximum and minimum values. After injecting sensor events in both the accelerometer and gyroscope sensors, it waits for a random amount of time between 7 and 12 milliseconds. Slogger server keeps on injecting till it receives a stop request.

Slogger is installed in the Android device in a similar way as described in [16]. A script is used to push the Slogger server and another script to "/data/local/temp/" folder on the Android device, then run the other script which is responsible for running the Slogger server. Like SMASheD, Slogger does not require the phone to be rooted.

Evaluation Scenarios: We implemented our system in accordance with the following three scenarios later used to evaluate Slogger against touchstroke logging attacks:

- *Slogger Absent*: In this scenario, we assume that the touch en-

abled device has not implemented any defense mechanism, in particular Slogger, against touchstroke logging attack. So, both training and testing phases of the touchstroke logging attacks (as described in Section 2.1) use normal (noise-free) stream of sensor measurements. We consider this scenario as the baseline scenario to evaluate the impact of our defense mechanism against the touchstroke logging attack.

- *Slogger Present, Attacker Trained with Non-Noisy Data*: In this scenario, we assume that Slogger has been activated on the touch-enabled devices. We also assume that it has been implemented in a way that it works with only a subset of applications that the system or the user thinks are sensitive, and the user has marked the malicious application collecting the training data as non-sensitive (i.e., unprotected by Slogger). Here, the training phase of the touchstroke logging attack uses regular (non-noisy) stream of sensor measurements while the testing phase uses the noisy stream of sensor measurements containing the Slogger injected noise.

- *Slogger Present, Attacker Trained with Noisy Data*: Similar to our second scenario, we assume that Slogger has been implemented on the touch-enabled devices in a way that it works with only a subset of applications that system or user thinks are sensitive. We also assume that the user has marked the malicious application collecting training data as a sensitive application. In such setting, Slogger noise injection will be activated when the user interacts with the malicious application. In this scenario, both the training and the testing phases use stream of sensor measurements with Slogger injected noise.

5. EXPERIMENTS AND EVALUATION

In this section, we present our experiment to evaluate the impact of Slogger against touchstroke detection based on the *Tap-detector* algorithm described in Section 2.2, followed by the impact of Slogger against touchstroke inference methodology explained in Section 2.3. Then, we show that even if fusion of multiple (motion/position) sensors is used for the touchstroke inference attack, Slogger serves as a viable defense. We also show that Slogger does not have a significant impact on any other common benign applications that utilizes motion sensors.

For our evaluation, we built an Android application with two view layouts. The first view layout consists of a number pad as described in TapLogger experiments that portray a standard number pad on a smartphone. Standard number pad on a smartphone usually contains 12 keys: 10 keys for (0 - 9) numbers, and the remaining two are '$*$' and '#'. The second view layout consists of 10x6 array of buttons covering the entire screen, as described in ACCessory experiments. The first layout is used to collect data set to investigate the impact of Slogger against touchstroke detection while the second layout is used to collect data set to investigate the impact of Slogger against touchstroke inference. The data set includes measurement of various motion sensors (accelerometer, linear acceleration, gyroscope, gravity, and rotation sensor) and log of touch events. Since all the sensors that are being recorded are tri-axial, each sensor record includes three values corresponding to the three axes and the timestamp of the record. Each record in the touch event log includes the timestamp of key-pressed or key-released event as they are dispatched by each button and the coordinates of the touch event that facilitates the establishment of the ground truth in the subsequent analysis.

We performed the experiments in normal scenario (i.e., Slogger Absent scenario) and the defensive scenario with Slogger in two different settings (i.e., when the attacker is trained with or without the noisy data). In all the experiments, we collected key presses data samples from one of the researchers involved in this study.

5.1 Slogger against Touchstroke Detection

In this experiment, we evaluate Slogger against one of the state-of-the-art touchstroke detection attacks, Tap-detector presented in (Section 2.2).

5.1.1 Data Collection

For evaluation purposes, we collected 1200 samples of key presses such that each key receives 100 touchstrokes using the number pad view layout of our application. These samples were used to train the Tap-detector system. Then, we collected 5 sets of 100 key presses to test Tap-detector, and the detection accuracy is computed by averaging the accuracies for these 5 sets. All the touchstroke samples were collected using the same setting as considered in the TapLogger experiments, i.e., phone is held by one hand and the keys on touchscreen are pressed by the index finger of the other hand. We conducted this experiment twice, one without activating Slogger, the normal scenario, and another with Slogger running in background, the defensive scenario.

5.1.2 Results

The result of Tap-detector in the normal and the defensive scenarios with Slogger are summarized in Table 2. In the absence of Slogger, Tap-detector successfully detects the touchstrokes with accuracy (F-measure) of 85.97%, which is in line with the result reported in [23]. Considering the second scenario, where Tap-detector is trained with regular stream of sensor readings while it is supplied with Slogger injected stream of sensor readings to detect the touchstroke event, Tap-detector is not able to detect any of the touchstrokes. The features that Tap-detector extracts from each window of the stream of accelerometer readings to predict the touchstroke events are now no longer present in the touchstroke signal because the touchstroke signal has been completely obfuscated by the noise injected by Slogger. Figure 2 clearly explains the impact of Slogger in accelerometer readings that completely prevent Tap-detector from detecting any of the touchstrokes. Even when both of the training and the testing sensor readings are collected while Slogger is activated (i.e., the attacker was trained with noisy data), the accuracy of Tap-detector dropped by a considerable amount, from 85.97% (normal scenario) down to 32.20%.

Table 2: Results of Tap-detector in three different scenarios: Slogger Absent, and two different defensive scenarios with Slogger.

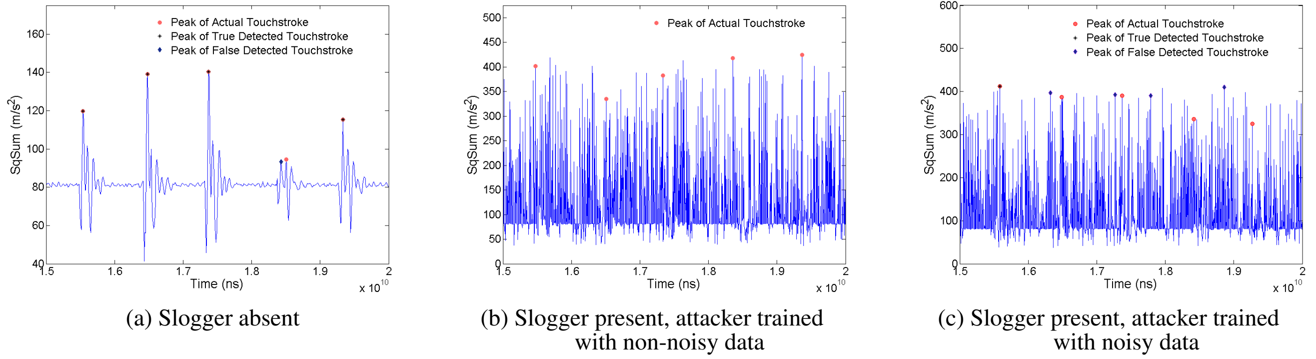
Scenarios	Precision	Recall	F-measure
Slogger Absent	88.31%	84.01%	85.97%
Slogger Present, Attacker Trained with Non-Noisy Data	0	0	N/A
Slogger Present, Attacker Trained with Noisy Data	38.07%	27.93%	32.20%

5.2 Slogger against Touchstroke Inference

In this experiment, we evaluate Slogger against one of the state-of-the-art touchstroke inference attacks, ACCessory (presented in Section 2.3).

5.2.1 Data Collection

There are several ways in which users hold their phones and several typing styles that users employ while providing input to the phone device. Addressing all possible phone holding patterns and



(a) Slogger absent

(b) Slogger present, attacker trained with non-noisy data

(c) Slogger present, attacker trained with noisy data

Figure 2: $SqSum$ plot of Accelerometer signal and results of Tap-detector in three different scenarios.

typing styles is not a feasible endeavor. Rather, in our study, we consider two realistic, commonly used phone holding settings:

- *In Hand*: In this setting, user holds the touch-enabled device using both of his hands in the landscape orientation, and provides the input to the device using the thumbs of both of these hands (similar to the setting used in [17]).
- *On Surface*: In this setting, touch-enabled device is placed on a smooth surface (e.g. a table) and the user types using the index finger of his dominant hand.

We collected 1200 samples of key presses using 10x6 buttons view layout such that each key receives 20 key presses. These samples were used to train the touchstroke inference model. Then, 5 sets of 100 key presses were collected and the inference accuracy was computed by averaging the inference accuracies over these 5 set of samples. We repeated this experiment for both the *In Hand* and *On Surface* settings in both normal and defensive scenarios.

5.2.2 Results

We evaluated Slogger against the touchstroke inference attack in all the scenarios described in Section 4. The results are summarized in Figure 3 and Figure 4.

Slogger Absent: Figure 3 shows the inference accuracy in the absence of defense system in two different settings, *In Hand* and *On Surface*. The results show that the touchstroke inference accuracy is more than 90% (compared to 50% for a random guessing attack) on average for 2 region splits (i.e., for two halves of the screen) in both the settings. For the eight-region granularity of the screen, inference accuracy drops to 45.4% in the *In Hand* setting and to 56.2% in the *On Surface* setting. Intuitively, it is obvious that increasing the granularity level of screen regions decreases the inference accuracy (but is significantly higher than random guessing accuracy of 25%). Further increasing regions granularity to 60 regions, the inference accuracy substantially drop to 17.8% in case of the *On Surface* setting and to 10.2% for the *In Hand* setting (compared to 1.67% for a random guessing attack). From Figure 3, we can find that the inference accuracy is higher in the *On Surface* setting than in the *In Hand* setting for all the granularity levels. We believe that the reason behind this is the higher amount of movement-based noise generated on the phone when it is held in hand than when it is placed on a smooth surface (table) where the phone typically remains stationary.

Slogger Present, Attacker Trained with Non-Noisy Data: Figure 3 shows that in the presence of our defensive mechanism and when the attacker is trained with non-noisy data, inference accuracy significantly drops below the random guessing accuracy. For instance, inference accuracy drops to 35.5% (random being 50%) for two halves of the screen, and to 7% (random being 12.5%) for 8 regions while for 60 regions, it drops to 0.6% (random being 1.67%).

Since the inference model is trained with regular (non-noisy) stream of touchstroke signal and is supplied with totally obscured signal of touchstroke for inference that it has never encountered (in the presence Slogger), it does not know how to predict the touchstrokes. Therefore, it gives a prediction much worse than a random prediction model. Thus, Slogger serves to provide a strong defense system against sensor-based touchstroke inference attack that enforces the inference model to behave worse than a random prediction model under all granularity level of screen areas.

Slogger Present, Attacker Trained with Noisy-Data: Figure 4 also shows the inference accuracy when the inference model is trained and tested with the stream of sensor readings that contain noise inserted by Slogger. This model seems to reduce the impact of noise in dropping the inference accuracy, as it may learn the pattern taking into account of the noises. However, the noise injected by Slogger is randomly generated as described in Section 4, the touchstroke inference model still could not learn the touchstroke features well.

In the *On Surface* setting, Slogger is able to reduce the inference accuracy to nearly random in all level of regions granularity. For instance, for two halves of the screen areas, when enabling the Slogger system, the inference accuracy drops to 56.5%, for 8 regions the inference accuracy drops to 15.8%, and it drops to 1.5% for 60 regions granularity of screen areas.

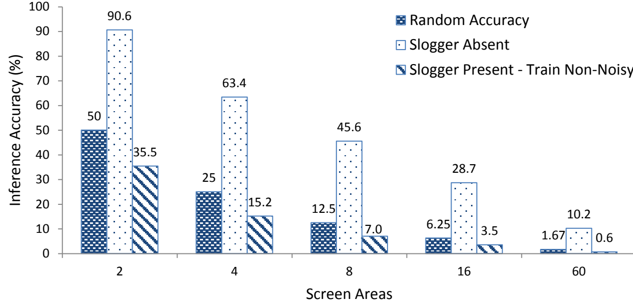
In the *In Hand* setting, inference accuracy of touchstroke inference model is reduced by nearly or more than 20% in almost all level of regions granularity. For 16 and 60 regions granularity level, inference accuracy drops to, nearly random accuracy, 10.4% and 2.6%, respectively.

5.3 Slogger against Sensor Fusion

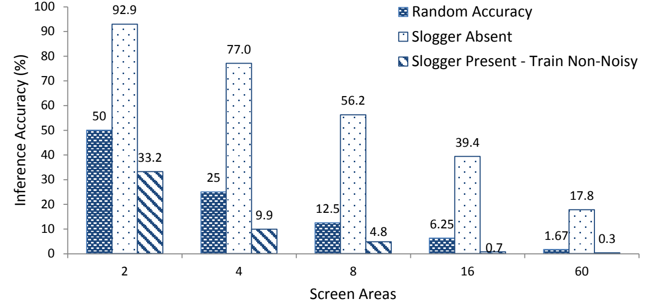
As the attacker can gain unfettered access to all the motion/position sensors on an Android device in a similar way as access to the accelerometer sensor, the attacker can utilize other sensors or a combination of sensors to enhance the inference accuracy. In this section, we first show that such a *fusion* of sensors can significantly improve the accuracy of touchstroke inference, and then evaluate the impact of Slogger against this powerful fusion attack.

5.3.1 Data Collection

We used the same data set that we collected to evaluate Slogger against touchstroke inference attack in the previous subsections. To

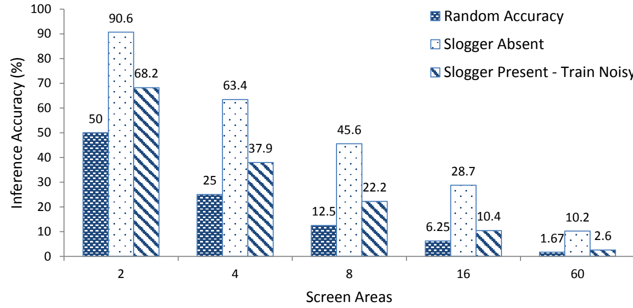


(a) In Hand

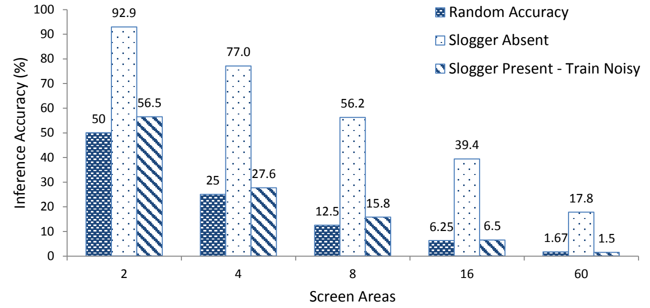


(b) On Surface

Figure 3: Touchstroke inference accuracy for different screen region granularity in the *Slogger Absent* scenario and the *Slogger Present, Attacker Trained with Non-Noisy Data* scenario.



(a) In Hand



(b) On Surface

Figure 4: Touchstroke inference accuracy by screen region granularity in the *Slogger Absent* scenario and the *Slogger Present, Attacker Trained with Noisy Data* scenario.

recall, earlier we considered the touchstroke inference attack based only on accelerometer sensor readings while now we consider the *fusion* of various motion sensors enhancing the touchstroke inference attack. The motion sensors used in our study are accelerometer, linear acceleration, gravity, gyroscope, and rotation.

5.3.2 Results

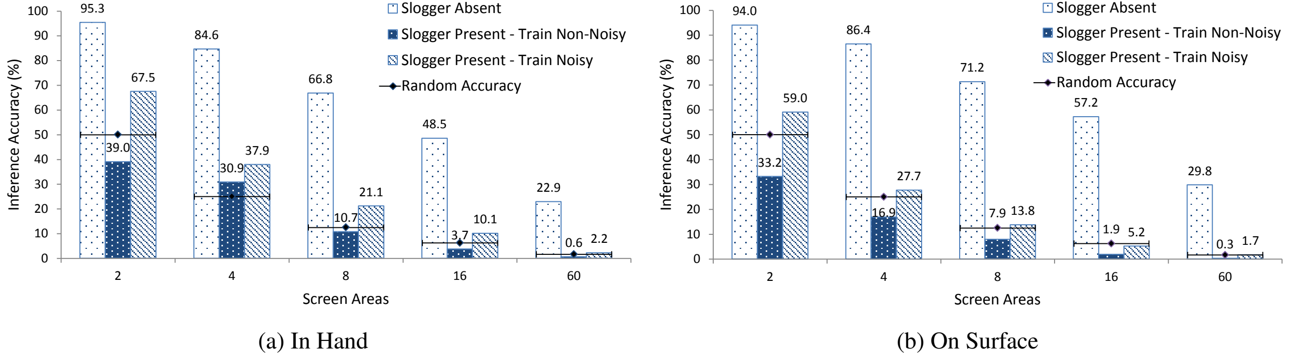
We extracted the same set of features for each of the motion sensors that we used for the accelerometer sensor. Out of all possible combination of sensors, we noted the sensor combinations which yield the maximum inference accuracy for each of the regions granularity levels in both the *In Hand* and *On Surface* settings separately. The results are shown in Table 3. As we can see, the accuracies resulting from the fusion are higher compared to the accelerometer only case.

In Slogger defensive scenarios, we evaluated the reduction in the maximum inference accuracy of the inference model while employing the fusion of multiple sensors. Results are shown in Figure 5. In the first scenario, where inference model is trained with regular stream of sensor readings and tested against Slogger injected stream of sensor readings, inference accuracies drop significantly below the random guessing inference. In the second scenario, where both the training and the testing stream of signals have Slogger injected noises, Slogger is able to reduce the maximum inference accuracy to nearly random inference in case of the *On Surface* setting, while in the *In Hand* setting, Slogger is able to reduce the accuracy by more than 25%, which is a significant degradation in the inference accuracy. This analysis shows that even if fusion of multiple sensors were to be used for the touchstroke inference attack, Slogger is still able to effectively defend against such attack.

5.4 Impact of Slogger on Benign Applications

Since Slogger works by injecting sensory noise, one natural question is whether it can have an adverse effect on benign applications that rely upon the sensor data. To this end, we performed an experiment to study the impact of Slogger on benign application. We tested Slogger with one of the Android applications, pedometer[1], in two different settings, *In Hand* and *On Surface*. Pedometer primarily records the number of steps the user has walked as well as the distance covered, walking time and speed per hour. When the phone was placed on a surface, while the pedometer and Slogger applications were running in the background, the step count as shown by the application was affected. However, when phone was held in hand in a similar setting, the step count was not affected. This is because the range of values of noise that Slogger injects is similar to the user typing movements. So, when the phone is placed on the surface, the difference between the injected values and the readings generated by the hardware sensors becomes high, contrary to the case when the phone was in hand, that fools the pedometer application to count it as a step. It is important to note that Slogger does not inject noise all the time, rather it is activated only when sensitive input is being provided by the user. Typically, in a real-world scenario, the user enters the sensitive input while holding the phone in hand - the scenario in which Slogger does not have noticeable impact on the benign applications such as Pedometer. We further tested Slogger with other benign apps, such as “fall detection” [2] and “shake to clear notes” [3], and confirmed that Slogger injection did not affect these apps.

[1]Pedometer – https://goo.gl/RFB64p
[2]FADE: fall detector – https://goo.gl/YFIsO0
[3]Any.do: To-Do List, Task List – https://goo.gl/7oIHZN



(a) In Hand (b) On Surface

Figure 5: Touchstroke inference accuracy by screen region granularity of *fusion of sensors* in presence and absence of Slogger.

Table 3: Maximum touchstroke inference accuracy gained for various regions granularity as a result of *Fusion of Sensors*, and respective sensor combination.

	Screen Areas	Accelerometer Only – Accuracy	Fusion – Maximum Accuracy	Sensor Combination Yielding Maximum Accuracy
In Hand	2	90.62%	95.34%	accelerometer, linear acceleration, gyroscope, gravity, rotation
	4	63.41%	84.62%	linear acceleration, gyroscope, gravity
	8	45.56%	66.83%	linear acceleration, gyroscope, gravity
	16	28.71%	48.51%	linear acceleration, gyroscope, gravity
	60	10.20%	22.91%	linear acceleration, gyroscope, gravity
On Surface	2	92.88%	94.01%	linear acceleration, gyroscope
	4	77.04%	86.42%	linear acceleration, gyroscope, rotation
	8	56.21%	71.25%	linear acceleration, gyroscope, gravity, rotation
	16	39.35%	57.12%	linear acceleration, gyroscope, gravity, rotation
	60	17.78%	29.78%	linear acceleration, gyroscope, rotation

6. DISCUSSION AND FUTURE WORK

Summary of Analysis: Our evaluation demonstrates that Slogger is able to defeat both the touchstroke detection and touchstroke inference steps, and hence it can defeat sensor-based touchstroke inference attack as a whole.

In both of the attack scenarios, *Attacker Trained with Non-Noisy data*, and *Attacker Trained with Noisy-Data*, Slogger is able to defeat the touchstroke detection attack. In the first attack scenario, the noise injected by Slogger completely prevented *Tap-detector* to detect any of the touchstrokes present in the sensor signal, while in the second attack scenario, Slogger is able to significantly reduce the detection accuracy down to as low as 32%. The touchstroke inference attack is also undermined by considerable amount with Slogger noise injection. Slogger is able to drop the inference accuracy nearly as low or even lower than the random guessing accuracy in almost all the scenarios. One exception is the *'In Hand', Attacker Trained with Noisy-Data* scenario, where detection accuracy still dropped significantly, by more than 20%.

In a realistic keylogging attack, where the attacker has to combine the touchstroke detection and inference steps, the error in the detection step propagates to the inference step, which would further lower the overall inference accuracy. This is because if the inference algorithm works with wrongly detected tap regions, the accuracy of inference model will degrade considerably. Even if we assume a scenario where the attacker achieves the highest possible accuracies in both of these steps (32% in touchstroke detection and 68.2% for touchstroke inference step for two-halved regions) in presence of Slogger, the overall inference accuracy becomes 21.76%, which is far below than the random guessing inference for two-halved regions.

Further, even if fusion of multiple sensors is used for touchstroke inference attack, which usually enhances the inference accuracy, Slogger is still able to effectively defend against such an attack.

All these results show that Slogger serves to provide an effective defense mechanism against sensor-based touchstroke logging.

Deliberate Noise Filtering: We argue that removing the noise injected by Slogger is a challenging task. First, Slogger injects the noise at random intervals, and the attacker cannot gain access to the sensor files and therefore there is no way that the attacker can get the information as to when the noise is injected. Second, the injected noise does not have a profile as it is random and therefore there is no way the attacker can try to reproduce the noise and then remove it from the signal. Third, the injected noise sometimes overwrites the original sensor values (or part of them), so any method that tries to delete part of the signal will delete some of the original readings as well and therefore degrade the original signal. Finally, the injected noise values lie within the range of the values corresponding to the user typing.

Touchstroke Inference over Multiple Rounds: We performed an experiment to check if the attacker can infer the touchstrokes after sniffing the accelerometer readings multiple times while the user is pressing the same sequence of buttons (e.g., in case of a PIN or password typed during each login attempt) in the (10×6 button grid). In our experiment, the user pressed on a sequence of 8 buttons for 20 times, while the phone is left on a surface, and we recorded the corresponding accelerometer readings. Then, for each button in the sequence, we predicted the pressed button as the one that got predicted in the majority of the trials. Without noise injection, the attack succeeded in predicting all the pressed regions (for 2 regions screen granularity), 6 out of 8 (for 4 regions screen granularity), and 5 out of 8 (for 60 regions screen granularity). In contrast, when Slogger was activated, the attack could only predict 5 out of the 8 for (2 regions screen granularity), 1 out of the 8 for (4 regions screen granularity) and 0 out of 8 for (60 regions

screen granularity). The results of this study show that Slogger is an effective mechanism for even defending such a multi-round, powerful attack. Since the injected noise is random, every time the attack inferred different touchstrokes in presence of Slogger.

Slogger to Defeat Other Side-Channel Attacks: Other side channel attacks based on motion sensors have been proposed in prior research, including location tracking and device fingerprinting. Location tracking side-channel attack, named "ACComplice", was presented in [11]. ACComplice is a malware that can track the location information of the users based on the accelerometer data. It uses the accelerometer readings to infer the trajectory and the starting point of the user who is driving. A device fingerprinting attack, called AccelPrint, was presented in [6]. The attack shows that each accelerometer has unique fingerprints which can be exploited for tracking users. These fingerprints are due to the hardware imperfections inculcated during the sensor manufacturing process, which makes every sensor chip respond to the same motion stimulus in a different way. Both these attacks are reviewed in detail in the next section. We believe that Slogger can be utilized in defeating such attacks given its ability to inject sensor events. To defend against ACComplice, Slogger can inject noisy accelerometer data to the system whenever a user is driving at random or specific intervals of time. To defend against AccelPrint, Slogger can inject accelerometer events with values similar to the accelerometer data used by AccelPrint to fingerprint the phone. However, this approach may also affect the benign apps which use sensor data for different purposes. Hence, Slogger should only inject negligible sensor events such that the benign apps are not affected and/or inject sensor events for only a short period of time (like in the Slogger defense against touchstroke logging attacks). Further work will be needed to test the viability of this defense mechanism.

7. OTHER RELATED WORK

In the current Android security model, motion, position and environmental sensors are considered *insensitive* resources – any app can read these sensors without any security permissions. However, many researchers have shown that many sensitive information about the user can be inferred only by monitoring these benign-looking sensors (especially the motion sensors), and thereby compromising the user privacy.

In this paper, we explored the effectiveness of, and defense against, touchstroke detection and touchstroke inference based on motion sensors by re-implementing two of the state-of-the-art attacks - Tap-detector [23] and ACCessory [17].

Another side channel attack for touchstroke logging is presented in [2]. They presented an attack based on accelerometer data to learn user's PIN/password or Android's pattern unlock while the user is unlocking her smartphone. This attack system uses machine learning techniques to infer four-digit PINs and password patterns. Since the accelerometer data is varied by subtle tilts and shifts, the approach normalizes the data, extracts 774 different features based on signal processing and polynomial fitting techniques, and then feeds these features to a classifier. It is worth noting that Slogger would affect the accuracy of such attack, as it injects random values in the accelerometer sensor.

Another interesting type of side-channel attack is motion-based location tracking. Han et al. in [11] presented "ACComplice", a malware which can track the location information of the users based on the accelerometer data. ACComplice uses the accelerometer readings to infer the trajectory and the starting point of the user who is driving, and thereby compromising the user privacy. ACComplice first tries to infer the trajectory based on accelerometer readings and then associates that trajectory to the most likely location on a map. One of the main challenges of this system is to deal with "drifting error". The drifting error occurs as the position of vehicle at time t depends upon the position at time $t-1$ and the displacement occurred at the time interval $[t-1, t]$. Also, the drifting error aggregates over time. In order to estimate the location accurately, Han et al. use a probabilistic dead reckoning method called Probabilistic Inertial Navigation "ProbIN". ProbIN treats sensor measurements only as observation of the underlying motion and maps the vehicle displacement based on a statistical model. As discussed in prior section, Slogger may be used as a defense system to address this attack.

Further, Marquardt et al. [14] showed that it is possible to infer key presses on a regular keyboard using motion sensor of a phone when the phone is placed two inches away. They developed an spying application, (sp)iPhone, that records and interprets the surface vibrations sensed by the phone's accelerometer to predict what was typed on the keyboard.

Moreover, a few sensor and device fingerprinting attacks have been proposed based on motion sensors. "AccelPrint" presented in [6] showed that each accelerometer has unique fingerprints which can be exploited for tracking users. These fingerprints are due to the hardware imperfections inculcated during the sensor manufacturing process, which makes every sensor chip respond to the same motion stimulus in a different way. They argued that the differences in responses across accelerometers are negligible for the higher level apps such that it does not affect their performance. The analysis of these negligible differences showed that these fingerprints emerge with consistency and can even be somewhat independent of the stimulus that generates them. To get the stimulus, they use the period when the vibration motor is turned on. In order to remain stealthy, the developed malware does not turn on the vibration motor itself, but rather waits for some other apps/event to turn the vibration motor on. The malware also uses the accelerometer data to detect this vibration.

Another fingerprinting attack is presented by Das et al. in [5] utilizing accelerometer and gyroscope sensors readings. Moreover, Das et al. presented two mitigation mechanisms, one is based on sensors calibration and the other is based on obfuscating the anomalies by injecting random noise. Both of the methods have shown decreased in the accuracy of the fingerprinting and shown to be promising to defeat device fingerprinting based on motion sensors imperfections. However, the authors did not show how to implement such mitigation mechanisms without modifying the operating system or the device manufacture. We believe that Slogger can serve as a viable defense to such attacks at the app level alone (i.e., without changing the OS or the kernel).

8. CONCLUSIONS

We presented Slogger, a practical defense to the problem of motion-based touchstroke logging attacks applicable to current Android devices. The Slogger app can be installed on the device easily using the ADB workaround (just like typical screenshot apps), without requiring to root the device or make any changes to the Operating System. Once installed, Slogger can protect the touchscreen input to any app by obfuscating the motion/position sensor readings with randomly generated noisy readings. It works invisibly in the background without affecting the user or other benign apps that need to use the raw sensor measurements. Slogger also boasts to defeat powerful and sophisticated attackers who may combine multiple sensors readings or use deliberate noise filtering mechanisms to infer the touchstrokes provided by the user.

Acknowledgements

This work is partially supported by National Science Foundation (NSF) grants: CNS-1209280 and CNS-1526524.

References

[1] A. Al-Haiqi, M. Ismail, and R. Nordin. Keystrokes inference attack on android: A comparative evaluation of sensors and their fusion. *Journal of ICT Research and Applications*, 7(2):117–136, 2013.

[2] 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*, pages 41–50. ACM, 2012.

[3] L. Cai and H. Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. *HotSec*, 11:9–9, 2011.

[4] J. Dai, X. Bai, Z. Yang, Z. Shen, and D. Xuan. Perfalld: A pervasive fall detection system using mobile phones. In *Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2010.

[5] A. Das, N. Borisov, and M. Caesar. Exploring ways to mitigate sensor-based smartphone fingerprinting. *arXiv preprint arXiv:1503.01874*, 2015.

[6] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi. Accelprint: Imperfections of accelerometers make smartphones trackable. In *NDSS*. Citeseer, 2014.

[7] S.-H. Fang, Y.-C. Liang, and K.-M. Chiu. Developing a mobile phone-based fall detection system on android platform. In *Computing, Communications and Applications Conference*, ComComAp, 2012.

[8] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 3. ACM, 2012.

[9] M. Frank, R. Biedert, E. Ma, I. Martinovic, and D. Song. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *IEEE Transactions on Information Forensics and Security*, Jan 2013.

[10] H. Gascon, S. Uellenbeck, C. Wolf, and K. Rieck. Continuous authentication on mobile devices by analysis of typing motion behavior. In *Sicherheit*, 2014.

[11] J. Han, E. Owusu, L. T. Nguyen, A. Perrig, and J. Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–9. IEEE, 2012.

[12] S. Hemminki, P. Nurmi, and S. Tarkoma. Accelerometer-based transportation mode detection on smartphones. In *ACM Conference on Embedded Networked Sensor Systems*, SenSys. ACM, 2013.

[13] P. G. Kelley, L. F. Cranor, and N. Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3393–3402. ACM, 2013.

[14] 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*, pages 551–562. ACM, 2011.

[15] 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*, pages 323–336. ACM, 2012.

[16] M. Mohamed, B. Shrestha, and N. Saxena. Smashed: Sniffing and manipulating android sensor data. In *Conference on Data and Application Security and Privacy*, 2016.

[17] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, page 9. ACM, 2012.

[18] D. Ping, X. Sun, and B. Mao. Textlogger: inferring longer inputs on touch screen using motion sensors. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 24. ACM, 2015.

[19] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[20] Y. S. Ryu, D. H. Koh, B. L. Aday, X. A. Gutierrez, and J. D. Platt. Usability evaluation of randomized keypad. *Journal of Usability Studies*, 5(2):65–75, 2010.

[21] D. Shukla, R. Kumar, A. Serwadda, and V. V. Phoha. Beware, your hands reveal your secrets! In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 904–917. ACM, 2014.

[22] Y. Song, M. Kukreti, R. Rawat, and U. Hengartner. Two novel defenses against motion-based keystroke inference attacks. *arXiv preprint arXiv:1410.7746*, 2014.

[23] 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*, pages 113–124. ACM, 2012.

[24] C.-W. You, N. D. Lane, F. Chen, R. Wang, Z. Chen, T. J. Bao, M. Montes-de Oca, Y. Cheng, M. Lin, L. Torresani, and A. T. Campbell. Carsafe app: Alerting drowsy and distracted drivers using dual cameras on smartphones. In *Mobile Systems, Applications, and Services*, MobiSys, 2013.