Gesture On: Enabling Always-On Touch Gestures for Fast Mobile Access from the Device Standby Mode

Hao Lü and Yang Li
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
hlv@google.com, yangli@acm.org

ABSTRACT
A significant percentage of mobile interaction involves short-period usages that originate from the standby mode—users wake up a device by pressing the power button, unlock the device by authenticating themselves, and then search for a target app or functionality on the device. These additional steps preceding a target task imposes significant overhead on users for each mobile device access. To address the issue, we developed *Gesture On*, a system that enables gesture shortcuts in the standby mode by which a user can draw a gesture on the touchscreen before the screen is turned on. Based on the gesture, our system directly brings up a target item onto the screen that bypasses all these additional steps in a mobile access. This paper examines several challenges in realizing Gesture On, including robustly rejecting accidental touches when the device is in standby, battery consumption incurred for continuous sensing and gesture-based user authentication methods for automatically device unlocking. Our analyses based on a set of user data indicated that Gesture On demonstrates a feasible approach for leveraging the standby mode for fast access to mobile content.

Author Keywords
Gesture-based interaction; gesture recognition; mobile computing; power efficiency; standby mode.

ACM Classification Keywords
H.5.2. [User Interfaces]: Input devices and strategies, Prototyping. I.5.2. [Pattern Recognition]: Classifier design and evaluation.

INTRODUCTION
A large number of mobile accesses have a rather short time span (e.g., a previous study reports 49.8% of the logged app usage is shorter than 5 seconds [3]) and originate from the standby mode of a mobile device [3,5], i.e., when the screen is off. To access a mobile functionality such as making a phone call, users will need to first press the power button to wake up the screen, unlock the phone by authenticating themselves, and then locate the target application before they can perform the desired action (see Figure 1a). These steps preceding a target action are always needed for many common mobile accesses, such as sending a text message, checking email, jotting down a note, or searching for a restaurant. They impose significant overhead for each mobile access from the standby mode.

Due to the frequent occurrences of these standby-originated accesses, reducing the overhead in these tasks can significantly improve the efficiency of mobile interaction. Existing commercial products and prior research literatures have investigated this issue in several aspects. Gesture Search alleviates the effort by allowing users to quickly access mobile content using mnemonic gesture shortcuts [10]. However, it still requires the user to activate the device and locate and invoke Gesture Search in the first place. The always-on speech feature has recently been enabled in Motorola's Moto X smartphone[1]. It allows the user to wake up the phone from standby to a speech input interface that is readying for a set of voice commands by speaking the phrase "Okay, Google Now". The person can

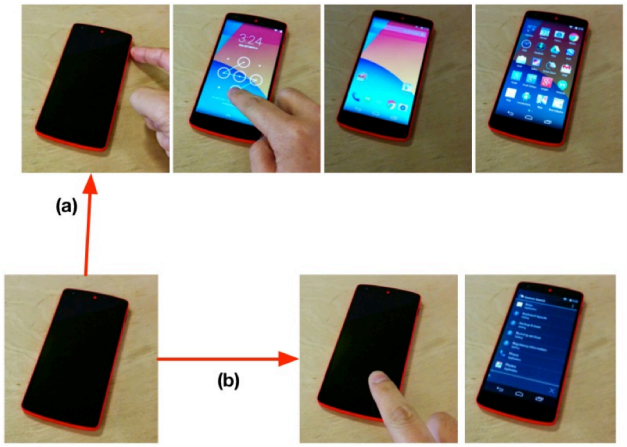

Figure 1. (a) Existing mobile systems require additional steps before the user can perform a target action, including pressing the power button, unlocking the device, and locating the target application. (b) Gesture On eliminates all these steps by directly accessing a target item from the device's standby mode by drawing a gesture on the touchscreen before it is turned on.

CHI 2015, Apr 18-23, 2015, Seoul, Republic of Korea
ACM 978-1-4503-3145-6/15/04.
http://dx.doi.org/10.1145/2702123.2702610

then speak out a command to quickly perform a common action, such as sending a text message, making a phone call, and navigating to a place. This always-on speech feature eliminates the extra interaction steps from waking up the device to getting to the target application. However, the method suffers several drawbacks. It requires the user to utter a special phrase before speaking a target command. In addition, speech input is not always appropriate or feasible (e.g., in a shared office, on a bus or in a noisy restaurant). Finally, further authentication is still needed if the desired action is considered privacy sensitive.

To address these issues, we developed *Gesture On*, a system that enables touchscreen gestures when a device is in the standby mode. With Gesture On, a user can draw a symbolic gesture on the touchscreen before the screen is even turned on; based on the recognition of the gesture, it directly brings up a target item onto the screen. Because symbolic gestures can often be drawn without visual feedback (such as trajectories), they can be articulated when the screen is off [7]. Gesture On overrides the default kernel behavior of mobile platforms by constantly monitoring and analyzing touchscreen input when the device is in standby. When a symbolic gesture is detected on the device touchscreen, Gesture On wakes up the device, authenticates the user based on the gesture, and performs the corresponding action, e.g., drawing a triangle to launch the Camera app or a circle to invoke the Alarm Clock.

In this paper, we examine several technical challenges in realizing Gesture On. First, Gesture On needs to differentiate the desired gesture input from accidental touches. There are many accidental touches registered on the touchscreen throughout the day when a device is in standby, e.g., when a user grab their device or when the device rests in a pocket. To avoid wasting computational power or triggering unwanted actions, Gesture On needs to identify and discard accidental touches as fast as possible. We also investigated the potential impact on battery consumption due to the continuous sensing as required by Gesture On. Second, to avoid a dedicated step for user authentication, we intend to authenticate the user based on the biometric information as demonstrated by a touch gesture itself.

The paper makes the following contributions:

- A gesture shortcut system that is embedded at the core of a mobile platform, which enables touchscreen gesture shortcuts when the device is in the standby mode;
- A set of methods for differentiating symbolic (particularly alphanumeric) gestures from accidental touches in the standby mode, and a thorough analysis over the characteristics of these touch behaviors;
- An examination of the impact on mobile battery consumption with the continuous sensing of Gesture On and the architectural design for addressing it;
- A preliminary investigation of methods for authenticating users based on their touchscreen gestures, and how gesture-based authentication can be integrated into an interaction flow.

In the rest of the paper, we first provide an overview of the related work. Next, we discuss our methods of learning and validating a gesture recognizer for differentiating desired gestures from accident touches, including our data collection procedure and an analysis over the collected data. Then, we investigate implicit user authentication methods, by creating and validating a classifier for authenticating users based on their touchscreen gestures. Finally we discuss our implementation and limitations and then conclude the paper.

RELATED WORK

To ease the effort for searching and activating a target functionality or command, previous work has investigated a variety of gesture shortcut systems [2,10,16]. However, these systems only address part of the problem in that they themselves need to be invoked before a user can benefit from these approaches. The benefit of reducing the interaction overhead from the very beginning of the interaction process—the standby mode—has attracted significant effort from both research and commercial products. A large body of work exploits various input capabilities of mobile devices to allow waking up a device and launching an application faster.

Motion gestures utilizing on-device accelerometers and gyroscopes can wake up the device from standby and launch a specific action. For example, many smart watches can be turned on when a user twist the wrist[2]. Google Glass[3] can wake up when a user looks up beyond a certain angle. A flipping gesture [18] is used by Moto X to allow a user to launch the camera for taking photos.

Touchscreen have also been used to wake up devices and perform predefined actions. For example, Nexus 9 tablet [4] allows a user to double tap the touchscreen to wake up the device. Previous methods have also demonstrated using a directional swipe to wake up the phone and execute an action that corresponds to the direction of the swipe[5]. Compared to these methods, Gesture On allows a richer set of gesture input by enabling alphanumeric gestures. It is the first work that examines accidental touches and the user authentication challenges using these gestures.

Other on-device sensors including proximity sensor, light sensor, IR sensors have also been used to wake up a device. For example, Moto X smartphones employ all these sensor data to infer when a user is intending to check the phone and automatically wakes up the phone to show useful information such as time and notifications.

[2] Android Wear. https://www.android.com/wear/
[3] Google Glass. https://www.google.com/glass
[4] Nexus 9. www.google.com/nexus/9/
[5] Android lock screen. http://www.android.com

Lastly, speech input through microphones is an expressive modality for waking up the device and performing actions. Again with Moto X, a user can speak the specific phrase, i.e., "Okay, Google Now", to wake up the phone to a voice listening interface[6] that is readying for a rich set of voice commands. In contrast, Gesture On does not require a separate activation step and allows users to directly articulate their target gestures for both activating the device and bringing up the target action.

Prior work has extensively studied methods for recognizing touchscreen gestures, e.g., [2,9,11,22]. Most of the prior recognition systems focused on gestures with well-defined touch characteristics, such as the trajectory of a rectangle shape [9,22]. These systems do not attempt to reject out-of-vocabulary gestures, and for any unknown input, they always output the best matched gesture class from a given gesture set. In contrast, accidental touches studied in this paper often do not have well-defined trajectories. Wed need to efficiently rule them out for further processing.

Little work has been devoted to discerning intended gestures from accidental touches. Recently, Schwarz et al. [19] used a set of spatiotemporal touch features (e.g., touch area, trajectory size, trajectory smoothness) to differentiate and filter out touch events from the hand palm. The accidental touches as concerned by us include palm touches, e.g., when the user fetches a phone. Thus, we borrowed several features of [19] to differentiate trajectory-based gestures from accidental touches.

Another related area is user authentication on mobile devices, which has been a longstanding problem. Much work has focused on replacing user passcodes with other forms of authentication methods that are more user friendly. For example, fingerprint readers[7] allow users to press or swipe their fingers to authenticate themselves. Trusted physical tokens [8] are another genre of technologies that authenticates the user by touching a trusted token on the device via NFC[8] or by simply connecting a trusted token via Bluetooth[9]. Face recognition has also been used in commercial mobile platforms such as Android devices , which unlocks the device by simply asking the user to face the device camera.

Previous work has also looked at touchscreen gestures for user authentication. Feng et al. [4] investigated methods for identifying users by observing a series of simple touchscreen gestures (e.g., swipe up, swipe down), but in a post-login setting. GEAT [20] authenticates users by how they perform (about 3) touch gestures from a set of 10 simple gestures. Because these systems need multiple gesture observations, they are not ideal for the authentication scenario we want to address here that only observes a single unknown alphanumerical gesture.

[6] Google Now. https://www.google.com/landing/now/
[7] iPhone 6. http://www.apple.com/iphone-6/
[8] Motorola Skip uses NFC to unlock smartphones.
[9] Android 5 supports adding any Bluetooth device as trusted token.

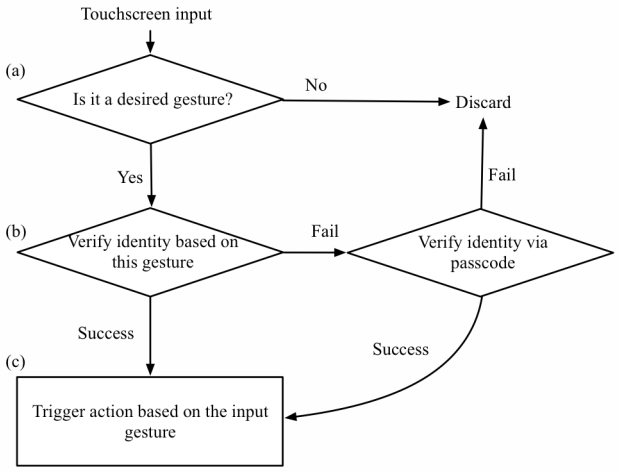

Figure 2. The processing flow of Gesture On: The two main components are (a) filtering out accidental touch input, and (b) verifying user identity based on this gesture. The gesture action component (c) is pluggable and we use Gesture Search in our first prototype to trigger search activity.

Handwritten signatures have been extensively investigated for user authentication or identification [17]. Sherman et al. [21] studies the security and memorability of using user-defined free-form multi-touch gestures for authentication. These forms of data have rich biometric characteristics of the user.

Gesture On investigates authenticating users only based on their touch gestures for launching actions, which avoids the need of additional hardware or extra user action. In contrast to the existing touch-based authentication, Gesture On does not authenticate against a user-defined signature, but a set of simple gestures that contain much less biometric characteristics.

Contextual information has also been used for user authentication. For example, a user can set home location to be trusted in Android and the device will automatically authenticate the user at home. CASA [6] provides a probabilistic framework for choosing an appropriate form of authentication according to the available contextual information. It balances between security and usability. We also took into account such a tradeoff in Gesture On that can benefit from these authentication frameworks.

THE DESIGN OF THE GESTURE-ON SYSTEM

In this section, we discuss the design of Gesture On. Assume Alice wants to make a phone call to her friend Bob. Typically, she would take out the phone, press the power button, unlock the phone with a passcode, locate and tap on the contacts app, and then search for Bob.

With Gesture On, after taking out the phone, Alice only needs to draw a letter "B" on the blank screen. Gesture On will wake up the phone, authenticate the user, and search the device with the gesture using Gesture Search [10]. Based on a list of presented search results, Alice can tap on

Bob's name from the search results to immediately make a phone call.

The detailed processing flow is illustrated in Figure 2. When a sequence of touch events occurs, Gesture On first determines whether the touch events are gesture input intended by the user. If not, it ignores the touch events, otherwise Gesture On wakes up the device and continues to process the gesture. If the user originally has set passcode protection, Gesture On will first verify if the input gesture is from the same user—authentication. If the gesture does not authenticate the user, the passcode guard will be shown and a correct passcode needs to be provided before the Gesture On can proceed. After authentication, Gesture On will launch Gesture Search with the input gesture as a search query.

Gesture On processes multi-touch input to filter out accidental touches, and authenticates the user. The handling of a detected gesture to fire desired actions can be delegated to a gesture shortcut system. For example, our current implementation sends the detected gesture as a query to Gesture Search [10] that allows random access to mobile content using alphanumerical gestures. Alternatively, we can connect Gesture On with other gesture shortcut systems, e.g., drawing a rectangle to invoke the camera functionality and a circle to bring up the Clock app.

In addition to flexibility, an important reason to separate the detection of target gestures (from accidental touches) from the recognition of them for invoking actions is to reduce unnecessary computation thus battery consumption caused by the accidental touches. As discussed later in the paper, detecting gestures from accident touches needs to run continuously but the recognition only needs to be performed when a gesture is detected. By separating the two and making detection an independent and compact component, we can run the detection on a low-power DSP to filter out the accidental touches while the CPU is in the sleep mode. Later, we will discuss our experimentation with such an architecture by evaluating of our gesture detection component on an ARM Cortex-M3 DSP[10], which shows that the separation allows a minimal impact on the power consumption.

The rest of paper focuses on the two key components to Gesture On are (1) algorithms for filtering out accidental touches, and (2) methods for authenticating users based on their input gestures.

DETECTING GESTURES FROM ACCIDENTAL TOUCHES

When a mobile device is in the standby mode, many screen contacts could register touch events. These touch events are accidental and not meant for initiating interaction. For example, when a person takes out a phone from the pocket or simply grabs the phone in their hand, their fingers could easily contact the screen and register touch events. Even when a phone rests in the pocket, depending on the fabrics of the cloth, screen contacts with the body over the cloth could also register touch events. Discerning intended gestures from accidental touches is crucial so as to avoid battery drain that is caused by unnecessary processing of the touch events and false activation of unwanted actions. In this section we discuss our methods for filtering out these accidental touches.

Most touchscreens nowadays detect multi-touch input. Although Gesture On focuses on enabling single-touch gestures for mobile access, both the accidental and the intended gesture input may contain touch input from multiple body parts simultaneously. For instance, when a user grabs a phone, multiple fingers could contact the screen, which generate accidental touches. Accidental touches can also co-occur with intended gestures. For example, while gesturing with one finger, a user may have touched the bezel of the phone unconsciously with the holding hand, which has become frequently occurred as many smartphones or tablets today are becoming frameless. It is even difficult for the user to realize accidental touches in the standby mode because there is no visual feedback in this mode while the screen is in sleep. Consequently, although the target gestures we aim to support are single-stroke, we need to handle multi-touch input that consists of one or multiple parallel touch trajectories (or touch strokes).

In our method, we classify each touch stroke as either accidental or intended given its context in a multi-touch input—the stroke being evaluated in relation to other co-occurring strokes as well as other sensor data such as acceleration. The classification only takes place at the end of each stroke, i.e., when a touched finger lifts up.

Data Collection

To understand how intended touch gestures are different from accidental touches, we collected touch data about these behaviors from users that consists of two consecutive stages: one for collecting accidental touch events and one for collecting intended gestures.

In the first stage of the experiment, a participant was given a customized Nexus 5 smartphone and was asked to use it as their primary phone (replacing their own phone) for a minimum of three consecutive days. The customized Nexus 5 contained a modified Linux kernel that we instrumented to enable logging touch events when the phone is in the standby mode and an Android app that we developed to log other types of sensor data. Because replacing the kernel on the phone can cause factory reset, we could not use participants' own devices for this study.

The second stage involved a laboratory study session in which we asked each participant of the first stage to perform a set of gesture tasks on the same device. Each participant needed to draw a set of symbolic gestures in five conditions involving both sitting and walking situations.

[10] http://www.arm.com/products/processors/cortex-m/cortex-m3.php

The sitting situation includes: 1) the user gestures on the phone that is resting on a table, 2) the user gestures with one hand and holds the phone with the other hand—the two-handed situation, and 3) the user gestures with the thumb of the hand that is holding the phone—the single-handed situation. The walking situation includes 4) the two-handed and 5) the single-handed situation while the user is walking.

For each of the 5 conditions, our data collection tool instructed the user to draw 3 samples for each alphanumeric symbol (i.e., the 26 English letters and 10 digits). When the instruction for drawing a specific gesture symbol shows at the top of the screen (see Figure 3), the participant is supposed to draw the gesture on the blank canvas and no visual feedback is provided while the participant performs the gesture. The order for which symbol to draw is randomized. This procedure resulted in a total of 540 gestures per participant.

We collected the same types of sensor data when touch events occur in both stages of the data collection study. The first set of data comes from the touchscreen sensor. For each touch event, we collect:

- The number of touches (each with an ID);
- The action of the touch event (i.e., move, release);
- The on-screen location of the touch event;
- The size (in pixels) of the touch area;
- The pressure associated with the touch event.

The second set of sensor data comes from other sensors. We recorded the readings from the proximity sensor, light sensor, accelerometer, magnetometer, and gyroscope. While these sensor data are not directly related to touch events, prior work has shown the strong correlation between these sensor dimensions and the screen touches [14]. We sampled these sensor readings within a 2-second window before, during and after each touch event.

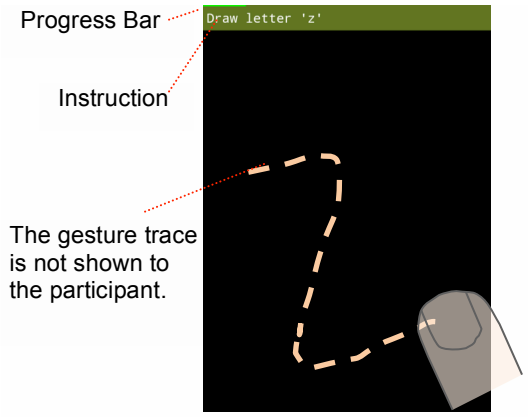

Figure 3. The interface for collecting gestures in the laboratory study. To simulate our target situation where no visual feedback is provided on the screen in the standby mode, the interface does not render the trajectory of the gesture while the participant performs it. The task automatically progresses when each gesture is finished.

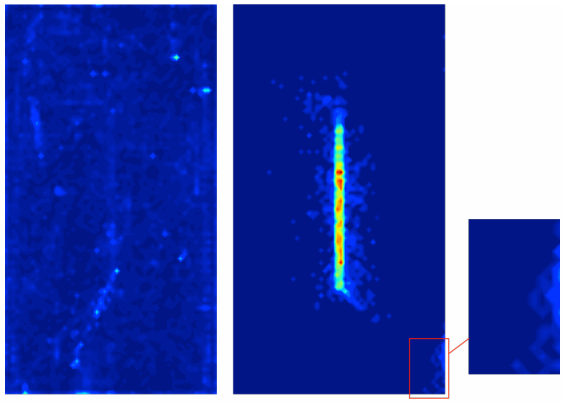

Figure 4. The heatmap of stroke locations as measured from the farthest point of a stroke to the bezel for (a) accidental touches, and (b) desired gesture inputs.

Collected Data and Features

We recruited 14 participants (8 female, 6 males, aged from 20-30). All were Nexus 5 users and right-handed. The total time span of the first stage of data collection was 1,359 hours, about 4 days per participant on average. During this period of time, a total of 3,834 screen-turned-on events were observed, about 68 screen-turned-on events per day per participant on average.

For the accidental touch data, we collected a total of 642,670 touch strokes. Notice that we treated all the touches occurred when the screen is off as accidental because no touch interaction is supported in the standby mode during the data collection. For intended gestures, we acquired a total of 9,995 touch strokes from the laboratory session. Based on this dataset, we analyzed different characteristics these two types of behaviors demonstrate, by which we extract useful features for automatically classifying them for our gesture detection purpose.

On-Screen Locations

We found accidental touches and intend gestures have distinct location distributions. A touch stroke consists of a sequence of touch points and we use the point that is the farthest to the bezel as the stroke's location. Our data indicated that strokes of accidental touch were scattered across the screen and many were near the bezels (see Figure 4a). In contrast, strokes of intended gestures were mostly cluttered at the center of the screen (see Figure 4b). However, we observed a noticeable amount of touch strokes of intended gestures occurred at the lower right corner of the screen (see Figure 4b). By looking into the data, we found all these near-corner touches co-occurred with other finger touches for articulating gestures, during the single-handed condition. Because all of our participants were right-handed, we infer that these touch events were actually resulted from the participants' accidental palm contact with the touchscreen.

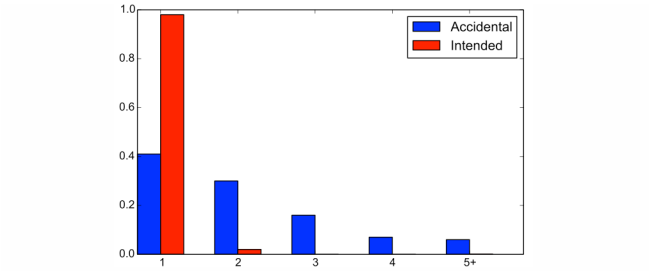

Figure 5. The distribution of the number of simultaneous touches from accidental touches and desired gestures. 60% of accidental touches involved more than 2 fingers or body parts, while the majority of intended gestures involved a single touch finger.

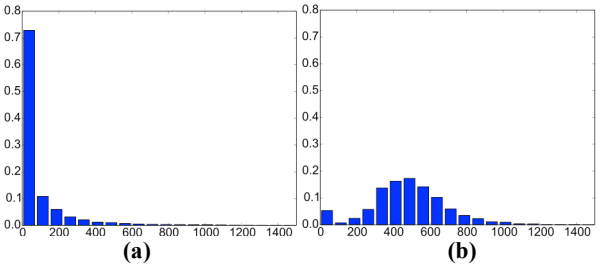

Figure 6. The distribution of the larger dimension of the bounding box of each touch stroke. The majority of accidental touches had small bounding boxes (a). In contrast, the bounding size of the intended gestures were widely spread (b). The touch strokes that had small bounding boxes in (b) were mostly resulted from part of a multi-stroke gesture such as the dots in letter "i" or "j".

Number of Simultaneous Touches

Another important aspect that we examined was simultaneous touches from multiple fingers. Single-stroke gestures, i.e., our target gesture type, should ideally only involve a single finger touching at a time. However, as discussed previously, accidental contacts with touchscreens can result in simultaneous touches while performing a gesture, although such situations are extremely rare compared to accidental touches that often involved more than one finger registered (see Figure 5).

Touch Movement

The movement range of a touch stroke can be a great indicator to whether the stroke is intended or accidental, because many intended gestures have a trajectory with a significant length while accidental touches involve relatively less movement. Our data confirmed our intuition that the touch strokes of intended gestures often had much larger bounding boxes than accidental touches (see Figure 6).

Time Duration

We hypothesize that touch strokes in intended gestures would have relatively more concentrated time durations, while accidental touches are more random and could have very brief or extended time duration, e.g., when grabbing the phone in hand. This hypothesis was confirmed by our data (see Figure 7). The majority of accident touches have a very short duration while some lasted for an extended time period (>1400 milliseconds).

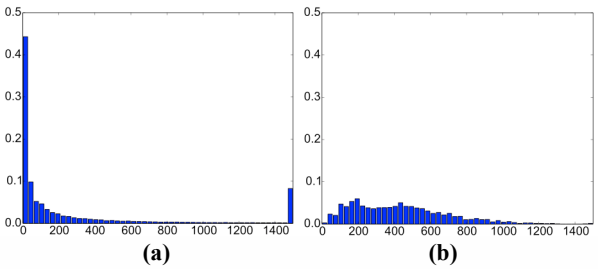

Figure 7. The distribution of time duration (ms) of a touch stroke: (a) the strokes from accidental touches mostly had a very brief duration with a long tail spread for an extended time period, and (b) the strokes from intended gestures had a relatively more concentrated duration.

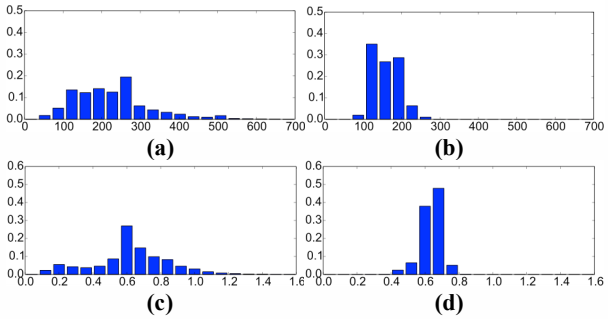

Figure 8. The distribution of touch width and pressure: (a) the widths from accidental touches, (b) the widths from intended gestures, (c) the pressure from accidental touches and (d) the pressure from desired gestures. In both metrics, accidental touches resulted in larger variations.

Touch Size and Pressure

A user typically performs an intended gesture by drawing with the fingertip as in our laboratory study that leads to a relatively consistent contact area size and pressure value. In contrast, accidental touches could be much more varying in these measures. For example, when a phone rests in the back pocket of a jean, the entire screen could be pushed against the body and the touch area from such a contact can be large so is the sensed pressure. Because the Nexus 5 phone, i.e., our experimental device, is only able to detect the major width of a touch area (a hardware limitation), we use the touch width instead of the actual size to characterize touch area differences between these two types of touch behaviors (see Figure 8).

Proximity

A proximity sensor is able to detect nearby objects and their distance from the sensor. However, Nexus 5 phones only report two values, 0 for "near" and 5 for "far", to indicate whether there is an object within a given range (i.e., 5cm). For smartphones, the proximity sensor is typically located on the top near the headset for detecting if the person is holding the phone against their ear during a phone call, to disable the screen to avoid accidental touches.

Because the proximity sensor is located at the top bezel, it is difficult to trigger the "near" signal while the user is gesturing on the touchscreen (see Figure 4). However, because accidental touches are more scattered around, the proximity sensors can be more easily triggered to send a

"near" signal. The statistics about proximity sensor readings based on our data has also confirmed this (see Figure 9). Because the proximity reading may change in the course of a gesture articulation, i.e., switching between 1 and 5, we used the average proximity for each sample that resulted numerical values between 1 and 5 as shown in Figure 9. Overall, more than half of the accidental touches took place when the "near" signal was triggered. In contrast, few intended gestures were associated with the "near" proximity value.

Motion Sensors

Prior work has shown that motion signals can be useful to assist touch screen interaction and to enable finger tapping beyond the touchscreen [13,14]. We noticed that accidental touches and intended gestures generated different distributions for the standard deviation of the acceleration magnitudes during a touch stroke (see Figure 10). Intuitively, when performing a gesture on a touchscreen, users try to keep the phone stable, which might not be the case for accidental touches.

Features for Learning

Based on the above observation from our dataset, we choose the measures listed in Table 1 as our features in training a classifier for detecting intended gesture input from accidental touches.

Training and Validation

We chose to use a decision tree model for our gesture detector because our features are good indicators themselves. We also experimented with other classification models that achieved similar precision and recall, but a decision tree has relatively lower computational cost and is easier for us to port the detector onto low-power DSP chips for continuous sensing as discussed in the following sections.

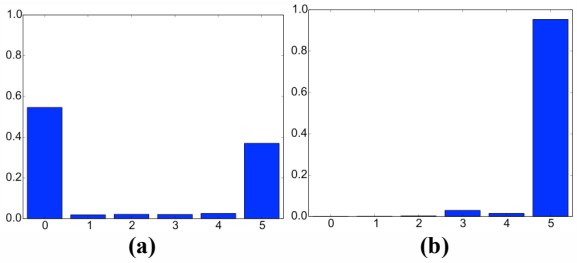

Figure 9. The distribution of the average proximity during a touch stroke: (a) the proximity during accident touches. Many touches that were recorded as "near" might have taken place when the phone was a pocket, and (b) the proximity of intended gestures is rarely detected as "near".

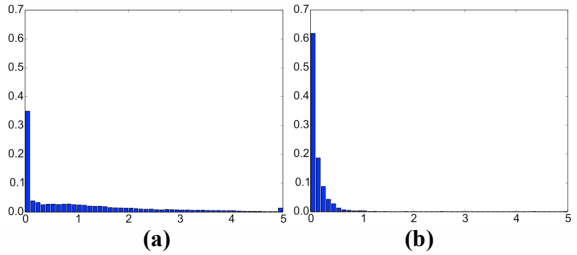

Figure 10. The distribution of the standard deviation of the acceleration magnitudes during a touch stroke: (a) the distribution of accidental touches, and (b) that of intended gestures. The acceleration from accidental touches tends to have a larger deviation and thus is less stable.

1	Maximum distance of the stroke to device bezels
2	Larger dimension of the bounding box of the stroke
3	Time duration of the touch stroke
4	Maximum number of simultaneous touches when the stroke is generated
5	Maximum width of a touch area associated with the stroke
6	Maximum pressure of a touch area associated with the stroke
7	Average proximity while the stroke is generated
8	Deviation of acceleration magnitudes in the course of a stroke

Table 1. The features used for detecting intended gestures from accidental touches.

For evaluation, we used a cross-user validation strategy. Each user's data was classified using a model that was learned from all other users' data. This simulates the user-independent setting where learning from specific users is not required after the system is deployed. Based on our dataset, we acquired 98.2% precision and 97.6% recall on detecting gestures from accidental touches. For accidental touches, this translates to 0.028% of them being misclassified as intended gestures, which is about 3 false activations per user per day.

USER AUTHENTICATION BASED ON A GESTURE

After a touch stroke is detected as an intended gesture, Gesture On checks if the device is locked with a passcode. If it is not locked, the gesture is directly sent to the connected gesture shortcut system for triggering the target action. If it is locked, Gesture On will try to authenticate the user based on the gesture, to avoid the overhead of requiring an additional authentication step. If the authentication fails, Gesture On will fall back to the default system authentication interface, e.g., via a passcode. Unlike prior work for user authentication using handwritten signatures, a gesture here often has much simpler trajectories, such as a circle for letter 'o'. These gestures have much less biometric information than a signature does. In this section, we discuss an initial investigation into this problem.

Authenticating a user through a simple gesture reduces interaction overhead but can sacrifice security due to false accepted gestures. The more accurate Gesture On can authenticate a user, the higher efficiency and better security it can achieve. To authenticate a user based on a single symbolic gesture, the system would need to first collect a sample of gesture data from the user before the user can use it, similar to collecting fingerprint data before using the fingerprint reader to authenticate. Since the biometric information might vary when a person draws different

symbols, we require a user to draw at least one sample for each gesture symbol. The more gesture samples there are, the better the system can model the user's biometric profile.

To formalize our problem, verifying a person here is to calculate the probability that a person is as claimed given the gesture, i.e., $P(u(g) = u^*|g, G^*)$, where $u(g)$ is the person who perform the gesture g, u^* is the claimed user—the person who owns the device, and G^* is the pre-collected gesture sample set of u^*. Because the biometric information can vary from symbol to symbol, we introduce a hidden variable s for the gesture symbol that the person intends to perform by which gives the following equation:

$$P(u(g) = u^*|g, G^*) = \sum_{s \in S} P(u(g) = u^*|g, s, G^*)P(s|g)$$

where S is the target symbol set, G^* is the set of gesture samples for S collected from the owner u^*. $P(s|g)$ can be computed using a user-independent gesture recognizer such as a handwriting recognizer. We can calculate $P(u(g) = u^*|g, s, G^*)$ as the following:

$$max_{g_i \in G_s^*} P(u(g) = u(g_i)|s)$$

$P(u(g) = u(g_i)|s)$ is the probability of whether two gesture samples are authored by the same user given that these gestures are intended for a specific symbol. In the following section, we discuss our initial exploration in estimating this probability by learning a binary classifier for each gesture symbol.

Data Collection

Although the gesture data we collected in our laboratory study can still be used here, it is limited in the sample size (only 14 participants), which is insufficient for training and validating our classification method. Thus, we used another dataset that was collected from 94 participants on their own touchscreen mobile devices, via a crowdsourcing platform [1]. Participants finished the data collection tasks remotely on their own rather than in a controlled laboratory environment, which resembles a realistic situation.

Each participant contributed 5 samples for each English letter. This amounts to 470 samples per letter symbol. Since the classification involves a pair of gestures, there are C_{470}^2 =110,215 training pairs for each letter symbol.

Feature Extraction & Classification

Each gesture sample pair consists of five groups of features. First, in the preprocessing step, each gesture in the pair is resampled to 129 equidistant points and is aligned with each other using Protractor [9]. These points split the gesture into 128 path segments, and we then partition these segments to form 8 segment groups with each group containing 16 consecutive segments (thus 17 points). Based on these segment groups, we compute the following features for each pair of gesture samples.

- $d_i, 1 \leq i \leq 8$. d_i is the cosine distance between the i-th group's point coordinate sequence of the two gesture samples in the pair;
- $c_i, 1 \leq i \leq 8$. c_i is the cosine distance between i-th group's point curvature sequence of the two gestures;
- $s_i, 1 \leq i \leq 8$. s_i is the ratio between the i-th group's displacement of the two gestures. The displacement is the Euclidean distance between the beginning and end points of a group;
- $v_i, 1 \leq i \leq 8$. v_i is the ratio between the i-th group's velocity of the two gestures. The velocity is the displacement divided by the duration of a group;
- $a_i, 1 \leq i \leq 8$. a_i is the ratio between the i-th group's acceleration of the two gestures. The acceleration is the velocity divided by the duration.

On top of the above features, we used the ratio between the bounding circle's diameters of the two gestures. The displacement, velocity and acceleration have been used previously for signature-based verification [17]. In contrast to prior work, based on these features, we used a data-driven approach and trained an SVM classifier with radial basis kernels for each letter symbol.

Validation and Results

Similar to evaluating our gesture detector, we split our dataset for training and testing across users. For each letter symbol, we randomly pick 5 participants (a total of 25 gestures) to create a test dataset of 300 sample pairs. We use the rest participants' data to form the training sample pairs. We chose 5 participants for testing because it simulates a scenario of 4 attackers who try to pass the authentication test—a reasonable amount of attacks.

For each letter, we repeat the above process for 5 times and report the average false positive rate (or false acceptance rate, FAR) and false negative rate (or false rejection rate, FRR) (see Figure 11). FAR reflects how easily a person can break in the device (the vulnerability). FRR indicates how likely the system will reject an authentic user's gesture. Because Gesture On falls back to the default passcode interface when authentication fails, higher FRR means the reduction to the interaction efficiency. The average false positive rate was 25.8% and the average false negative rate was 25.5% across gesture symbols.

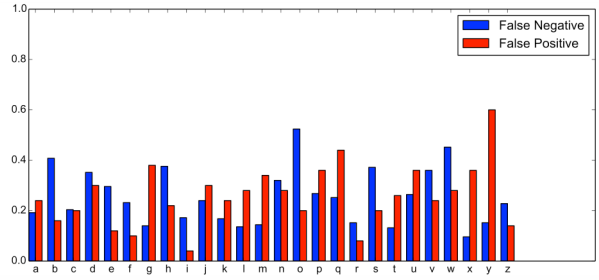

Figure 11. The false negative rate and the false positive rate for each letter symbol. Gestures for some symbols are more discriminative for differentiating users than others.

There are two extremes along the spectrum of interaction efficiency and security. Gesture On always fires an action without the authentication process (FAR=100%). This maximally reduces the authentication overhead, but at the expense of security. In the other extreme, Gesture On always asks the person to go through the passcode interface (FRR=100%), which does not reduce authentication overhead, nor sacrifice security.

It is worth mentioning that we can thus adjust the decision boundary of the binary classification to trade efficiency for security, or vice versa. We can expose this decision to users and let them finely-tune such a balance. For example, we offer three levels of authentication in a setting UI: a default level using a .5 decision threshold that yields the above FAR and FRR, a high-security level using a higher threshold for a low FAR, a low-security level using a lower threshold for a low FRR.

Before implementing our own solution, we experimented with a state-of-the-art solution from signature verification research [17]. With this previous method, we were only able to achieve 35%-40% FRR and FAR on single symbol verification. The fact that this previous method was able to achieve less than 2% FRR and FAR [17] for signature-based authentication indicated that a single-letter symbol often contain significantly less biometrics than handwritten signatures.

IMPLEMENTATION

Gesture On consists of two components. First, we implemented a virtual driver (for Nexus 5) in the Linux kernel that listens to and processes touchscreen events when the device is in the standby mode. The driver writes all the touch events to on-device storage files in our data collection study. To enable touch sensing in the standby mode, we modified the kernel to keep the touch sensor and its microcontroller active, and held a wake lock on CPU.

Second, we implemented an Android service that listens to all other sensor events and the touch events from our driver. This part can be alternatively implemented in the driver as well but it is much easier to implement using Android platform for our fast prototyping purpose. The communication between the service and the driver is through *sysfs* [15]. We used Weka [12] to train and validate our decision tree and SVM models.

POWER EFFICIENCY

The implementation of Gesture On that we discussed serves the purposes for data collection and prototyping and it works around the existing architecture of Nexus 5. However, while utilizing the sensors is relatively cheap, such an implementation requires Gesture On to hold a wake lock on CPU that consumes precious battery power.

A more power efficient solution is to move the gesture detection component that requires continuous sensing to a separate, low-power DSP (i.e., a sensor hub). The DSP only wakes up the CPU when a gesture input is detected, for more expensive computation such as gesture recognition and user authentication.

In fact, many today's mobile devices have already been utilizing such an architecture with a low-power DSP solution (MPU) for continuous sensing. For example, Apple iPhone 5s uses a M8 for processing sensor data in the background; Motorola Moto X uses a TI C55x DSP[11] for always-on voice command. This architectural design has a much less impact on the battery life.

Due to the limited access allowed to the current platform cores, we are yet able to harness these on-device DSP for our prototype. Instead, to examine the power efficiency Gesture On in such an architecture, we used a standalone low-power DSP through a development board. We chose NXP ARM Cortex-M3 as our DSP that has been used in iPhone 5s, and LPCXpresso1549 as the development board that offers good development support.

We ported our gesture detection component onto the DSP and used a Monsoon power monitor to measure the energy consumption of the entire development board. The development board uses a constant voltage of 5.04V during the experiment. We let the DSP perform 50,000 detection tasks, each consisting of featurizing and classifying a touch stroke with 30 touch events and 120 acceleration readings. We subtracted the energy consumption when the DSP is idle from that of the detection tasks. The mean energy consumption of each detection task is 0.02μAh, which is negligible compared to the entire battery capacity of Nexus 5 (2300mAh). Because there were about 811 accidental touch strokes per day per user based on our data, the average battery consumption for ruling out accidental touches is about 16μAh.

DISCUSSION

There are several ways to improve our authentication accuracy. We can utilize other sensor data such as the accelerometer in addition to the trajectories. The acceleration associated with touch events has been shown effective on user authentication [20]. Another direction is to allow a user to perform more than one gesture although this approach requires an additional authentication gesture that hampers the interaction efficiency. Alternatively, we can allow the user to use a pre-determined secretive gesture prefix or postfix for the desired gesture, e.g., a pigtail postfix. A prefix or postfix still allows the user to both authenticate and issue a target gesture in a single step. This approach can improve the authentication accuracy with introducing minimal extra effort for gesturing.

Our user authentication method requires a user to provide at least one example for each gesture in the target set. This requirement might be acceptable for 36 alphanumeric symbols. However, it is not feasible to a language with a large number of characters, e.g., Chinese characters. Future

[11] http://www.ti.com/lsds/ti/dsp/c5000_dsp/overview.page

work can explore the approach of clustering characters with a similar biometric profile such that such a requirement can be relaxed.

Our learned model for detecting gestures from accidental touches may not apply to another device with a different form factor. We conducted our feature analyses based on the data collected from Nexus 5 smartphones and some features are dependent of the specific form factor of the device (e.g., the bezel distance would be different on a larger device). This limitation would require device manufacturers to train a model for each device form factor.

Lastly, while our evaluation validates the accuracy performance and feasibility with Gesture On in an offline fashion, a user study with the deployed system is necessary to understand how such a system impacts the user's mobile experience. In the future, we want to deploy the system to users and iterate on the design of the interaction flow and UI to balance the efficiency and security.

CONCLUSION

We present Gesture On, a system that enables touchscreen gesture shortcuts from the standby mode. We discussed the design and implementation of two key components: gesture detection from accidental touches and user authentication based on a single gesture. Our experiments indicated that Gesture On demonstrated a useful approach for fast mobile access.

REFERENCES

1. Amini, S. and Li, Y. CrowdLearner: rapidly creating mobile recognizers using crowdsourcing. *Proc. UIST 2013*, ACM Press (2013), 163–172.
2. Appert, C. and Zhai, S. Using strokes as command shortcuts. *Proc. CHI 2009*, 2289–2298.
3. Böhmer, M., Hecht, B., Schöning, J., Krüger, A., and Bauer, G. Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage. *Proc. MobileHCI 2011*, 47–56.
4. Feng, T., Liu, Z., Kwon, K.-A., Shi, W., Carbunar, B., Jiang, Y., and Nguyen, N. Continuous mobile authentication using touchscreen gestures. *2012 IEEE Conference on Technologies for Homeland Security (HST)*, IEEE (2012), 451–456.
5. Ferreira, D., Goncalves, J., Kostakos, V., Barkhuus, L., and Dey, A.K. Contextual experience sampling of mobile application micro-usage. *Proc. MobileHCI 2014*, ACM Press (2014), 91–100.
6. Hayashi, E., Das, S., Amini, S., Hong, J., and Oakley, I. CASA: context-aware scalable authentication. *Proceedings of the Ninth Symposium on Usable Privacy and Security - SOUPS '13*, Article No. 3.
7. Kane, S.K., Wobbrock, J.O., and Ladner, R.E. Usable gestures for blind people : understanding preference and performance. *Proc. CHI 2011*, 413–422.
8. Kennedy, P.R., Hall, T.G., and Hardy, D.A. Security token and method for wireless applications. Patent US6084968, 2000.
9. Li, Y. Protractor : a fast and accurate gesture recognizer. *Proc. CHI 2010*, ACM Press (2010), 2169–2172.
10. Li, Y. Gesture search: a tool for fast mobile data access. *Proc. UIST 2010*, ACM Press (2010), 87–96.
11. Lü, H., Fogarty, J., and Li, Y. Gesture Script: recognizing gestures and their structure using rendering scripts and interactively trained parts. *Proc. CHI 2014*, ACM Press (2014), 1685–1694.
12. Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, I.H.W. The WEKA Data Mining Software: An Update. *SIGKDD Explorations 11*, 1 (2009).
13. Mcgrath, W. and Li, Y. Detecting Tapping Motion on the Side of Mobile Devices By Probabilistically Combining Hand Postures. *Proc. UIST 2014*, 215–219.
14. Miluzzo, E., Varshavsky, A., Balakrishnan, S., and Choudhury, R.R. Tapprints: your finger taps have fingerprints. *Proc. MobiSys 2012*, 323–336.
15. Mochel, P. The sysfs filesystem. *Linux Symposium*, (2005), 313–326.
16. Ouyang, T. and Li, Y. Bootstrapping personal gesture shortcuts with the wisdom of the crowd and handwriting recognition. *Proc. CHI 2012*, 2895–2904.
17. Pirlo, G., Cuccovillo, V., Impedovo, D., and Mignone, P. On-line Signature Verification by Multi-Domain Classification. *Proceedings of International Conference on Frontiers in Handwriting Recognition (ICFHR 2014)*, IEEE (2014), 67–72.
18. Ruiz, J. and Li, Y. DoubleFlip: : a motion gesture delimiter for mobile interaction. *Proc. CHI 2011*, 2717–2720.
19. Schwarz, J., Xiao, R., Mankoff, J., Hudson, S.E., and Harrison, C. Probabilistic palm rejection using spatiotemporal touch features and iterative classification. *Proc. CHI 2014*, 2009–2012.
20. Shahzad, M., Liu, A.X., and Samuel, A. Secure unlocking of mobile touch screen devices by simple gestures. *Proc. MobiCom 2013*, 39–50.
21. Sherman, M., Clark, G., Yang, Y., Sugrim, S., Modig, A., Lindqvist, J., Oulasvirta, A., and Roos, T. User-generated free-form gestures for authentication. *Proc. MobiSys 2014*, 176–189.
22. Wobbrock, J.O., Wilson, A.D., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *Proc. UIST 2007*, 159–168.