Itus: An Implicit Authentication Framework for Android

Hassan Khan, Aaron Atwater, and Urs Hengartner
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON Canada
{h37khan,aatwater,urs.hengartner}@uwaterloo.ca

ABSTRACT

Security and usability issues with pass-locks on mobile devices have prompted researchers to develop implicit authentication (IA) schemes, which continuously and transparently authenticate users using behavioural biometrics. Contemporary IA schemes proposed by the research community are challenging to deploy, and there is a need for a framework that supports: different behavioural classifiers, given that different apps have different requirements; app developers using IA without becoming domain experts; and real-time classification on resource-constrained mobile devices. We present Itus, an IA framework for Android that allows the research community to improve IA schemes incrementally, while allowing app developers to adopt these improvements at their own pace.

We describe the Itus framework and how it provides: **ease of use**: Itus allows app developers to use IA by changing as few as two lines of their existing code—on the other hand, Itus provides an oracle capable of making advanced recommendations should developers wish to fine-tune the classifiers; **flexibility**: developers can deploy Itus in an application-specific manner, adapting to their unique needs; **extensibility**: researchers can contribute new behavioural features and classifiers without worrying about deployment particulars; **low performance overhead**: Itus operates with minimal performance overhead, allowing app developers to deploy it without compromising end-user experience. These goals are accomplished with an API allowing individual stakeholders to incrementally improve Itus without re-engineering new systems. We implement Itus in two demo apps and measure its performance impact. To our knowledge, Itus is the first open-source extensible IA framework for Android that can be deployed off-the-shelf.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Authentication

MobiCom'14, September 7-11, 2014, Maui, Hawaii, USA.

ACM 978-1-4503-2783-1/14/09 ...$15.00.
http://dx.doi.org/10.1145/2639108.2639141 .

Keywords

Security; Implicit Authentication; Behavioural Biometrics

1. INTRODUCTION

Smartphones contain a wealth of personal data including banking information, contacts, photos, videos, texts and emails. Smartphones that are used as bring your own devices (BYODs) may also contain confidential corporate data. In order to protect confidential data on smartphones from unauthorized access, primary authentication mechanisms (including PINs, pass-locks, draw-a-secret, fingerprint- or facial-recognition systems) are used. Organizations also use mobile device management (MDM) solutions to enforce password policies on BYODs to protect corporate data.

However, a recent survey found that 56% of smartphone owners do not configure PINs or pass-locks on their devices [24]. Since smartphone sessions tend to be short and frequent, all-or-nothing pass-locks reduce usability [4, 19]. While MDM solutions can enforce password policies, they aggravate the usability issues. According to a prediction by Gartner, 20% of enterprise BYOD programs will fail by 2016 due to overly restrictive policies of MDMs [36]. In addition to usability issues, pass-locks have also been subject to operating system flaws [39], smudge attacks [3] and shoulder surfing attacks. The limitedly available fingerprint- and facial-recognition systems have also been shown to be vulnerable to attacks [41].

These limitations have prompted researchers to develop implicit authentication (IA) schemes as a second line of defense to mitigate security and usability issues with explicit authentication mechanisms (such as pass-locks) [9]. IA schemes for smartphones authenticate a user by using distinctive, measurable patterns of device use that are gathered from the device users without requiring deliberate actions [9]. These IA schemes allow enterprises to relax pass-lock policies and complement them with IA. In case an adversary bypasses a pass-lock, IA may be able to detect it and lock the device or alert the user via email. Similarly, smartphone owners who do not wish to use pass-locks due to inconvenience can use IA to get limited protection from unauthorized use. Despite this utility and the reasonable performance of contemporary IA schemes, they are not deployed in practice. The lack of IA adoption is due to the challenges involved in creating an IA framework that is flexible enough to be used by a general audience. To date, there is no publicly available implementation of such a framework addressing these challenges.

Since contemporary IA schemes require the interception of user input events, some researchers have proposed including IA mechanisms at the platform level [14, 23], such that the operating system or app framework is responsible for providing IA to all apps on the system in an app independent manner. However, this approach has its own limitations in terms of flexibility and extensibility. First, different apps have different characteristics and a generic platform-level behaviour-based classifier may not be suitable for different apps. For example, a classifier based on keystroke behaviour may not be suitable while using a banking app where the user interacts through swipes. A classifier that constantly monitors all possible kinds of input would be problematic in terms of performance and battery life. The difference in app characteristics can also severely reduce the accuracy of IA schemes by as much as 20% [11, 22]. Furthermore, IA performed at the platform level would not be able to distinguish which activities within an app require protection by an authentication mechanism [22]. For example, the banking app may have a "locate ATM" feature that anyone using the device would be permitted to use (without the annoyance of triggering an IA rejection). Secondly, platform-level IA mechanisms would need to be managed by the platform developers or some central authority. However, IA is a relatively new area that still experiences radical revisions due to the research findings in such areas as the use of novel sensors or wearable devices for IA [28, 35], and the research findings on the usability and acceptance of IA schemes. The platform developers in this case are more inclined toward accepting mature contributions, which will lock out many developers.

These usability and flexibility limitations can be circumvented by enabling the apps themselves (that require IA) to implicitly authenticate users. By delegating IA to apps, we provide: usability—an app would only require authentication of a user while he is using the app, and since the app is in the foreground, it has access to event data and can launch an activity to lock the functionality in case of misuse—and flexibility: app developers can choose the behavioural classifier that is suitable for their app. However, the app developer requires domain-specific knowledge (e.g., of suitable behavioural classifiers, underlying machine learning algorithms, and their parameterization) and significant effort to provide IA support. An IA library could ease the burden of the app developers by abstracting away most of the details with a convenient API.

The challenges to creating such a framework include making the resulting library easy to use for app developers who are interested in providing IA to their users without the need to become domain experts. On the other hand, it should also be flexible enough to cater for app-specific functionality. Moreover, it must be extensible by the researchers developing novel IA schemes. Finally, it should not have a noticeable impact on performance or battery life to end-users, especially given the computational constraints of smartphones.

We present Itus[1], a framework that separates the domain knowledge of IA from its deployment. Itus has been designed to enable app developers to effortlessly provide IA support in their apps by modifying as few as two lines of code. At the same time, we provide app developers with an oracle that bridges the domain knowledge gap by forming recommendations on optimal sets of behavioural features and classifiers tailored for their particular apps. Itus is designed with extensibility in mind, allowing IA developers to iteratively contribute improvements to the framework. To demonstrate this we port three existing IA schemes to Itus. We demonstrate ease of use by adding Itus to two demo apps and we perform empirical evaluations to demonstrate Itus' low performance overhead. We hope that Itus will enable the research community to collaborate better to further the research in the IA domain.

The main contributions of this paper include:

1. We provide[2] the first IA framework for Android that can be used by app developers with minimal effort. It is implemented as a library at the user level and thus can be used right away without the need to root any devices or be added to the OS, although it could become part of mobile platforms in the future.

2. We provide researchers with an open-source, fully implemented system for rapid prototyping and deployment of new IA schemes. This is exemplified by the fact that we are able to implement three prominent IA schemes [5, 12, 14] using Itus.

3. We provide the Itus Oracle, a tool for app developers to automatically determine appropriate behavioural classifiers, suitable sets of features and optimum operating points of various configuration parameters for their apps. The Oracle also provides a way for us to curate the availability of IA schemes without hindering the ability for other developers and researchers to contribute to Itus.

4. We show that flexible and adaptable IA is possible on smartphones with acceptable performance overhead.

2. MOTIVATION

We consider our audience to be two-fold: the app developers who wish to incorporate IA into their apps, and the IA developers who wish to improve existing IA schemes. In this section, we first discuss some sample apps and the motivations their developers might have for including IA support. We then discuss the community of app developers that Itus targets in § 2.1, and the community of IA developers in § 2.2.

We consider the following apps as potential candidates for using IA:

An Enterprise Email Client: Provides employees access to corporate emails from their mobile devices.

A Web Browser: Provides all the standard web browser features including a password manager.

The email client app gives access to large amounts of sensitive corporate data. If employees were to install the app on their personal devices, the risk of information leakage is significantly increased. Adding IA at the app level allows the employer to ensure only authorized users are accessing sensitive data without disincentivizing users from installing the app on their own devices due to inconvenient security mechanisms. The developers of the browser, on the other hand, might be interested in providing their users the convenience of a password manager with the increased security of IA. The web browser could be used by anyone for non-sensitive sites

[1]Itus, Greek god of protection

[2]`https://crysp.uwaterloo.ca/software/itus/`

(e.g., lending the device to a friend to check the weather). When a site with a stored password is accessed, the password manager might then hand over a password only if IA decides that the user's recent activity pattern is as expected. Otherwise, the password manager will explicitly ask for the user's master password.

2.1 App Developers

Next, we refine our model of the spectrum of involvement by the app developers adding support for IA. In particular, we consider two levels of developer interest in IA frameworks:

Cursory interest: app developers who only want to add support for IA to their apps without tuning its accuracy or providing any application-specific behaviour. There are several types of app developers that may fall into this category. In the most trivial case, an app developer may simply be interested in experimenting with IA and wants to add it to their app without spending significant time on configuration and re-engineering of their app. Or, the app developer may be developing an app that contains such generic tasks that the default behaviour is "good enough" out-of-the-box. For example, consider the email client described above—user typing cadence tends to be unique enough [12] that the default keystroke classifier can easily deal with most usage scenarios of this app.

Significant interest: app developers who wish to fine-tune Itus for accuracy and performance. Such developers might do one or more of the following: restricting calculation of behavioural features to some subset to minimize computational overhead; configuring parameters of machine learning algorithms; experimenting with beta users to determine which configurations work best and retaining the data gathered during this phase for later training. It is important that Itus be able not only to support these developers in their efforts but to facilitate them by providing tools to assist and automate in these scenarios.

2.2 IA Developers

We now discuss the other class of audience— the developers who wish to contribute to Itus' IA schemes. To this end, we consider two representative scenarios.

Consider first those developers who wish to improve existing IA schemes (for example, investigating a novel set of behavioural features or simply tuning parameters). To demonstrate the efficacy of their proposal, they need to develop a prototype and evaluate its performance. Ideally, they should have access to an existing framework, giving them the necessary components to perform tasks common to all IA schemes (e.g., data storage, training, classification) without re-engineering. Itus should allow these developers to rapidly prototype their ideas without having their contributions moderated.

Secondly, consider developers who want to add support for behavioural classification modules that the core Itus framework does not anticipate. Specifically, the framework should allow for new machine learning classification algorithms and new sensor-derived behavioural features to be included alongside the core framework.

Itus should support these stakeholders and provide them with interfaces so that they can use or contribute to the framework as smoothly as possible.

3. DESIGN GOALS

In the previous section we highlighted the two-fold audience we consider while developing the Itus framework: app developers and IA developers. Both groups of developers are important for a successful IA framework to be adopted and remain relevant over time. In this section, we outline our design goals for Itus' framework for IA, and how they support both groups of developers in their efforts to provide usable IA to end users.

Separation of roles: Performing end-user authentication in a secure, usable manner is a challenging task that involves collaboration between IA developers and app developers. Our IA framework should synergize the efforts of different stakeholders to protect user data on mobile devices. In this spirit, we aim to incorporate a clear separation of roles that distinguishes between app developers, who may not be domain experts in the mechanisms underlying the system, from the IA developers or researchers working to improve the system.

Ease of Use: Our framework should not simply permit app developers to provide IA support in their apps, but also provide the app developer with an API to do so with minimal effort. App developers should be rewarded for their extra attention to user security, not punished with a burden of extended development time. Our goal is for Itus to be deployable in a reasonable default configuration with the absolute minimum number of lines of code changed as possible. The API should also allow configuration directives to be provided in as straightforward a manner as possible.

Flexibility: While ease of use is critical for adoption of IA by the broader audience of app developers, it is also important to make considerations for developers who have unique app needs or who wish to fine-tune the accuracy or performance of the IA scheme to their app. For example, for the web browser example discussed in § 2, the developer might be interested in making authentication decisions in a heavily context-aware manner. To this end, he might decide that after users start playing an embedded video, they are likely to hand off the device to another person and thus it would be undesirable for the IA mechanism to interrupt viewing with an explicit authentication prompt. In the case of the enterprise email client, the company's IT department may wish to invest heavily to provide high accuracy to ensure data security. In this case, the app developers should be able to configure Itus with pre-recorded training data or parametrize the classifier to improve its accuracy. We consider design features that allow for such examples of app-specific functionality when providing flexibility to app developers.

Extensibility: Extensibility is a prerequisite design goal of any system that is to be adopted by the community. Individual IA developers should be able to create and evaluate prototypes for various subcomponents of the Itus framework without any dependence on a centralized authority, and they should be able to distribute successful IA schemes independently from the core Itus framework. Ideally, this leads to a situation in which IA developers are able to deploy iterative improvements to the IA schemes while simultaneously allowing app developers to adopt these improvements at their own, perhaps asymmetric, rate. Our framework should allow each of these groups to make contributions without becoming embroiled in the specifics of framework subcomponents unrelated to their particular tasks.

Performance: Finally, performance in terms of computational overhead and power consumption is important to both the app developers and the researchers working on IA. App developers are especially concerned with any performance penalty that is going to be imposed on their apps, as this has a direct effect on their end-users' experience. Therefore, it is important that the framework itself consumes minimal resources so as to allow the behavioural classification subcomponents as much time as possible to perform their tasks before the end user is able to discern any differences.

4. ARCHITECTURE

This section gives an overview of the Itus system architecture. The core architecture of Itus, illustrated in Figure 1, is as follows: app developers extend a customized Android activity called `SecureActivity`, provided by the Itus library. Itus is then able to intercept user interaction events, which are passed through feature extractors called `Measurement`s to obtain feature vectors. The Itus Agent gathers these feature vectors and hands them to the classification algorithms for training and classification. In the event a classifier returns a negative result (failed authentication), the Itus Agent then either notifies the parent app of the failure or independently switches the app view to a lock screen. In what follows, we will elaborate on how each of these links in the chain fit into the Itus framework.

4.1 SecureActivity

Interactive Android apps are composed of one or more "activities", which are app components that present the device user with a user interface to the app's functionality. When a developer creates a new Android app, one of the first steps is to create a subclass of Android's `Activity` class. Itus provides its own subclass of `Activity` called `SecureActivity`. To add IA to an app using Itus, a developer simply changes any classes that extend `Activity` to extend `SecureActivity` instead. This provides an incredibly simple way for developers to add Itus to their existing apps, supporting our ease-of-use goal discussed in § 3. It also makes it trivial for developers to partition apps composed of multiple activities into those that require authentication and those that do not (for example, a banking app may not want to authenticate a person when he is locating a nearby ATM); this supports our flexibility goal. This implementation puts our framework in an ideal place to intercept user interaction events with the app in order to then use them as behavioural features for classification.

4.2 Measurement and subclasses

Behavioural biometrics research examines a broad array of sensor values that may be useful for distinguishing between different device users. Examples include touch-screen input [5, 10, 11, 14, 23, 33], keystrokes on a physical or rendered keypad [8, 12, 20, 27, 40], and readings taken from an on-board accelerometer [13, 15]. In the Itus framework, we generalize these widely varying channels to say that behavioural features are generally calculated using some form of *measurements* taken from some on-board sensor. Thus we provide an abstract class in Itus called `Measurement`, subclasses of which are intended to extract measurements from any source accessible within the Android API (or extensions). To further our examples of sensor values, Itus might contain subclasses of `Measurement` called `Touch`, `Keystroke`, and `Movement`, respectively.

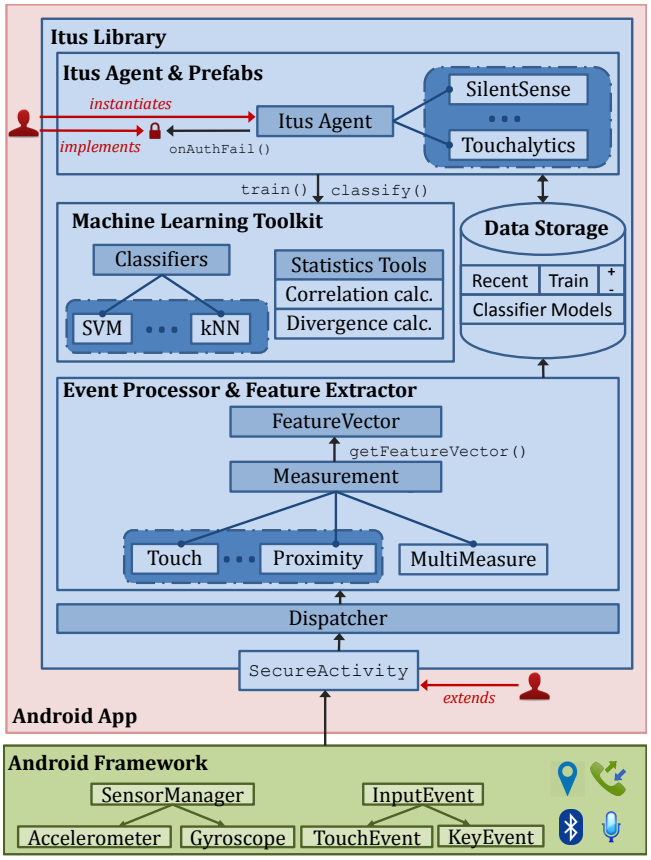


Figure 1: Itus framework architecture. Subclasses that are intended to be contributed by IA developers are in dark blue dotted boxes. The interfaces exposed to app developers are in red font.

`Measurement` objects can register to receive events allowing them to process input data, which are subsequently passed to them via a callback method called `procEvent()`. There are two types of sensor readings: event-driven inputs and continuous readings. Event-driven inputs are intuitive to handle; registered handlers are called to process an event as soon as it occurs. The touch and keystroke examples are cases of event-driven inputs. Continuous readings, in which a sensor provides measurements any time it is polled (e.g., the accelerometer), are more subtle to handle. To deal with these, we create `Periodic` events, which function similarly to clock timer interrupts. `Measurement` objects analyzing continuous feedback sensors are then invoked periodically at parameterizable intervals. Some `Measurement`s may provide feature values calculated from an aggregate of sensor readings and not at every regular time interval; for these objects, we allow the event-processing function to simply consume data and indicate that it is not yet ready to provide any feature values.

4.3 Dispatcher

In the last subsection, we said that events are given to various `Measurement` subclasses for feature extraction. In

keeping with our performance goals outlined in § 3, it would not be efficient to invoke the `procEvent` method of every single `Measurement` for every single event. Instead, we create a `Dispatcher` class, which is responsible for delegating event processing to any configured `Measurement` objects. When Itus is configured at run-time to use a given `Measurement` subclass, an initialisation method is run from within that class. The primary purpose of initialisation method is to register with the `Dispatcher` and specify which events the `Measurement` would like to receive. Any events handled by Itus—either intercepted from user input by `SecureActivity` or generated periodically—are thus given to the `Dispatcher`, which in turn looks up the event type in a table to see which `Measurement`s have registered for it, optionally adds context information, and passes them the event data via `procEvent()`.

4.4 FeatureVector

As discussed previously, `Measurement` subclasses are responsible for taking data from events and turning them into useful feature values. When `procEvent()` is called to deliver event data to a `Measurement` object, the `Measurement` object will signal the `Dispatcher` if it has a new set of feature values ready to be exported. After receiving the signal, the `Dispatcher` will invoke the `getFeatureVector()` method of the `Measurement` subclass to retrieve these values. Specifically, the feature values are stored in a class called `FeatureVector`. This class is a 2-tuple made up of an array of `double` values and a boolean representing the class of the feature vector (valid/invalid user).

4.5 DataStorage

A `FeatureVector`, as obtained above, is processed as follows: if training has already been performed, then we simply classify this most recent sample. If training has yet to be performed, the `FeatureVector` should be stored until enough data is collected to perform training. In both cases, the `Dispatcher` will hand the `FeatureVector` over to a `DataStorage` object, which will store the `FeatureVector` in a list (hereafter called a "bin") depending upon the classifier state. In the former case, when training is complete, the `FeatureVector` will be placed into a bin labeled **bin_recent**, which can have an upper limit imposed on its size; this results in a sliding window, or FIFO queue, in which older data is discarded after consumption by the classification module or upon arrival of newer data. In the latter case, when training is pending, the `FeatureVector` is placed in the longer-term **bin_training** to be used for training once the bin reaches sufficient size.

The `DataStorage` object is also used to store classifier models once they have been trained. It has `get()` and `put()` methods allowing arbitrary data to be stored, allowing IA developers to use it as a convenient storage mechanism via the app's internal storage.

4.6 Itus Agent

Now that we have discussed the basic modules of our framework, we provide details of how the actual invocation of training and classification are coordinated. These are done by a class called `Itus`, which we refer to as the 'Itus Agent'. The Itus Agent runs in a separate thread from the main app. It is the main object that an app developer will interact with when they are adding the Itus framework to their app and configuring it. An instance of the Itus Agent class performs the configuration for all subcomponents of the Itus framework, including which `Measurement` subclasses are used and which classifiers are employed (see § 4.8). The Itus Agent drives the periodic events described in § 4.2, enforces the authentication policy with training/classification and, where necessary, locks the app in the event of a failure to implicitly authenticate.

In more detail, once training has been completed and a classifier has been obtained, the Itus Agent then simply runs it periodically against the recent `FeatureVector`s stored in **bin_recent**. If a classifier returns a negative result representing an unauthorized user, the Itus Agent reacts by performing its configured (or default) lock-screen action. If training has yet to be completed, the Agent explicitly authenticates the user in order to establish the ground truth for training data.

4.7 Itus Oracle

While we separate the domain knowledge of IA from its deployment by providing app developers with a high-level API to provide IA support in their apps, we understand the importance of correct selection of behavioural classifiers and suitable parametrization of the underlying machine learning algorithms. To bridge the gap between app developers' inexperience with IA schemes and the need for advanced configurations, we provide the Itus Oracle to automatically determine the right classifier and optimal configuration parameters.

To use the Itus Oracle, the app developer deploys Itus in 'configuration mode'. In this mode, Itus collects and logs all feature vectors and gathers other measurements concerning the data collection and performance of the app, such as timestamps, data sources, and processing times. After multiple sessions of beta use in the configuration mode, the developer then connects the device to a development computer and runs the Itus Oracle. The Itus Oracle downloads the logged data from the device and analyzes it. It experiments with various machine learning tools, including any classification algorithms compatible with Itus, and provides suggestions based on its analysis to the app developer. Finally, it repackages the data and configurations as a standalone Itus library that the developers can then import directly to their app. Repackaging also enables the Itus Oracle to bundle only the required Itus Prefabs which reduces the runtime memory footprint of the Itus library. More information about the implementation of the Itus Oracle is provided in § 6.

4.8 Machine Learning Toolkit

The number of viable candidates for machine learning algorithms for classifying behavioural feature sets is continually expanding. This is due not only to the ongoing research in the field but also due to the fact that a classifier may work well in combination with a certain feature set, while other classifiers may perform much better in cases where it is weak. Therefore, it is highly desirable that the Itus framework be able to add support for many classification algorithms without the development overhead of coupling them tightly with the rest of the IA framework.

Itus does this by defining a `Classifier` interface with two self-documenting methods: `train(List<FeatureVector>)` and `classify(FeatureVector)`. The `classify()` method

should return `boolean` value `true` for the positive class (authorized user) and `false` for the negative class (unauthorized user). More fine-grained control, such as interacting with confidence values, can be obtained by accessing `Classifier` objects directly. The learned classifier is transparently loaded to and from disk between app launches via the `DataStorage` object. We initially provide Itus with an implementation of the k-nearest-neighbours (kNN) classifier, and a wrapper that allows the Java version of libSVM (Support Vector Machines) [6] to be used as `Classifier`s.

Finally, the machine learning toolkit also contains statistics tools to measure correlation and divergence between features to determine effectiveness of features of behavioural classifiers. The Itus Oracle uses these tools to generate recommendations on features which should be used for an app.

5. WORKFLOW

In this section we outline the workflow that app developers must follow to provide IA support in their apps using Itus. We also discuss how IA developers can benefit from Itus and in return how they can contribute.

5.1 App Developers

As discussed in § 2.1, we broadly consider two types of app developers: those who want to use IA out-of-the-box, and those who want to tune the accuracy or performance of Itus. For the former type, Itus can be effortlessly imported and used in their apps. The app developer simply (i) identifies activities that should be protected, (ii) extends the activities from Itus' `SecureActivity` class, and (iii) starts the main thread of an Itus object. Training and classification are performed automatically at this point (see § 6 for details).

On the other hand, if the developer wants to tune the performance or the accuracy of a classifier, he uses the Itus Oracle to determine the best configurations for his app. To this end, the app developer (i) identifies activities that require implicit authentication and extends them from `SecureActivity`, (ii) starts Itus in configuration mode and gives device(s) to beta users for data collection, (iii) connects the device to the Itus Oracle (on the desktop) so that it can analyze data and generate recommendations, (iv) chooses from the recommended configurations so that the Itus Oracle can repackage the library, and (v) adds the repackaged library to the app and disables configuration mode.

Most classifiers require negative (non-owner) training data in order to provide high accuracy. For this, an app can either use default negative instances included with Itus, or it may package a subset of data collected during configuration mode as negative training data. Since data collected during the configuration mode is labeled against each user, the negative training set of a user is created by excluding his data. We anticipate that after Itus is released in the public domain, it will also benefit from IA datasets that have been made publicly available by the research community [13, 14, 31, 37].

5.2 IA Developers

§ 2.2 presented two types of IA developers we envision contributing to Itus. Itus provides deliberate separation for contributions to the IA mechanism. In § 4.2 we described how new sensor features are added to Itus by extending the `Measurement` class. Similarly, § 4.8 described how new classification algorithms are added to Itus be extending the `Classifier` class. These subclasses can be contributed to Itus independently by the second group of IA developers discussed in § 2.2.

§ 4.6 explained how app-specific configurations are stored in an `Itus` object. Itus can provide for a variety of default configurations by simply distributing pre-built `Itus` objects. In the Itus framework, we call such objects Prefabs and implement them by deriving subclasses of `Itus`. These subclasses inherit all the functionality of the default Itus Agent, and simply need to add a constructor that performs any configuration directives that would normally be added by the app developer. These Prefabs can be compiled and distributed separately from the Itus framework, allowing the first group of developers discussed in § 2.2 to propose new configurations or even modified behaviour of the Itus Agent without needing their contribution to be accepted upstream by the core Itus framework.

6. IMPLEMENTATION

We implemented the Itus framework in Android Java and the Itus oracle in Python. Itus is currently about 22,000 lines and has been released in open source. Itus is distributed as a standalone library of 773`KB` and it can be imported and used by any Android project that supports Android version 2.23 and above. This allows Itus to support the majority of the Android devices in use today (98.8% Android devices in-the-wild as of March, 2014 [16]). In this section, we discuss some of the significant implementation decisions of Itus.

Event Interception: The API of Itus has been designed to abstract away the underlying low-level event collection from the app developer. While there exist mobile sensor collection frameworks such as SoundSense [25] and Jigsaw [26], unfortunately these frameworks exist for Nokia and Apple iOS and not for the Android OS. For event handling in Android Activities, Android provides `EventListener` and `EventHandler` to the app developers. Every user activity in an app is derived from the Android `Activity` class and these events are first delivered to `Activity.dispatchTouchEvent()`. We provide the `SecureActivity` class, which extends the `Activity` class and additionally provides constructs to optionally intercept and copy events before delivering them to the users. App developers who want to provide IA support in their apps are expected to extend `SecureActivity` for transparent event interception.

Data Storage: For permanent storage, Itus uses an app's internal storage. We chose internal storage since: an app's internal storage can only be accessed by the app itself and a malicious app cannot gain access to the training model; and accessing an app's internal storage space does not require any explicit permissions. We require the training model classes be `Serializeable` so they can be written and subsequently read from the permanent storage automatically.

Training Set Size and Retraining: To perform training when the user first interacts with a new app, a sufficient amount of data must be collected. The definition of 'sufficient' here has some room for interpretation, and so we provide two distinct ways for developers to specify the training data threshold: absolutely and empirically. In the absolute case, the developer sets the minimum threshold to some integer value (N), and when **bin_training** receives N samples of `FeatureVector`s, training is triggered. In the empirical case, the developer specifies some minimum accuracy level

to be attained during training before considering training to be complete. Here, the Itus Agent runs training periodically and evaluates accuracy using x-fold cross-validation (where x is another parameter) and stops when it achieves the desired target accuracy. This second method is obviously significantly more performance-heavy than simply training at N instances, so the oracle tool in § 4.7 helps developers determine appropriate values of N.

In order to improve the user experience by reducing false rejects, Itus provides app developers with the option to automatically retrain the classifier to improve its accuracy. To this end, Itus first temporarily stores the `FeatureVectors` that are classified as non-owner's by the behavioural classifier. It then triggers the lockout activity to explicitly authenticate the user. If the user successfully authenticates, Itus uses the misclassified feature vectors to retrain the behaviour-based classifier. While retraining improves the user experience, in addition to training overhead, it requires additional storage space to save misclassified `FeatureVectors`.

Lockout Action: In case of authentication failure, the app developer may launch the device's default authentication mechanism (PINs/pass-locks). However, a large number of device owners do not configure pass-locks on their devices and furthermore, if the attacker has gained access to the device, he has already compromised the primary authentication mechanism. Therefore, to support our off-the-shelf design goal, Itus provides app developers with a default `PasswordConfigure` activity, which is displayed when the app is launched for the first time to configure a password that should be used for explicit authentication in case the IA scheme detects misuse. Itus also provides a `LockoutActivity` to lock the app when misuse is detected. The `LockoutActivity` is also used during the training phase to establish the ground truth during data collection. The `LockoutActivity` overrides the `onBackPressed` method of `Activity` to ignore in-app navigation attempts.

In addition to this, we provide the app developer with a more flexible option to deal with authentication failures. The app developer registers a callback object, implementing our `AuthFailedListenter` interface. This interface specifies a single `onAuthFailed` method, which will be called whenever a classifier fails to implicitly authenticate the user. The app developer is then able to handle authentication failure in any manner desired, supporting our flexibility goal from § 3. For example, a browser might choose to delete the session cookies for any websites the user is currently logged in to.

Managing Timeouts: Itus provides app developers with the ability to pause and resume IA to reduce performance overheads and to save battery life. Itus also provides app developers with the functions to configure timeout intervals so that once a user is successfully authenticated, Itus pauses feature collection and training for the specified interval and resumes its operations when interval times out. App developer can also specify whether they want to reset timeouts and resume IA in case of a screen-off event.

Multi-measurements and Multimodal Classifiers: For advanced IA scenarios, we enable the app developers to use multi-measurements by employing feature samples from different measurement modules. For example, the SilentSense classifier [5] uses events from touch input and data from the motion sensor. Itus provides constructs that can

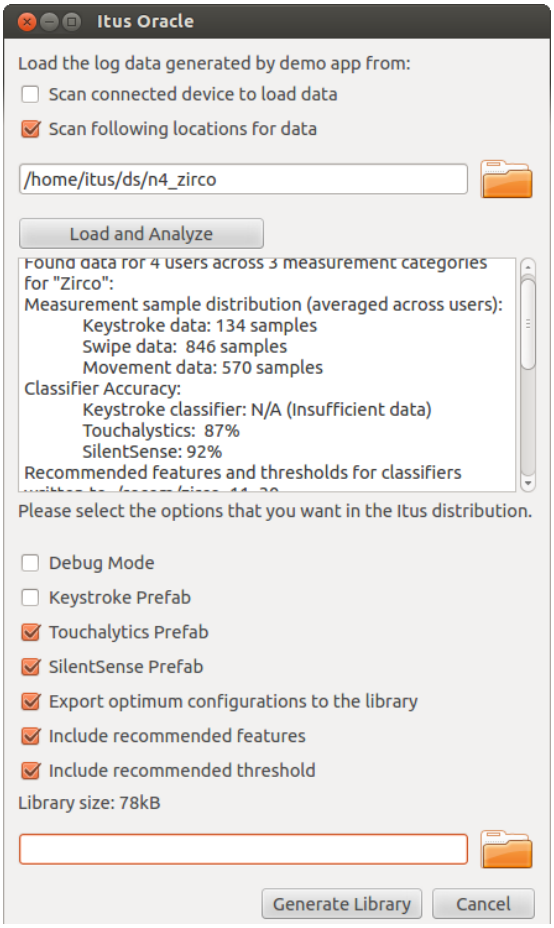


Figure 2: Itus Oracle utility

be used by the app developer to define relationships between measurement data from different sources (`MultiMeasurement` in Figure 1). The app developer simply defines the causal relationship between two events and the resultant `FeatureVector` is automatically generated. Similarly, Itus provides high-level constructs to the app-developer to use multiple behavioural classifiers. These constructs can be used by the developers to combine the authentication score from different behavioural classifiers in a pre-condition or majority score setting. For example, the enterprise email client might use a location-based classifier as a pre-condition to trigger a touch-based classifier.

Itus Oracle: The Itus Oracle identifies suitable classifiers, optimal feature sets, and operating threshold recommendations for the app developer. It determines an appropriate classifier by evaluating the accuracy of Itus Prefabs on the data collected during the configuration mode. This decision also takes into account the availability of data for different Itus Prefabs. The optimal feature set for a particular app is determined by calculating information gain for each feature on collected data. For example, for a navigation app, Itus would automatically detect that the direction of a swipe is not a good feature (due to its high variance on sampled data). Finally, the Itus Oracle provides the app developer with the optimum threshold values by determining the operating point with the highest accuracy.

We provide a screenshot of the Itus oracle in Figure 2. The summary and recommendations generated by the Itus Oracle are shown in the text box. For now, this information (such as confidence intervals and cross-validation results) is simply displayed in raw form to the developer, which may require some statistics knowledge to interpret. In future iterations, we aim to parse these results for more concise recommendations. Based on the recommendations, the Itus Oracle also generates an Itus library that the app developer can readily use in his app.

Prefabs Selection: In the future, we envision using the Oracle as a method of curating the implementations of IA presented to app developers. This will allow us to prevent malicious schemes from entering the ecosystem, such as an IA developer creating a keystroke `Measurement` that also functions as a keylogger. However, as running the Oracle or accepting its recommendations are not necessary steps in order to use Itus, this approach allows for researchers to freely experiment with IA, and for IA developers to distribute standalone extensions to Itus. For now, we want to provide a diverse set of prefabs with Itus to demonstrate its extensibility. To this end, we choose a keystroke classifier [12], a classifier based on touch behaviour (Touchalytics) [14] and a classifier based on the micro-movements of the mobile device caused by touch (SilentSense) [5]. The keystroke classifier and Touchalytics only use user generated events (`KeyEvent` and `TouchEvent`, respectively) while SilentSense uses data that merges user generated events (`TouchEvent`) with periodic events (accelerometer data). Furthermore, the keystroke classifier and Touchalytics use kNN for classification while SilentSense employs SVM for classification.

```
122 public class MainActivity extends SecureActivity
123     implements IToolbarsContainer, OnTouchListener,
124     IDownloadEventsListener {
125
126     @Override
127     public void onCreate(Bundle savedInstanceState) {
128         super.onCreate(savedInstanceState);
129         INSTANCE = this;
130         /*create an Instance of Touchalytics classifier
131          * & start the Itus thread*/
132         (new Touchalytics()).start();
```

(a) Using a Prefab to provide IA support

```
126     @Override
127     public void onCreate(Bundle savedInstanceState) {
128         super.onCreate(savedInstanceState);
129         INSTANCE = this;
130
131         /*Configure & launch Itus*/
132         Itus itus = new Itus(this);
133         String [] featureList = {"Start X", "Start Y",
134                 "Average Velocity", "Interstroke Time"};
135         Classifier svm = new SVMClassifier(
136             featureList.length, "negative_instances");
137         svm.setFeatureScaling(true);
138
139         Measurement touch = new Touch();
140         touch.setFeatureList(featureList);
141         itus.useMeasurement(touch);
142         itus.setTrainingSize(100);
143         itus.useClassifier(svm);
144         itus.start();
```

(b) Using low-level constructs to provide IA support

Figure 3: Itus' development overhead for Zirco Browser

7. PERFORMANCE EVALUATION

To provide a seamless user experience, it is critical for Itus to have minimum performance overhead. Furthermore, since smartphones are power constrained devices, high battery consumption of Itus may reduce its utility. In this section, we first discuss the experimental setup that we use for performance evaluation and we then provide the results of our evaluation.

7.1 Experimental Setup

Device Selection: For performance evaluations, we use an HTC Nexus 1 and an LG Nexus 4. The HTC Nexus 1 has Android OS v2.23 (`Gingerbread`) on a 1GHz processor with 512MB of RAM. The LG Nexus 4 has Android OS v4.4 (`KitKat`) on a quad-core 1.5 GHz processor with 2GB of RAM. Our selection of these diverse devices supplies an overview of Itus performance on both old and recent hardware.

Performance Metrics For empirical evaluations, we are only interested in the performance of Itus and we do not evaluate the accuracy of the IA schemes employed. We refer interested readers to the original papers [5, 12, 14] for the accuracy evaluations and default parameters used here (such as window sizes and sampling rates). For performance evaluations, we measure the performance overhead in terms of elapsed CPU time and heap size of the app. We also evaluate the battery consumption overhead of Itus. Finally, we evaluate the impact of Itus on user experience by measuring the relative performance overhead of Itus with our demo apps.

Demo Apps For demo apps to evaluate for performance purposes, we choose: (i) Zirco Browser[3]: an open-source browser with between 50,000 and 100,000 installs at the Google Play Store; and (ii) TextSecure[4]: a popular open-source encrypted communication app with between 100,000 and 500,000 installs at the Google Play Store. These apps were selected as our demo apps for two reasons: (i) both apps manage private data of the user, and (ii) usage of these apps results in different event types (`TouchEvent` for Zirco and `KeyEvent` for TextSecure), which allows us to test the different classifiers discussed in § 6.

7.2 Evaluation Results

Development Overhead: For a developer who is employing Itus Prefabs, the development overhead is minimal. While quantification of development overhead is a non-trivial task, we herein provide our experience of adding Itus to the demo apps. To avoid any bias due to the absence of a learning curve, we only report development overhead in terms of the number of lines of code added/modified. To provide default IA support in Zirco Browser and TextSecure, in addition to importing the Prefab class, we only modified 2 lines of code. As discussed in § 5, the app developer extends the `SecureActivity` class and launches a suitable Prefab (both operations are highlighted in Figure 3a).

However, if the app developer wants to optimize the classifiers, there would be more development overhead (depending

[3]`https://play.google.com/store/apps/details?id=org.zirco`

[4]`https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms`

		Initialization		Feature Extraction		Training		Classification	
		CPU(ms)	Heap(kB)	CPU(ms)	Heap(kB)	CPU(ms)	Heap(kB)	CPU(ms)	Heap(kB)
N1	Keystroke	21 (2.08)	185(5.6)	<1 (≃0)	<1 (≃0)	<1 (≃0)	<1 (≃0)	0.2 (≃0)	3.1 (0.19)
	Touchalytics	5 (0.27)	17.3 (2.6)	0.27 (≃0)	7.8 (0.03)	65 (2.16)	48.9 (1.4)	1.7 (≃0)	51.3 (1.4)
	SilentSense	1162 (81)	1236 (15)	0.75 (≃0)	14.6 (1.1)	10384 (91)	2472 (18)	0.12 (≃0)	3.7 (0.04)
N4	Keystroke	12 (0.95)	186 (2.3)	<1 (≃0)	<1 (≃0)	<1 (≃0)	<1 (≃0)	0.05 (≃0)	2.9 (0.13)
	Touchalytics	3 (0.27)	16 (0.62)	0.05 (≃0)	11 (0.48)	15 (0.5)	53 (0.74)	1.08 (≃0)	56 (5.11)
	SilentSense	972 (67)	776 (34)	0.55 (≃0)	16.8 (0.4)	5937 (329)	2453 (39)	0.07 (≃0)	4.2 (0.28)

Table 1: Performance evaluation of different configurations of Itus. 95% confidence intervals are provided in parenthesis. N1: Nexus 1 and N4: Nexus 4.

on the type of optimizations). The app developer can use the Itus Oracle to perform optimizations automatically or choose to manually perform optimizations in order to control certain aspects of IA scheme. In Figure 3b, we show one of the possible workflows an app developer might follow to manually employ touch behaviour features for IA using the SVM classifier. The 10 lines of code (Line 132 - 144) in Figure 3b show how the developer: (i) instantiates Itus (Line 132); (ii) defines a list of feature that should be used by the `Measurement` module (Line 133, 139, 140); (iii) configures parameters of the SVM classifier (Line 135 - 137); and (iv) configures Itus to use the `Measurement` and `Classifier` objects, and starts the Itus Agent (Line 141 - 144).

Performance Evaluation of Itus: We instrument Itus to measure its performance overhead in terms of elapsed CPU time and size of heap memory for different runtime configurations. Since the operations of Itus depend on events that cannot be accurately controlled manually, for repeatable experiments we use Monkey scripts [2] to simulate event generation. We repeat each experiment 15 times for three different runtime configurations and report averages in Table 1. Table 1 shows that both the keystroke classifier and Touchalytics require much less CPU time and heap memory. More specifically, the feature extraction and classification operations that are triggered for every input event, take under 1 and under 2 ms for the keystroke and touch classifiers, respectively on the Nexus 4 device. On the other hand, SilentSense—which uses SVM—takes close to 1 and 6 seconds for initialization and training, respectively. The CPU and memory overhead in the initialization process is due to the loading of negative instances from an app's internal storage. However, this overhead is incurred only once; after the creation of the training model, feature extraction and classification takes less than a millisecond for a swipe. The execution results show that both devices are able to extract features, and classify in a reasonable amount of time.

Battery Consumption Overhead of Itus: We use PowerTutor [42] to measure battery consumption overhead by Itus. Micro-level overheads are recorded only during individual user input events and are computed relative to the individual demo apps, while macro-level overheads are recorded across a longer period of usage and computed relative to the device. For reproducible experiments, we do not perform any network operations (i.e., swipes are made on pre-downloaded pages in Zirco and for TextSecure the typed message is not transmitted). Finally, battery consumption overhead results do not include the one-time training cost of Itus prefabs.

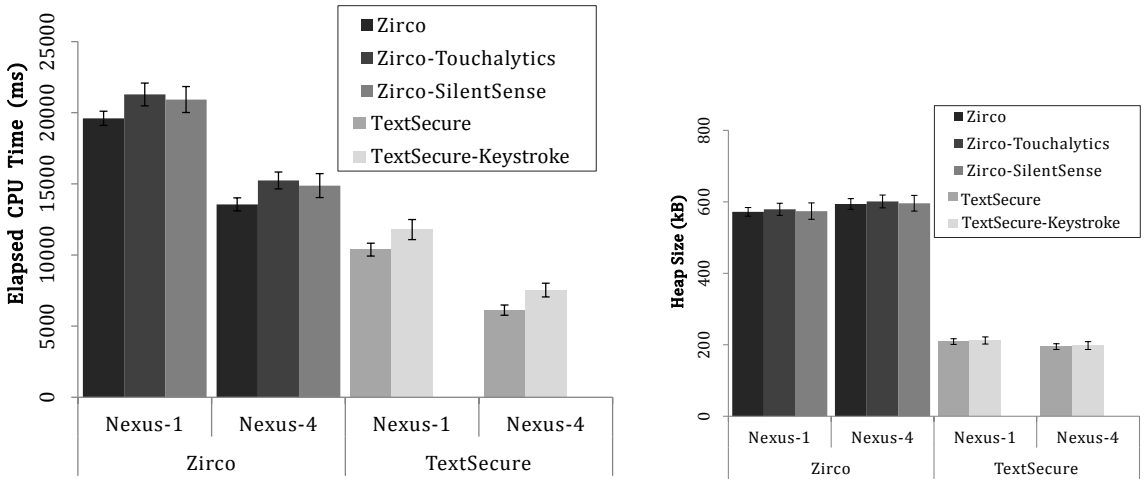
	Overhead on the app	Overhead on the device
Keystroke	38.6% (2.8)	1.23% (0.24)
Touchalytics	8.91% (0.64)	3.74% (0.53)
SilentSense	14.2% (0.95)	6.23% (0.89)

Table 2: Battery consumption overhead of Itus Prefabs on demo apps with a Nexus 1 device. Standard deviations are calculated across every hour of testing for each of 12 hours, and are shown in parentheses.

For overhead at the micro-level, we measure the power consumption of the Keystroke prefab by generating 160 keystroke events for classification. Similarly, we generate 40 swipe events for classification by the Touchalytics and SilentSense prefabs. Our empirical evaluation results, provided in the "overhead on the app" column of Table 2. The battery overhead for TextSecure is significantly higher because the battery consumption for normal operation is negligible (unlike Zirco which requires relatively expensive graphic rendering when swiped).

Computation of the macro-level overhead is a non-trivial task due to the large number of variables involved, such as the number of running apps, network connection (3G/WiFi) and the usage of other apps by the device user. To mitigate these variations, we create a simple setup in which Nexus 1 devices are only executing the demo app, the Google Mail app, the Google Talk app and the Launcher app. These devices are also executing the default supporting systems Network Location, User Dictionary, Media Server and the Radio and WiFi subsystems. Monkey scripts [2] were used to generate 40 swipe and 160 keystroke events after every 10 minute interval for 12 hours. Table 2 provides the results for the macro level overhead in the "overhead on the device" column. It can be observed from these results that all Itus Prefabs incur reasonably low overhead (1.23%, 3.74% and 6.23% for Keystroke, Touchalytics and SilentSense prefabs, respectively) even on Android devices with a minimal set of apps running.

Performance Overhead on Demo Apps: In addition to the performance evaluation of standalone Itus, we also measure the performance overhead imposed by Itus on two demo apps. The objective of this evaluation is to show that the relative performance overhead of Itus is negligible enough to not compromise user experience. We instrument and measure elapsed CPU time and heap memory size for the demo apps with and without Itus. The experiment with Zirco was conducted by accessing the BBC and CNN home-



(a) CPU Overhead

(b) Memory Overhead

Figure 4: Itus' CPU and memory overhead for demo apps. Error bars represent 95% confidence intervals.

pages using Zirco and swiping 20 times on each website without any delays. The experiment with the TextSecure app was conducted by composing a text message of 160 characters (network transmission was not included). Figures 4a, 4b show the CPU overhead and memory overhead for demo apps, respectively. It can be observed from Figure 4 that the CPU and memory overhead for Itus is negligible and Itus can be used without compromising user experience.

8. LIMITATIONS

Itus is part of our on-going research work. We have evaluated Itus on a variety of Android devices from different manufacturers and found it to be robust. However, there are a few limitations to deployment on old and low-end Android devices. Processing intensive IA schemes on Android devices with low-end processors might affect user experience. Similarly, for IA schemes that rely on sensor data from different on-board sensors, the accuracy of Itus will be negatively affected on some of the old devices with low sampling rates. However, these are trivial limitations given the high penetration of modern Android devices.

Another limitation of our approach is that each instance of Itus executes in isolation from other instances on the same device. Therefore, every app that uses Itus requires a separate training model and an instance of the Itus library in memory. This requires additional storage and increases apps' memory footprints (although it should be noted that the Itus Oracle can be used to reduce the memory footprint, as discussed in § 4.7). Finally, while independent execution of multiple Itus instances precludes any information sharing across these instances, this is by design to prevent any potential security issues arising from malicious apps.

Itus deployment is of course subject to all the limitations of existing IA mechanisms, including the requirement that it be configured to track representative input data from the current operator of a device. If, for example, an intruder comes across a device resting on a stable surface and protected by the SilentSense prefab, there would be no movement data available to classify him. Furthermore, it may be possible for attackers to use mimicry attacks to defeat IA [30] or to root a device. One of the benefits of providing an extensible IA framework is that Itus can easily incorporate defences against mimicry attacks as soon as they are proposed by security researchers. Attackers rooting a device are hard to defend against in general and not part of our threat model.

9. RELATED WORK

Explicit authentication schemes (such as pass-locks, facial and fingerprint recognition) are only partly related to our work since we understand their importance as a primary defense mechanism. We only advocate application of IA as a second line of defense due to usability [4, 19] and security issues [1, 41, 39] of explicit authentication schemes.

For IA on smartphones, various behaviour-based classifiers have been proposed that employ users' call / text patterns [34], location patterns [38], keystroke patterns [8, 12, 20, 27, 40], proximity to known devices [21, 29], gait patterns [13, 15], and touch screen input behaviour [5, 10, 11, 14, 23, 33]. Furthermore, some approaches have proposed to combine behaviour-based classifiers and contextual information from multiple sources [29, 32, 34] to implicitly authenticate a user. We differ from these prior research efforts in that our work places emphasis on the architecture and deployment aspects of an IA approach, rather than the efficacy of behaviour-based classifiers. In fact, any existing IA scheme can be deployed using our architecture to mitigate the limitations discussed in § 2.

Clarke et al. [7] proposed the design of a framework to support continuous and transparent authentication using facial, voice and keystroke biometrics. Crawford et al. [9] proposed a similar system that supports multimodal combination of these biometrics. However, the focus of their work is to provide design guidelines of such a framework for an IA scheme that operates at the device-level. Contrary to their approach, we propose a framework that enables app developers to integrate continuous authentication in their apps directly. Furthermore, the authors of [7, 9] only provide design guidelines for transparent and continuous authentication systems and stop short of providing an implementation of their de-

sign that can be used by device owners. Riva et al. [29] have built a prototype to use face recognition, proximity, phone placement, and voice recognition to progressively authenticate a user. Their prototype on Windows phone reduces the number of required authentications by 42%. While they have a common goal with our work in terms of when to authenticate, their scheme would require kernel-level support to operate and therefore has the limitations discussed in § 1.

There are a few approaches that have a common goal with ours in terms of providing an optimal trade-off between usability and security. For example, Hayashi et al. [19] discuss the inefficiency of the all-or-nothing access model and suggest that a user should be authenticated only when a sensitive app is launched. They also discuss shared access scenarios and propose an activity lock that can be used by a device owner to share specific screens in an app or a group account to share a specific set of apps between multi-user environments. In a related work, [17, 18] use multiple implicit factors (from user context) to determine how to explicitly authenticate a user. However, our work is different from these approaches since in addition to allowing an app developer to selectively invoke an authenticating module based on the type of an app, we delegate IA tasks to an app and not to the device.

10. CONCLUSION

We have proposed Itus, a framework for providing IA support on smartphones, and provided an open-source implementation for Android. Itus separates the domain knowledge of IA from its deployment to bridge the gap between IA research and practice. The architecture of Itus is designed with flexibility in mind for app developers, allowing them choice between modular subcomponents implementing different mechanisms for behavioural feature classification. The Itus framework is easily extensible to allow IA developers to easily provide such subcomponents. Itus also provides application developers with the ability to optionally configure the behaviour of the IA mechanism to their application's needs by using an Oracle program to determine suitable classifiers and configuration parameters. The API of Itus has been designed to trivialize the effort to provide IA in Android apps. Empirical evaluations of Itus in real-world demo apps show that it has minimum overhead. We have made Itus publicly available in open-source for Android and we are currently working on providing support for more IA schemes in Itus. In future work, we will conduct usability studies to determine the efficacy of IA on smartphones for users and the utility of Itus for app developers. We hope that Itus will enable the research community to collaborate better to further research in the IA domain and also to enable the adoption of IA.

11. ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their valuable comments. We also thank Google and NSERC for their support.

12. REFERENCES

[1] Android Authority. Android jelly bean face unlock liveness check easily hacked with photo editing. `http://www.androidauthority.com/android-jelly-bean-face-unlock-blink-hacking-105556/`, Mar. 2014.

[2] Android Tools. Application exerciser monkey. `http://developer.android.com/tools/help/monkey.html`, Mar. 2014.

[3] A. J. Aviv, K. Gibson, E. Mossop, M. Blaze, and J. M. Smith. Smudge attacks on smartphone touch screens. In *Usenix Workshop on Offensive Technologies*. Usenix, 2010.

[4] N. Ben-Asher, N. Kirschnick, H. Sieger, J. Meyer, A. Ben-Oved, and S. Möller. On the need for different security methods on mobile phones. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, pages 465–473. ACM, 2011.

[5] C. Bo, L. Zhang, X.-Y. Li, Q. Huang, and Y. Wang. Silentsense: silent user identification via touch and movement behavioral biometrics. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, pages 187–190. ACM, 2013.

[6] C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27, 2011.

[7] N. Clarke, S. Karatzouni, and S. Furnell. Flexible and transparent user authentication for mobile devices. In *Emerging Challenges for Security, Privacy and Trust*, pages 1–12. Springer, 2009.

[8] N. L. Clarke and S. Furnell. Authenticating mobile phone users using keystroke analysis. *International Journal of Information Security*, 6(1):1–14, 2007.

[9] H. Crawford, K. Renaud, and T. Storer. A framework for continuous, transparent mobile device authentication. *Computers & Security*, 39:127–136, 2013.

[10] A. De Luca, A. Hang, F. Brudy, C. Lindner, and H. Hussmann. Touch me once and i know it's you!: implicit authentication based on touch screen patterns. In *Proceedings of the 2012 ACM Annual Conference on Human Factors in Computing Systems*, pages 987–996. ACM, 2012.

[11] T. Feng, J. Yang, Z. Yan, E. M. Tapia, and W. Shi. Tips: Context-aware implicit user identification using touch screen in uncontrolled environments. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. ACM, 2014.

[12] T. Feng, X. Zhao, B. Carbunar, and W. Shi. Continuous mobile authentication using virtual key typing biometrics. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1547–1552. IEEE, 2013.

[13] J. Frank, S. Mannor, and D. Precup. Activity and gait recognition with time-delay embeddings. In *AAAI Conference on Artificial Intelligence*, 2010.

[14] 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*, 8(1):136–148, 2013.

[15] D. Gafurov, K. Helkala, and T. Søndrol. Biometric gait authentication using accelerometer sensor. *Journal of Computers*, 1(7):51–59, 2006.

[16] Google Play. Google play install stats. http://developer.android.com/about/dashboards/index.html, Mar. 2014.

[17] A. Gupta, M. Miettinen, N. Asokan, and M. Nagy. Intuitive security policy configuration in mobile devices using context profiling. In *International Conference on Social Computing, Privacy, Security, Risk and Trust*, pages 471–480. IEEE, 2012.

[18] E. Hayashi, S. Das, S. Amini, J. Hong, and I. Oakley. Casa: context-aware scalable authentication. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, page 3. ACM, 2013.

[19] E. Hayashi, O. Riva, K. Strauss, A. Brush, and S. Schechter. Goldilocks and the two mobile devices: going beyond all-or-nothing access to a device's applications. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 2. ACM, 2012.

[20] S.-s. Hwang, S. Cho, and S. Park. Keystroke dynamics-based authentication for mobile devices. *Computers & Security*, 28(1):85–93, 2009.

[21] A. Kalamandeen, A. Scannell, E. de Lara, A. Sheth, and A. LaMarca. Ensemble: cooperative proximity-based authentication. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, pages 331–344. ACM, 2010.

[22] H. Khan and U. Hengartner. Towards application-centric implicit authentication on smartphones. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. ACM, 2014.

[23] L. Li, X. Zhao, and G. Xue. Unobservable reauthentication for smart phones. In *Proceedings of the 20th Network and Distributed System Security Symposium*, volume 13, 2013.

[24] Lookout Blog. Sprint and lookout consumer mobile behavior survey. http://blog.lookout.com/blog/2013/10/21/sprint-and-lookout-survey/, Mar. 2014.

[25] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, pages 165–178. ACM, 2009.

[26] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 71–84. ACM, 2010.

[27] E. Maiorana, P. Campisi, N. González-Carballo, and A. Neri. Keystroke dynamics authentication for mobile phones. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 21–26. ACM, 2011.

[28] S. Mare, A. Molina-Markham, C. Cornelius, R. Peterson, and D. Kotz. Zebra: Zero-effort bilateral recurring authentication. In *IEEE Symposium on Security and Privacy*. IEEE, 2014.

[29] O. Riva, C. Qin, K. Strauss, and D. Lymberopoulos. Progressive authentication: deciding when to authenticate on mobile phones. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[30] A. Serwadda and V. V. Phoha. When kids' toys breach mobile phone security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pages 599–610. ACM, 2013.

[31] A. Serwadda, V. V. Phoha, and Z. Wang. Which verifiers work?: A benchmark evaluation of touch-based authentication algorithms. In *IEEE Sixth International Conference on Biometrics: Theory, Applications and Systems*, pages 1–8. IEEE, 2013.

[32] A. Shabtai, U. Kanonov, and Y. Elovici. Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method. *Journal of Systems and Software*, 83(8):1524–1537, 2010.

[33] M. Shahzad, A. X. Liu, and A. Samuel. Secure unlocking of mobile touch screen devices by simple gestures: you can see it but you can not do it. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, pages 39–50. ACM, 2013.

[34] E. Shi, Y. Niu, M. Jakobsson, and R. Chow. Implicit authentication through learning user behavior. In *Information Security*, pages 99–113. Springer, 2011.

[35] B. Shrestha, N. Saxena, H. T. T. Truong, and N. Asokan. Drone to the rescue: Relay-resilient authentication using ambient multi-sensing. In *Proc. Eighteenth International Conference on Financial Cryptography and Data Security*, 2014.

[36] P. Steiner. Going beyond mobile device management. *Computer Fraud & Security*, 2014(4):19–20, 2014.

[37] A. Striegel, S. Liu, L. Meng, C. Poellabauer, D. Hachen, and O. Lizardo. Lessons learned from the netsense smartphone study. *SIGCOMM Computer Communication Review*, 43(4):51–56, Aug. 2013.

[38] A. Studer and A. Perrig. Mobile user location-specific encryption (mule): using your office as your password. In *Proceedings of the Third ACM Conference on Wireless Network Security*, pages 151–162. ACM, 2010.

[39] Threatpost. Lock screen bypass flaw found in samsung androids. http://threatpost.com/lock-screen-bypass-flaw-found-samsung-androids-030413/77580, Mar. 2014.

[40] S. Zahid, M. Shahzad, S. A. Khayam, and M. Farooq. Keystroke-based user identification on smart phones. In *Recent Advances in Intrusion Detection*, pages 224–243. Springer, 2009.

[41] Zdnet. Apple iphone fingerprint reader confirmed as easy to hack. http://www.zdnet.com/apple-iphone-fingerprint-reader-confirmed-as-easy-to-hack-7000021065/, Mar. 2014.

[42] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 105–114. ACM, 2010.