Enhanced Password Security on Mobile Devices

by

Dongtao Liu

Department of Computer Science
Duke University

Date:_____

Approved:

_____
Landon P. Cox, Supervisor

_____
Bruce M. Maggs

_____
Xiaowei Yang

_____
Romit Roy Choudhury

Dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of Duke University

2013

ABSTRACT

Enhanced Password Security on Mobile Devices

by

Dongtao Liu

Department of Computer Science
Duke University

Date:_____

Approved:

_____
Landon P. Cox, Supervisor

_____
Bruce M. Maggs

_____
Xiaowei Yang

_____
Romit Roy Choudhury

An abstract of a dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of Duke University

2013

Abstract

Sleek and powerful touchscreen devices with continuous access to high-bandwidth wireless data networks have transformed mobile into a first-class development platform. Many applications (i.e., "apps") written for these platforms rely on remote services such as Dropbox, Facebook, and Twitter, and require users to provide one or more passwords upon installation. Unfortunately, today's mobile platforms provide no protection for users' passwords, even as mobile devices have become attractive targets for password-stealing malware and other phishing attacks.

This dissertation explores the feasibility of providing strong protections for passwords input on mobile devices without requiring large changes to existing apps.

We propose two approaches to secure password entry on mobile devices: ScreenPass and VeriUI. ScreenPass is integrated with a device's operating system and continuously monitors the device's screen to prevent malicious apps from spoofing the system's trusted software keyboard. The trusted keyboard ensures that ScreenPass always knows when a password is input, which allows it to prevent apps from sending password data to the untrusted servers. VeriUI relies on trusted hardware to isolate password handling from a device's operating system and apps. This approach allows VeriUI to prove to remote services that a relatively small and well-known code base directly handled a user's password data.

Dedication

To my dear maternal grandmother,

Yuzhen Wang (Oct. 27, 1934 ~ May 22, 2010).

Contents

List of Tables

List of Figures

Acknowledgements

It is an interesting, rewarding, as well as hardworking journey to pursue a doctoral degree. Depressing and exciting moments alternate during the past five years. It would not have been possible to finish this dissertation without the guidance of my advisor and other committee members, help from colleagues and friends, and support from my family. Here is my sincere thanks to them for being with me.

First and foremost, I would like to express my gratitude to my advisor, Landon P. Cox, for helping me grow as a researcher. He has taught me a lot of things from how to find meaningful topics for research, how to think critically, how to make trade-offs, how to write logically and accurately, and how to give a clear and attractive presentation. I will always remember those days we were working together for paper deadlines.

I am also very thankful to my committee members: Bruce M. Maggs, Xiaowei Yang, and Romit Roy Choudhury. I consider myself very lucky for having such a strong committee. They have provided me with invaluable suggestions during my preliminary exam and the final defense.

I would like to thank Eduardo Cuervo and Peter Gilbert, for their help on my research. I would also like to thank Dong Liu, Yin Lin, Ryan Scudellari, Zhiqiu Kong, Bi

Wu, Valentin Pistol, Xin Liu, Ang Li, Xin Wu and many more people I was lucky to meet at Duke. Without them life as a PhD student would be less enjoyable.

Most importantly, my greatest gratitude goes to my wife, Meng Zhang, for her love, care, support, encouragement, and accompany. I consider myself so lucky to meet her here at Duke and words cannot express how grateful I am to her for sharing her life with me for more than five years. The long trip towards the PhD would be a dull and miserable one if it was not for her who has stood by me no matter how difficult things might seem. I would like to thank her for cheering me up every time I was in low spirit. She has offered me love, comfort and assurance when things did not look as promising and she has been there to enjoy the better times with me. In short, I would not survive the graduate school without her support. My adorable and lovely little son, Grayson Liu, is the best and precious gift of life, encouraging me to pursue my research even in the most difficult time. I would also like to thank my mother-in-law, Wei Xu, for her greatest help in taking care of my little son during the final year of my PhD life. Without her help, I have no way to complete this dissertation in time. Finally, I want to thank my mom, Lianying Liu, for her unconditional love and support throughout my life, that help me evolve from an infant to a productive member of the society. Her education in my early years keeps me motivated through the PhD journey. I am forever indebted to all of them.

1. Introduction

1.1 Booming Mobile and Cloud Services

Mobile devices, such as smartphones and tablets have become immensely popular in recent years. According to a 2012 survey, Smartphone ownership has reached half of all U.S. mobile consumers [8]. Only in the second season of 2013, nearly 180 million Android devices and over 30 million iOS devices were sold worldwide [6]. Users are attracted by the powerful and interesting apps on the devices and spend a lot of time on them. By 2013, Google Play had nearly 1 million applications available in the market with 50 billion downloads in total [7], versus 0.8 million applications and 40 billion downloads in Apple iOS app store [2]. American users spend an average of 58 minutes per day on their mobile devices, of which 14% on browsing the mobile websites while the rest through mobile apps [11]. 27% of photos were taken through smartphone according a survey in Dec. 2011 [16].

At the meantime, lots of new cloud services are booming rapidly and taking up almost all the aspects of people's life: social networks, instant messaging, video streaming, photo gallery, e-business, location-based services (LBS), online storage, etc. These cloud services have attracted thousands of millions of users worldwide. Facebook reached 1.11 billion monthly-active users all over the world as of May 2013 [4], who contribute to 3 million messages and 1 million links every 20 minutes [5]. Over 1 billion

monthly-active users of YouTube upload 100 hours of video every minute, and over 6 billion hours of video are watched each month [17]. More and more people access cloud services from their mobile devices. Facebook has 680 million mobile users [5]. Instagram has attracted 150 million monthly active users who generate 55 million photos per day through the mobile app [12]. 80% smartphone owners use their mobile device to shop online, according to a study conducted in Sept. 2012 [3].

The reason for such high level of participation in the mobile and cloud services is obvious: they bring a lot of fun and convenience to the users. People can look up in the map through their smartphone when they are adventuring the new cities. Or they can watch funny YouTube videos while waiting in the subway. Users can make video calls with colleagues no matter how far away they are. Or they can share photos taken in the trip immediately with best friends all over the world through OSNs. As a result, the functionalities provided by mobile devices and cloud services restyle people's daily life and redefine how they interact with the whole world.

1.2 New Security Challenges in the Mobile Era

Mobile and cloud services bring convenience and fun to the users. However, enhancing security on mobile computing becomes an important and pressing problem, because mobile devices not only store and generate huge amount of personal or even

sensitive data in local systems, but also become portable and popular clients to access all kinds of cloud services which control the entrances to users' cloud data.

Firstly, smartphones and tablets contains lots of personal information in the system, a considerable part of it is highly private or sensitive. Mobile devices nowadays are far more than simple call phones to users. Instead, they are becoming important personal assistants and useful portable computers. Besides contacts and messages in these devices, they may also contain private data such as daily schedule, scratch notes, or even business presentations.

In addition, as most modern smartphones and tablets are equipped with a number of sensors such as GPS, camera, microphone, accelerometer, gravimeter, pressure and light sensors, etc., users are very easy to generate information-rich content through their mobile devices. The sensors on mobile devices can sense the environment as well as users themselves comprehensively. As a result, contents from user-centric sensing contains a lot of personal or even sensitive information about the users such as locations, life habits, health conditions, and social activities.

Lastly, more and more people routinely access all kinds of cloud services through their mobile devices because of availability and convenience. Due to the high portability of mobile devices and the wide availability of internet access, users are able to interact with the cloud services almost anywhere and anytime through their mobile

clients. It is very convenient to generate User Generated Contents (UCG) from mobile devices and send them to cloud services. Due to the limitation of power, storage, and computation capacities of mobile devices, many mobile apps only serves as light-weighted clients to display and handle user interactions, and they rely on cloud servers to process and store huge amount of data. Before mobile apps can access to cloud services, users have to give them credentials for authentication with servers. The credentials such as usernames and passwords are very sensitive because they are the keys to access users' properties in the cloud. There are lots of private and sensitive data in the clouds, such as commercial emails, social network photos, instant messenger contacts, and online documents, etc. Private data in the cloud will be in great danger if the credentials are not properly handled in the mobile devices and stolen by malicious entities. In addition, it is also unavoidable for users to perform other sensitive interactions with cloud services through mobile apps, such as online payment. Such sensitive interactions and operations must take place in a secure environment as well.

Since mobile devices such as smartphones and tablets contains lots of private information and access all kinds of cloud services on behalf of users, no doubt, they become valuable targets for malwares and attackers. Malicious apps may steal users' credentials or fool users to pay online. Improper or buggy apps may have vulnerabilities in their code and put users' sensitive information in high risks. Although centralized

app markets and app review processes do help relieve these problems, apparently they are not enough. Examinations on mobile apps before release cannot detect all the problems in them. On the other hand, it does allow installing apps from untrusted sources in some mobile platforms such as Android.

In this dissertation, we focus on protecting sensitive input data when users are accessing cloud services through mobile apps and taking sensitive operations, such as authentication and online payment. When users interact with mobile apps and input sensitive information, they need to make sure that the mobile apps handle their sensitive operations properly and securely. Our goal is to enhance the security for users' most important data, and also relieve the concerns of users when they are enjoying the fun and convenience provided by mobile apps.

Similar security problems have been studied in desktop computers. However, we are facing the following new challenges for touchscreen mobile devices:

Third-party apps: Third-party apps become more and more popular in the mobile app market. Many cloud services open their platform APIs to third-party developers and encourage them to contribute to the ecosystem. For example, popular OSNs such Facebook and Twitter allow third-party apps to access users' OSN data once users grant permits to them. These third-party apps greatly extended the services provided by the original platforms and fill in their gaps. People choose to use third-

party apps because they provide augmented functionality and integrated services, or just because first-party app is unavailable in the market. However, when users are accessing first-party cloud services through third-party mobile apps, they unavoidably have to give sensitive data to third-party apps for authentication, online payment, etc. This causes great security issues. Currently there is no enforcing mechanism to isolate third-party apps completely from the secrets between users and cloud services. And there is also lack of regulation mechanism to monitor how the sensitive data is being handled by third-party apps so as to prevent improper behaviors by buggy or even malicious code.

Software keyboard: The input entry is no longer trusted in current touchscreen devices. Most mobile devices are touchscreen devices, which do not equipped with hardware keyboards and only provide users with software input methods. In desktop computers with hardware keyboard, the keyboard driver is part of system code and handles user inputs directly. But in touch screen devices, only the touchscreen driver is part of the system code, and the input method works as an application to turn touch actions into key strokes. So any mobile app can implement a software keyboard within the app. It can translate the user taps into character inputs all by itself, without calling the default input method service. At a result, the entry for inputting sensitive information is no longer trusted in touchscreen devices.

Embedded web UI: The web UI is not secure and can be manipulated by mobile apps. Besides mobile browsers, mobile system also provides developers with an UI widget called WebView. WebView provides basic browser functionalities and can be embedded in the mobile apps to display web contents. It is widely used in mobile apps because it is a convenience way to communicate with cloud services. Many mobile apps embed it to serve as a highly-customized mini-browser. However WebView also raises new security issues to the system because it can be easily manipulated by its host mobile app. To help web applications tightly integrated with mobile apps, WebView provides a number of APIs for developers to register event listeners, and inject JavaScript code, etc. The host app can easily hijack the web content and sniffer user input in WebView. As a result, WebView compromise the trust base for browsers. Different from desktop browsers such as Chrome and Firfox in which people trust, WebView is not trusted unless its host app is completely trusted and secured.

Small screen size: The much smaller screen size in mobile devices makes it even more difficult to protect users against social-engineering attacks than in desktop computers. Screen is very precious estate in smartphones so that most mobile system supports full-screen mode to maximize display utilization. There is no reservation area in the screen and mobile apps are allowed to arbitrarily write to any part of the screen. As a result, secure areas or indicators which are used against spoofing attacks in desktop

computers and browsers, do not work anymore on mobile devices. What's even worse, small-size screen with less contents and visual clues displayed, makes users more difficult to judge the authenticity of the UIs and easier to be deceived. Mobile browsers which usually hide the address bar to save the screen, attracts little attention from the users on the domain changes in the URLs. All these factors combined together make the social-engineering attacks much more difficult to prevent.

1.3 Enhance Trusted UI in Touchscreen Mobile Devices

To solve the new security challenges in touchscreen mobile devices, we propose to enforce trusted UI to guarantee the security for sensitive operations and protect users' sensitive information against improper or even malicious mobile apps.

We practice the following three principles to design and implement trusted UI in our systems, which are also three major challenges we must consider and solve:

- **Secure environment:** Trusted UI must provide users with a secure environment to complete sensitive operations, such as authentication or online payment. It must be part of trusted computing base (TCB) in the system, or at least controlled and monitored by TCB. The display and operation of trusted UI must be handled by trusted code.
- **Usage Enforcement:** The usage of trusted UI by mobile apps must be enforced. In other words, there must be an enforcing mechanism to

guarantee mobile apps will use trusted UI to handle sensitive user interactions and input. Whenever mobile apps ask users to take sensitive operations, they have no way to circumvent trusted UI but only adopt it. At least, if an improper or malicious mobile app is trying to use its own UI to handle user interactions instead of trusted UI, TCB in the system must detect and prevent it in time.

- **Anti-spoofing campaign:** The authenticity of the trusted UI must be protected. Any security mechanism handling user interactions must consider social-engineering attacks. Malicious apps can display identical or highly similar UI to spoof the users, in order to steal sensitive information from them. TCB in the system must be able to detect and prevent phishing or spoofing attacks. More importantly, the user responsibility must be minimized in the anti-spoofing campaign. We cannot rely on the users to make critical judgments whether they are interacting with the trusted UI or not. Users are not expertise on security and prone to ignore warnings according to previous studies. They have less information than the system about what is happening on the device. If users have to be fully trained or educated to prevent spoofing attacks, the security mechanism will be definitely vulnerable and easy to fail.

In our research, we have done two case studies, ScreenPass and VeriUI to enhance the security of sensitive input through trusted UI. Both studies practice the three major principles in their designs and implementations.

1.4 Dissertation Statement and Overview

Security on mobile computing, especially the security of sensitive input data through mobile apps, is the focus of my dissertation. Sensitive input data includes but not limited to: credentials such as passwords, financial accounts such as credit card numbers, personal identities such as SSN numbers, etc. My dissertation statement is as follows:

> **When users access cloud services from touchscreen mobile devices, it is feasible to enhance the security for sensitive input by enforcing trusted UI in mobile apps.**

This dissertation validates the statement via the design, implementation, and experimental evaluation of two systems: ScreenPass and VeriUI.

We propose ScreenPass – a system that enhances password security on touchscreen devices through OCR and taint-tracking techniques. Passwords are highly sensitive data, and users need to ensure their passwords are handled properly by mobile apps. In order to provide users with trusted password entry, ScreenPass checks the

authenticity of software keyboard using optical character recognition (OCR) to validate it is legally called through system input method framework. ScreenPass provides a special UI to capture user intent explicitly when she inputs the password, and tags the password with the secure domain which the user selected. After the password is properly tagged, ScreenPass monitors the password usage throughout the system using taint-tracking, and enforce security policies to protect the password against improper or even malicious mobile apps.

We also propose VeriUI – a system that enhances security for sensitive operations on mobile devices using ARM TrustZone technique. VeriUI leverages ARM TrustZone technique to run two systems at the same time: a normal system with rich applications in the normal world, and a minimized secure system for users to take sensitive operations in the secure world. A minimized and secure webkit runs in the secure word to provide generalized service to third-party apps running in the normal world. Secure webkit sends requests to cloud servers together with attestations generated by the secure system, which proves the secure environment. First-party cloud services checks and verifies the remote attestations before they process the requests so that third-party apps are required to use trusted web UI in the secure world to handle sensitive user interactions with first-party cloud servers.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows: Chapter 2 provides the background knowledge and techniques related with ScreenPass and VeriUI, including Android system, OAuth 2.0 protocol, taint-tracking, and trust computing. Chapter 3 and Chapter 4 describe the major contributions of this dissertation. Chapter 3 describes the design, implementation, and evaluation of ScreenPass, a system that enhances password security in touchscreen device. ScreenPass verifies the authenticity of software keyboard through OCR, and monitors the usage of passwords by mobile apps through taint-tracking. Chapter 4 presents the design, implementation, and evaluation of VeriUI, a system that enhances security for sensitive operations in mobile devices. VeriUI leverages ARM TrustZone technique to provide users with trusted web UI in the secure world, and enforces the usage of trusted UI by third-party apps through remote attestation. Chapter 5 provides the related work on password security, anti-phishing techniques, trusted UI, and trust computing, etc. It discusses the state of art in these fields and compares it with ScreenPass and VeriUI respectively. Chapter 6 concludes the dissertation with a summary of our contributions. It also discusses our experiences on designing and implementing ScreenPass and VeriUI.

2. Background

This chapter provides some background knowledge regarding the design and implementation of ScreenPass and VeriUI. We first briefly introduce some important components of Android system in Section 2.1: Android framework, web browser, and display subsystem. After that, we provide an overview of OAuth 2.0 protocol for authorization and authentication in Section 2.2. In Section 2.3 we discuss the taint-tracking technique and an existing implementation for Android - TaintDroid. Finally, we present the background knowledge related to Trust Computing especially ARM TrustZone and remote attestation on Section 2.4.

2.1 Android System

The Android mobile platform includes a Linux kernel, Java programming environment, and several layers of trusted middleware. We briefly describe the most relevant of these subsystems below.

2.1.1 Android Framework, Apps, and Services

Application framework: An Android app consists of three components. An activity defines the app's UI, and controls all of an app's widgets (called "views" in Android). Activities interact with background services and content providers. An application's background service is a separate thread that contains the application's core

logic and soft state. An app's content provider offers an interface to persistent storage such as the file system or a SQL database.

Dalvik virtual machine: App components are written in Java, and this code is compiled into Dalvik Execution (DEX) byte codes. The Android team developed their own Dalvik Virtual Machine (VM), which is optimized for mobile devices. Android apps execute within separate VM instances, and VM instances are managed as processes by the underlying Linux kernel. Apps use Android's custom IPC protocol (Binder) to communicate with processes outside of their own.

Input method framework: The Input Method Framework (IMF) provides a way to invoke and manage input methods such as onscreen software keyboards. When a user selects a text-input box, the widget makes a request to the IMF for an Input Method Engine (IME) that can receive input from the user. The IME runs in a separate process from the requesting app and writes user input to a string field in the text-input widget via IPC.

2.1.2 Mobile Browser and WebView Widget

There are two types of web UI facilities on Android: the mobile browser and WebView widgets.

A WebView is a system UI widget that serves as an embedded browser for loading URLs within an app. Mobile developers can easily integrate WebView

components in their apps to render web pages and display contents from cloud services.WebView supports basic browser functionality such as navigation and JavaScript execution. It also provides an API for mobile apps to interact with and customize WebView output. Because of its usefulness, flexibility, and ease of use, WebView are widely used in mobile apps.

There are several ways that an app can monitor its in-app WebView components. Firstly, it can register event listeners through WebView APIs. With these event listeners, the app can hijack the web content, or even sniff everything the user does with the webpage. Secondly, the app can inject and execute JavaScript code into the web page. The injected JavaScript code can also invoke Java code to pass information back to the host app. Through these methods, the webpage loaded in WebView is tightly integrated with the mobile app and can be easily customized. However, these WebView APIs compromise the trustworthiness of the embedded browsers. When users interact with desktop browsers, they trust them because they are independent programs. But the WebView is only as trustworthy as its host app.

The mobile browser is a stand-alone app supporting full-fledged web features as its desktop counterparts, and can be called by other apps via IPC. Each Android app runs as a Linux process with its own unprivileged UID. An app can request the mobile browser to open any URL through Binder IPC calls. The app yields control to the mobile

browser and runs in background. The mobile browser becomes the foreground app to interact with the user. After finishing its work, the mobile browser can be redirected back to the app. Unlike the embedded WebView, third-party apps have no control over the mobile browser.

2.1.3 Display Subsystem

Frame Buffer: The Linux frame buffer (FB) is an abstraction that provides access to the graphical output of the device. Android's FB device is found at /dev/graphics/fb0 and can be accessed by only the root user and the graphics group. The FB has a front and back buffer: the front buffer contains the pixel data currently displayed on the screen, and the back buffer is used for composition by the SurfaceFlinger.

SurfaceFlinger: The SurfaceFlinger (SF) is Android's system-wide surface composition engine, and is responsible for managing surfaces and the virtual frame buffer device. When an activity creates a widget, it asks the SF to allocate a new surface. The SF allocates a back and front buffer for the surface and returns a handle to the requesting process. As the owner of this surface, the requesting process is free to update the back buffer as needed. The back buffer is used by the activity for rendering new frames. Once a process has finished rendering, it swaps the front and back buffers and asks the SF to draw the new frame to the screen. The front buffer is used by the SF for composition.

2.2 OAuth 2.0 Protocol

OAuth 2.0 [9] is an open and standard resource authorization protocol which is widely used by modern web services. It enables users to grant a third-party to access to their resources stored in the cloud to a limited extent, without sharing their long-term credentials, such as usernames and passwords. OAuth allows a cloud service to generate a capability called OAuth token for an app on the user's behalf which will be used as permit to access the user's data later. Every OAuth token corresponds to a list of resources in the cloud and has an expiration date. The user can also actively revoke an app's OAuth token before it expires through the cloud service. OAuth protocol supports different types of third-party apps, such as websites, browser plug-ins, mobile apps, and desktop applications. OAuth is designed as an authorization protocol, but it is also widely used by third-party apps in single sign-on (SSO) authentication services.

The procedure for mobile apps to obtain an OAuth token from cloud services is as follows: when a mobile app first request access to a resource in the cloud service, it redirect the user to the OAuth login website of the cloud service through WebView default browser, together with a list of resources it requires in the parameters. The user completes the authentication process through the web UI and approves the required permissions from the app. The cloud service will create an OAuth token, and a client

secret for the app to retrieve the OAuth token later. Then it redirects the WebView or default browser back to the app, returning it with the generated client secret.

After presenting the client secret to the cloud service and receiving its OAuth token, the app can make API calls to the cloud service to access the corresponding user data by presenting the OAuth token. The mobile app usually stores the OAuth token in the client for future use, or even send the token to the app's own server.

OAuth 2.0 provides a standard solution for first-party cloud services to share data with third-party mobile apps, and it has been widely accepted and deployed. However, not all third-party apps implement the protocol properly and completely, a portion of third-party apps ask users for account information through its app UI directly and then complete the whole process by itself.

2.3 Taint-tracking and TaintDroid

ScreenPass relies on TaintDroid for taint-tracking. TaintDroid is a system-wide taint-tracking extension for Android [27]. TaintDroid associates tags with sensitive data that is released to untrusted code via taint sources such as requests for a device's current location. Taint tags are 32-entry bit vectors that can be associated with program variables, IPC messages, and files. TaintDroid implements tag-propagation logic to capture data dependencies as the system executes.

2.4 Trust Computing and ARM TrustZone

2.4.1 ARM TrustZone

TrustZone [19] is a set of security extensions for trusted computing supported by ARM processors since ARMv6, such as ARM Cortex A8, A9, and A15. TrustZone's security architecture embodies a "two worlds" paradigm. One world is called the "normal world" or rich execution environment (REE), and the other is called "secure world" or trusted execution environment (TEE). The two worlds are separated by introducing a special 33rd address line on the system bus. Hardware-based access control will be enforced according to the state on the 33rd address line, so that untrusted code in the normal world cannot access to the protected memory pages or peripheral registers of the secure world.

TrustZone supports running two operating systems at the same time without degrading system performance. The rich OS such as an Android system and all applications run in the normal world, whereas a secure OS such as a trusted kernel and secure applications runs in the secure world. Software stacks in the two worlds can be bridged through a secure monitor call (SMC). System switch is managed by the TrustZone monitor, which is part of the secure world. Once SMC is executed, the hardware switches to the TrustZone monitor to perform a secure context switch into the

secure world. Besides SMC, TrustZone monitor can also be entered by a number of hardware interrupts.

TrustZone protects code and data integrity and confidentiality in the secure world by isolating untrusted code running in the normal world from protected resources. Secure boot ensures the system components enter trusted states. When the platform boots, it first boots into the secure world and the system firmware setups the entire runtime environment for the secure world. After the secure OS boots, the secure world yields to the normal world by loading the bootloader for the rich OS. When the booting process finishes, the normal world must use SMC to call back into the secure world. As a result, even though the rich OS in the normal world is compromised, the data and code in the secure world is still not tampered.

2.4.2 Remote Attestation

Remote attestation is the process to make and verify a claim about the properties of a target by providing evidence to the authority to check through network. In system security, the evidence can be a hash of the system states to verify that the OS has not been tampered before taking sensitive operations. The attestation can be created by either trusted hardware or software.

Even though ARM TrustZone provides isolation between normal world and secure world, it does not provide remote attestation capabilities originally. This

requirement can be full filled through hardware or software solutions. For hardware method, an external TPM [18] or MTM module can be introduced into the platform to provide binding, sealing, and remote attestation. In software solution, these capabilities can be implemented in a trusted kernel in the secure world. Since TrustZone is robust, venders can build secure storage in the secure world. The keys and sensitive data can be stored in the TEE-only addressable area as protected resources to be isolated from the untrusted code. The trusted kernel can provide attestation to remote authorities.

3. ScreenPass: Secure Password Entry via OCR and Taint-tracking

In this chapter we describe ScreenPass [34] – a system that enhances password security on touchscreen devices through OCR and taint-tracking techniques.

Users routinely access cloud services through third-party apps on smartphones by giving apps login credentials (i.e., a username and password). Unfortunately, users have no assurance that their apps will properly handle this sensitive information. In this chapter, we describe the design and implementation of ScreenPass, which significantly improves the security of passwords on touchscreen devices. ScreenPass secures passwords by ensuring that they are entered securely, and uses taint-tracking to monitor where apps send password data. The primary technical challenge addressed by ScreenPass is guaranteeing that trusted code is always aware of when a user is entering a password. ScreenPass provides this guarantee through two techniques. First, ScreenPass includes a trusted software keyboard that encourages users to specify their passwords' domains as they are entered (i.e., to tag their passwords). Second, ScreenPass performs optical character recognition (OCR) on a device's screenbuffer to ensure that passwords are entered only through the trusted software keyboard. We have evaluated ScreenPass through experiments with a prototype implementation, two insitu user studies, and a small app study. Our prototype detected a wide range of dynamic and static keyboard-spoofing attacks and generated zero false positives. As long as a

screen is off, not updated, or not tapped, our prototype consumes zero additional energy; in the worst case, when a highly interactive app rapidly updates the screen, our prototype under a typical configuration introduces only 12% energy overhead. Participants in our user studies tagged their passwords at a high rate and reported that tagging imposed no additional burden. Finally, a study of malicious and non-malicious apps running under ScreenPass revealed several cases of password mishandling.

3.1 Introduction

Users routinely access cloud services such as Facebook and Twitter via third-party applications (i.e., "apps") on touchscreen devices. To interact with these services, an app must provide a user's password to one or more remote servers. Passwords are highly sensitive data and handing them over to third-party apps raises the following question: how can users be sure that an app properly handles their passwords? The recent discovery of password-stealing apps and other vulnerabilities in Android demonstrates that users have reason to be concerned [31, 58].

A trusted information-flow monitor such as TaintDroid [27] can track the propagation of password data, but data must be tagged before it can be tracked. Past systems have tagged sensitive user input via a secure attention sequence (SAS), such as "@@", to indicate the beginning of a password [41, 47]. Trusted software (i.e., a keyboard

driver) must monitor the incoming character stream for the SAS and, when it appears, must treat subsequent characters as password data.

However, recognizing SASes in the text-input stream of a touchscreen device is difficult because of software keyboards. Trusted code such as the touchscreen driver and user-interface manager receive taps on a touchscreen as a set of coordinates, and can only understand the intended meaning of a user's taps when they understand the content of the screen.

The platform could reserve part of the screen for secure gestures, but modern devices' small screens make screen real estate a precious resource. Important third-party apps such as games, media players, and web browsers need write access to the entire screen.

Unfortunately, a malicious app can abuse this privilege to spoof a platform's user interface, including its trusted software keyboard.

Under a spoofing attack, a user may input an SAS meant for the trusted platform without realizing that the input was delivered directly to a malicious app.

To ensure that passwords are input securely, we have developed a system called ScreenPass. ScreenPass provides a special-purpose software keyboard for entering sensitive text such as passwords that allows a user to tag her input with a domain (e.g.,

Google, Facebook, or Twitter). ScreenPass uses these tags to taint-track the user's password data as it propagates through the app.

Tags can subsequently be used to enable a number of useful policies.

For example, the system may want to know when plaintext password data is written to disk or when password data is shared between apps via IPC. Similarly, if an app tries to write password data to the network, a guard can check the write's safety by reasoning about features of the network endpoint (is the destination port associated with unencrypted traffic?), the taint tag's domain (is the destination IP address in the appropriate domain?), or a password's usage history (is the app adhering to the OAuth specification and only sending a password once?). If an action is considered unsafe, then the guard can either block the data from being released, or it can raise an alert.

ScreenPass's primary goal is ensuring that password data is only entered through a trusted keyboard so that it can be tagged before it is given to an app. To achieve this goal, ScreenPass performs dynamic optical character recognition (OCR) on regions of the screen where users expect a software keyboard to appear. If text in this region of the screen is sufficiently similar to the "qwerty" text of a keyboard and the foreground app has not yielded control of the screen to the trusted keyboard, the OS can kill the foreground process or raise an alert.

ScreenPass makes the following contributions:

- Previous secure UIs have restricted where untrusted code can write to the screen [29, 55], but ScreenPass is the first system designed specifically for the limited screen real estate of a mobile device; ScreenPass protects sensitive input by restricting what untrusted code may write to sensitive parts of the screen.
- ScreenPass is the first system to prevent UI spoofing through efficient and robust online computer vision. Software-only computer-vision techniques such as OCR minimize ScreenPass's hardware requirements and allow our approach to generalize to any modern touchscreen device.
- We have implemented a ScreenPass prototype for Android and evaluated its robustness to attack as well as its energy and performance overheads. Our attack study found that ScreenPass is robust to a wide range of static and dynamic attacks while generating zero false positives. ScreenPass only failed to detect spoofed keyboards with noisy backgrounds that look significantly different than a standard system keyboard. ScreenPass consumes no additional energy when the screen is not updated or tapped. Under a typical configuration, ScreenPass introduces 12% energy overhead in the worst case. It consumes far less for apps that infrequently update the screen or require little user input. ScreenPass also had negligible impact on interactivity; under

a typical configuration, no workload experienced a statistically significant drop in frame rate under ScreenPass.

- We have also conducted two in-situ user studies with ScreenPass-like software keyboards. Our initial 18-user, three-week user study showed that tagging passwords imposes little additional burden on users, and showed that users will tag passwords at a high rate when prompted. A smaller, follow-up study demonstrated that integrating a password manager with ScreenPass incentivizes users to tag their passwords at a high rate even when they are not prompted.

The rest of this chapter is organized as follows: Section 3.2 describes ScreenPass's trust and threat model, Section 3.3 provides an overview of ScreenPass's approach to securing passwords on mobile devices, Section 3.4 describes the design and implementation of ScreenPass, Section 3.5 describes our evaluation, Section 3.6 gives our conclusions.

3.2 Trust and Threat Model

To ensure fractal behavior, Fractal Coherence requires a hierarchical logical structure. However, Fractal Coherence does not place any requirements on the physical topology of the system. The hierarchical logical structure can be implemented on any kind of physical topology, such as a 2D mesh, torus, ring, etc. Hereafter, when we refer

to a system's structure, we are referring to its logical structure. In this thesis, we confine our discussion to the tree structure with a consistent degree at each level, but we believe our methodology can also apply to other hierarchical logical structures.

We assume that a user trusts the mobile platform running on her device, and relies on the operating system's existing mechanisms to thwart attacks against the platform itself. We assume that the trusted computing base (TCB) consists of any code that sits behind the standard set of APIs on which a mobile app is implemented.

For example, the essential components of Android's security model are a Linux kernel, user-space daemons called services running under privileged UIDs, and an IPC mechanism called Binder. Each Android app is signed by its developer and runs as a Linux process with its own unique, unprivileged UID. Apps access protected resources such as the software keyboard through Binder IPC calls. Apps and services can limit interactions with other code by specifying access-control policies to the Binder dispatcher.

An information-flow monitor such as TaintDroid cannot track important data unless it is properly tagged and cannot protect tagged data without release policies. However, even though these tags can allow a monitor to provide stronger security for a user's passwords than is possible today, some classes of attacks are difficult or impossible to prevent with existing monitors.

For example, like many monitors, TaintDroid does not prevent data from leaking through covert channels such as a program's control flow or timing information. Recent work on selectively tracking implicit flows using information from the symbolic execution of a program is promising [32], and such techniques may be applicable to existing monitors. We consider attacks against the taint-tracker to be outside the scope of ScreenPass; our goal is to ensure that password input is always tagged. Furthermore, ScreenPass is not designed to prevent passwords from leaking via covert channels like a device's motion sensors [24, 40]. These vulnerabilities have straightforward solutions, such as disabling access to motion sensors whenever a password is input.

ScreenPass's taint tags associate data with a coarse-grained domain. As a result, ScreenPass alone cannot prevent attacks in which passwords are leaked within a domain. For example, untrusted code could log into a service using credentials that are hardcoded into the app binary or accessed from an external server. Once logged in, the untrusted code could leak another user's password through the original account (e.g., by writing the user's password on an attacker's Facebook wall). Alternatively, an untrusted app could encode a user's Facebook password as a world-readable message on the user's own wall, wait for an external machine to read the post, and then delete the post before the user noticed anything strange. These attacks are challenging, but allowing a system to monitor which domains have access to a password make it more likely that

problems can be detected than if passwords continue to be shared without restriction. For example, many services maintain detailed records of all user actions. If a service detects a same-domain attack on mobile users' passwords, then it may be able to use its logs and identify how many other users were affected.

Finally, this work assumes that passwords are input through a "qwerty" English keyboard, though we believe that our techniques can be generalized to more specialized keyboards, such as numeric keyboards or non-English keyboards. We assume that users will not trust apps that require passwords to be input through a keyboard with a non-standard layout or an unusual set of keys.

3.3 Approach Overview

ScreenPass must ensure that the TCB intercepts all password input so that it can be tagged and tracked by an information-flow monitor like TaintDroid. The original TaintDroid prototype [27] tracks sensitive data that is accessed through a small number of

API calls with well-defined semantics. By interposing on these taint sources, TaintDroid can tag several important classes of sensitive data before they are accessed by untrusted code. Once this data has been released, TaintDroid tracks its propagation by integrating dynamic taint analysis (taint-tracking) into Android's Dalvik VM, native system libraries, file system, and Binder IPC mechanism. Tagging sensitive data as it

enters a program by interposing on API calls with well-defined semantics is sufficient for many classes of sensitive data (e.g., a device's location and IMEI number), but this approach is not a good fit for passwords.

Apps access passwords through character-stream interfaces that do not distinguish between sensitive and non-sensitive data. Previous approaches to identifying password inputs have required users to explicitly identify their passwords through a secure attention sequence (SAS) [41, 47]. An SAS is a well-known sequence of characters (e.g., "@@") that labels bytes in the stream as password data. As long as trusted code can identify individual characters, it can look for the SAS and tag password data. Unfortunately, on touchscreen devices with software keyboards, untrusted apps can circumvent the SAS by hiding text input from the TCB.

Software keyboards translate touchscreen gestures to characters by correlating the screen location where a user tapped or gestured with what was displayed at that location. Well-behaved apps allow trusted code to map gestures to characters by invoking the platform's standard software keyboard. When invoked, the trusted software keyboard assumes control of the lower half of the screen, where it displays a virtual keypad. The keypad receives the coordinates of taps from the touchscreen driver, translates those inputs into characters, and returns the characters to the app. In addition, well-behaved apps using a platform's standard library of widgets can put text-input

boxes into "password" mode by manipulating its attributes. For example, Android apps can set the "android:password" attribute of a text-input widget to explicitly notify the TCB when input to the text box is a password.

Unfortunately, touchscreen platforms cannot force apps to yield control of the screen to the trusted UI. An untrusted app can bypass the trusted keyboard by retaining control of the screen, displaying a keypad that is visually indistinguishable from the trusted one, and translating touchscreen gestures to characters by itself.

Under such a spoofing attack, trusted code such as the touchscreen driver and UI manager would only see taps and swipes on the screen and cannot interpret the semantics of those gestures. Trusted code would have no way of knowing that a user's taps and swipes were meant to be text inputs, and a user would have no way of knowing that the TCB was oblivious to her password input.

To address these challenges, we developed ScreenPass while keeping the following design considerations in mind:

Minimize hardware assumptions. ScreenPass should work on as many touchscreen devices as possible. There are hundreds of touchscreen devices with a wide range of hardware configurations. However, because of the minimalist industrial design of modern smartphones we conservatively assume that all interactions with a user occur through the touchscreen. Furthermore, ScreenPass should not re-purpose already

overloaded inputs such as a home or power button for an SAS, or rely on a non-touchscreen output such an external LED to signal the user.

Maximize display utilization. Screen real estate is a precious resource for mobile apps. Prior approaches to secure UIs [29, 55] have reserved part of the screen for messages from the TCB to the user, but this is inappropriate for consumer touchscreen devices. Android, iOS, and other mobile platforms provide a bar at the top of the screen for system information and notifications, but allow apps such as games, media players, and web browsers to hide this bar when in full-screen mode. To preserve the functionality of these important apps, ScreenPass should also support full-screen mode.

Minimize users' responsibilities. ScreenPass should avoid burdening users' with easy-to-ignore and error-prone new responsibilities. For example, ScreenPass could display a secret image, known only to ScreenPass and the user whenever it detected password input. If the image was absent during password entry, a user would be expected to notice and not provide her password. We did not pursue this approach because (1) ScreenPass would have to rely on users to choose images that could not be guessed by untrusted code, and (2) the secret image would have to be managed through additional layers of UI (e.g., for setting and resetting), each of which would need to be secured.

Furthermore, Schechter's 2007 study of site-authentication images (SAIs) found that these secrets provide very little security in practice [49]. This study found that 92% of participants logged in to a banking site when their secret image was replaced with a generic maintenance message. Schechter's study also found that 100% of participants logged in when the secure-connection indicator (i.e., a red "https" string) was absent. Other studies of web-based systems have found that relying on users to heed warnings or notice the absence of a visual signal are often ineffective [25, 26, 57].

The main observation behind our approach to securing password input is that a keyboard consists of a unique and predictable sequence of characters. Thus, ScreenPass uses efficient optical character recognition (OCR) to search for text in the portion of the screen where a user would expect a keyboard (i.e., the lower half). If the OCR engine detects character sequences that are close to those of a keyboard (e.g., "qwerty" or "qvvrty"), it checks whether the trusted software keyboard has been invoked. If not, ScreenPass can take a variety of actions, such as killing the foreground process or warning the user.

3.4 ScreenPass

ScreenPass is a realization of the approach to secure password entry for Android touchscreen devices outlined in Section 3.3.

3.4.1 Capturing and Tainting Password

Figure 1 shows a high-level overview of how ScreenPass captures and tags a user's password. Arrows in the diagram represent IPC calls across components. All components in the figure are trusted except for the untrusted app in gray.

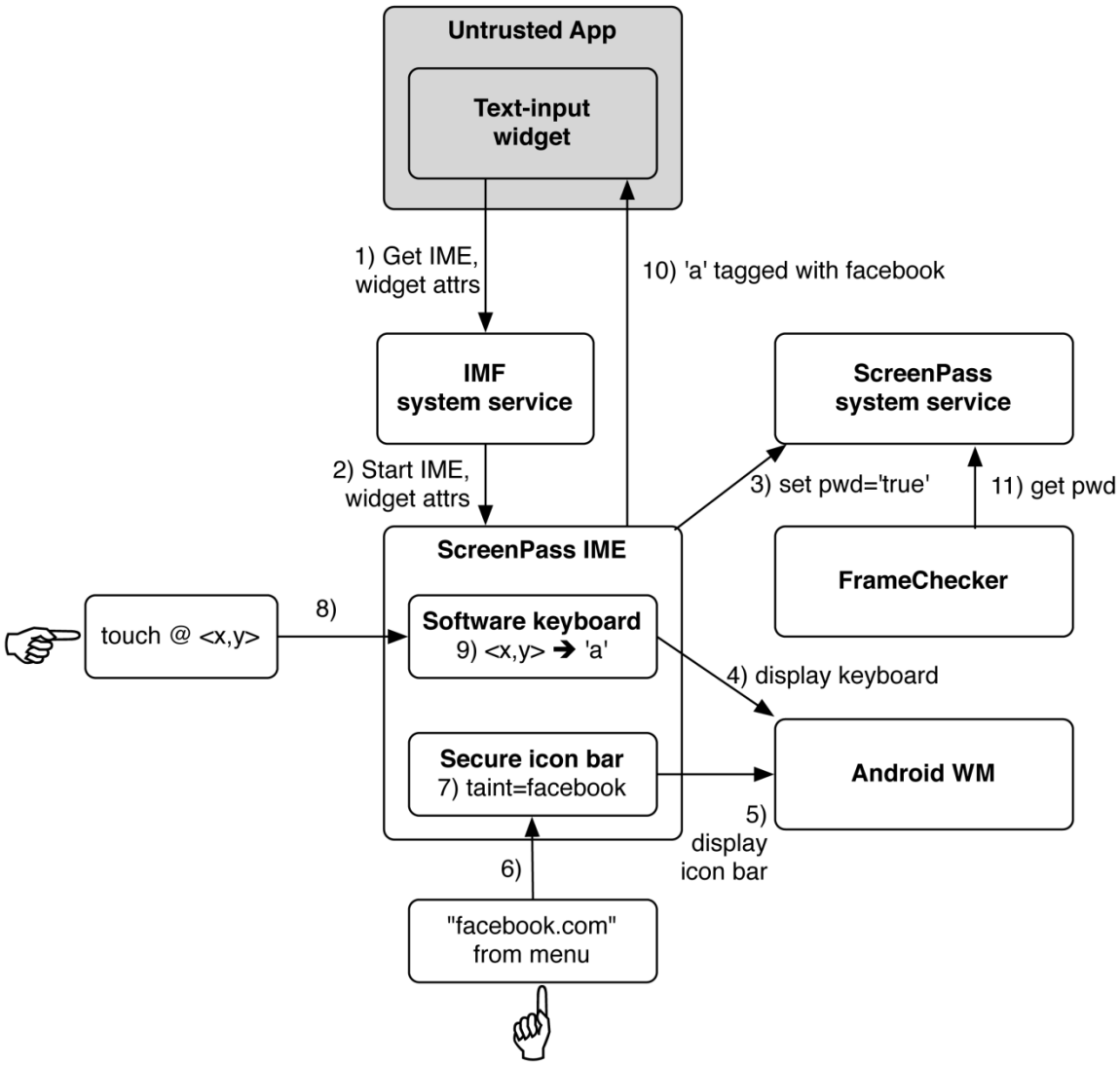

Figure 1: Overview of the ScreenPass architecture.

The first step in tagging a user's password is for the untrusted app to request an IME from the IMF system service. The request contains attributes describing the target widget so that the software keyboard can be appropriately configured. For example, if a text field is an email address, the software keyboard can display an "@" key along with the usual alphabetical keys. The IMF assigns the app to the ScreenPass IME and passes along the target widget's attributes. The IME checks these attributes to see if the widget is in password mode.

If a widget says that it is in password mode, the secure keyboard first contacts the ScreenPass system service to put the device in password mode. The ScreenPass system service acts as a central repository for ScreenPass state. IPCs to access this state can be controlled using Android's code-signing framework. Only code that has been signed by a trusted entity is allowed to communicate with the ScreenPass system service.

A malicious app could claim that the target widget was not in password mode, while still obfuscating text input as a user would expect of a password field. If this happens, the ScreenPass IME will not prompt the user to tag their password, and the user must either (1) avoid entering their password, or (2) use an area at the bottom of the secure keyboard to actively tag their password and put the device in password mode. We call this a silent-widget attack. We will return to this attack shortly.

Once the device is in password mode, the keyboard makes a request to the Android Window Manager to display the secure keyboard and domain chooser. Figure 2 shows the software keyboard and chooser. The tag list allows a user to associate an administrative domain with her password. This association must be set before a user inputs her password or else characters may not be properly tagged. As a result, the software keyboard ignores input until the user has chosen a tag, and the secure keyboard immediately prompts the user for a domain when triggered. Figure 2 shows the prompt a user sees after the keyboard is displayed. ScreenPass uses an integrated password manger to incentivize users to tag their passwords when they are not explicitly prompted (e.g., during a silent-widget attack).

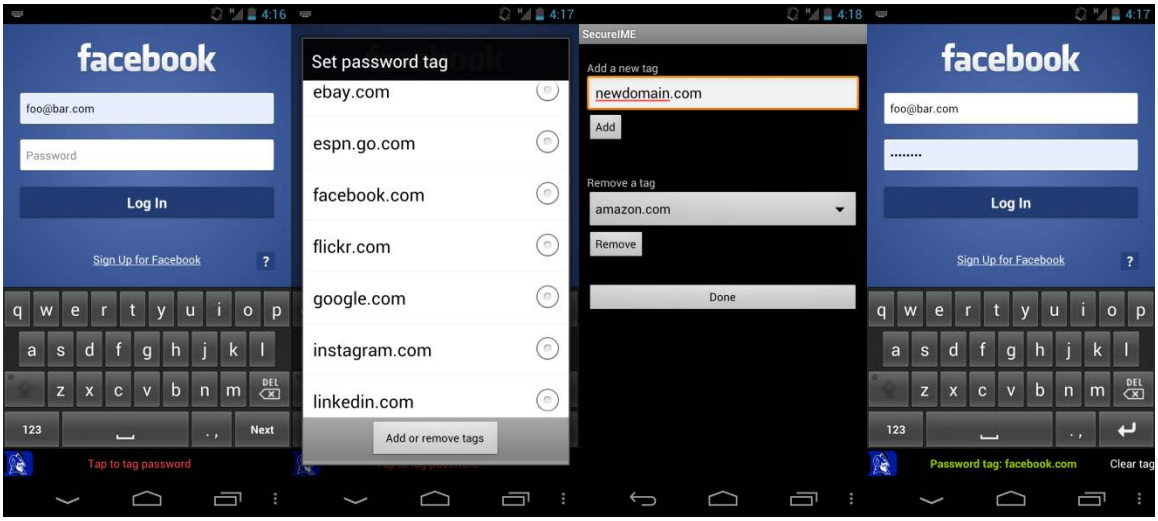

Figure 2: When a user taps on a password field, ScreenPass's secure keyboard prompts the user to tag their password, and then applies the tag to each character using TaintDroid.

Once the user has chosen a domain, the keyboard looks up the associated taint tag and applies the tag to all subsequent character inputs (whether input by the user using the keypad or loaded from the password database). It should be noted that once a user has chosen a tag, there is no way to change the tags that have been previously applied. However, our prototype allows users to choose from a preset list of domains or to define a new tag.

As described earlier, TaintDroid's taint tags are 32-bit bit vectors. The original TaintDroid design treats tags as bit vectors so that data derived from multiple sensitive sources can be represented. However, this approach leaves ScreenPass with too few bits to represent all of the services a user might use. As a result, ScreenPass reserves the highest 10 bits of each tag and treats those bits as a 10-bit namespace. Passwords do not need to be associated with multiple domains, and 1023 domains should be sufficient for most users.

When a user taps on the secure keyboard, it maps the coordinates of the taps to characters, and uses TaintDroid to tag the IPC message back to the untrusted app that contains password data. Once the untrusted app receives the tainted IPC message, TaintDroid propagates its tags as the app uses the password. When the app attempts to send a password over the network, to the file system, or over an IPC channel, TaintDroid inspects the buffer's tags and can enforce a system-defined policy.

3.4.2 Prevent Spoofing Attacks

The FrameChecker is responsible for detecting attempts to spoof the software keyboard. The FrameChecker is a thread running within the SF. The SF's main thread composes the system's surfaces together and posts the resulting frame to the frame buffer's (FB's) front buffer. Implementing the FrameChecker as a separate thread keeps spoof detection out of the critical path of the SF's main work.

At a high-level, a typical working cycle of the FrameChecker consists of three stages: (1) capturing the screen, (2) performing OCR to detect software keyboards, and (3) sleeping for a random, short amount of time. The random sleep time helps ScreenPass reduce its energy overhead to an acceptable level. As we will show in Section 3.5, performing OCR without resting could quickly drain a user's battery under certain workloads. Our current implementation sleeps for a randomly chosen time between 0ms and 1,000ms; that is, the expected sleep time is 500ms.

3.4.2.1 Screen Capture

Upon waking up, the FrameChecker first checks with the ScreenPass system service to see if the device is in password mode, if the touchscreen has not been tapped, or if the screen has not changed since it went to sleep. If any of these conditions are true, then the FrameChecker does not need to analyze the screen and can go back to sleep. The SF maintains information about which regions of the screen are dirty, which allows

the FrameChecker to avoid inspecting the FB if it has not changed since the last scan. In addition, a thread runs alongside the FrameChecker to detect touchscreen inputs. It detects all touch events by polling the device file /dev/input/event0 and maintains a status variable indicating whether a touch event occurred. After the FrameChecker wakes up, it checks this status variable.

The vast majority of the time, the screen will not have changed. Furthermore, only analyzing the screen while a user is interacting with it avoids performing analysis when the screen is updating but the user cannot be inputting her password. This is common for games that are primarily controlled by a device's motion sensors and for watching videos.

If the FrameChecker finds that the device is not in password mode, that the screen has been updated, and that the user is interacting with it, then the FrameChecker must perform OCR. However, taking a single instantaneous screen capture would leave ScreenPass vulnerable to dynamic attacks. Under a dynamic attack, malicious code rapidly alternates between frames of a partial keyboard so that a user viewing the screen sees a complete keyboard, but no single frame contains one. To combat this attack, another thread in the SF computes a "squashed image" composed of all instantaneous screen captures between FrameChecker requests. The squashed image contains the

average pixel values over a period of time, and, as we will see in Section 3.5, closely approximates what a user perceives under a dynamic attack.

To reduce the computational load of computing the squashed image, ScreenPass takes two steps. First, we only target the bottom two-fifths of the screen where a keyboard may appear. Our Nexus S prototype has a screen resolution of 800 480 pixels, and our squashed image is 320 480 pixels. This not only reduces the effort needed to compute the squashed image, but the work of the OCR engine as well. Second, we used the NEON 128-bit SIMD instruction available on the Nexus S's ARM processor. This instruction allows us to add up to 16 8-bit integers in a single instruction, and greatly increases the efficiency of computing a squashed image.

One danger of computing squashed images is that, for periods containing a transition from a non-keyboard scene to a keyboard, the averages will be polluted by pre-keyboard pixels. However, as we will see in Section 3.5.2 that users type passwords much more slowly than ScreenPass checks the screen. If a user starts to type her password, ScreenPass will have computed a "clean" squashed image well before she finishes.

3.4.2.2 OCR Analysis

The FrameChecker analyzes screen content using OCR. Our current implementation uses the well-known TesseractOCR package [53, 54, 56]. OCR converts

images of text into machine-encoded characters, and is widely used to digitize a number of paper-based data sources including books, documents, receipts, and checks. A typical OCR process integrates techniques from computer vision, pattern recognition, and artificial intelligence. Usually, the steps include locating and segmenting the characters from input images, preprocessing the images to remove noise, extracting patterns from the characters for classifiers, organizing the identified characters to reconstruct original words, and postprocessing to correct OCR errors by checking the context.

One of the advantages of using OCR to detect keyboards is that the analysis does not have to be completely accurate. The FrameChecker only needs to identify fragments of text that are sufficiently similar to the character sequences expected of a keyboard. Furthermore, attackers have no room to alter the character sequences on a spoofed keyboard, since a user will become instantly suspicious of a keyboard in which the keys have been moved around. A keyboard is usually divided into three areas: the characters, the keys, and the background. The keys cover the majority of this area. To improve character identification, we apply preprocessing to the screen captures to highlight the characters and reconcile the color of the key and background. The keys cover more pixels than the characters and the background, so we can determine the color of the keys by identifying the peak color for all pixels. Then we refill the background with the same color of the keys so that the characters are highlighted against a pure color background.

The preprocessing algorithm is fast and traverses all pixels twice. In the first traversal, we count the number of pixels in gray scale and identify the color for the peak. Then we change the color of the background pixels in the second traversal.

It is also worth pointing out that TesseractOCR itself can tolerate some color differences between the keys and background. We apply preprocessing only to handle some extreme cases, such as a black background and white keys, and to better facilitate character recognition. Finally, we use the training data for the English language from the TesseractOCR official site. When the FrameChecker starts, it loads the training data and initiates the OCR engine.

3.5 Evaluation

To evaluate ScreenPass, we sought answers to the following questions:

- How robust is ScreenPass to spoofing attacks?
- What is the perceived burden of tagging passwords?
- How likely are users to tag their passwords?
- How often should ScreenPass check for spoofing attacks?
- What is the performance and energy overhead of ScreenPass?
- Can taint-tracking detect when apps mishandle passwords?

To answer the first question, we subjected our ScreenPass prototype to a variety of attack keyboards. To answer the next two questions, we performed two user studies.

The first was a three-week, in-situ user study with 18 participants; the second was a one-week, follow-up study with eight of the initial participants. To answer the next two questions, we measured the performance and energy characteristics of our prototype using a mix of representative apps. To answer the final question, we ran 31 Android apps (30 benign and one malicious) on our ScreenPass prototype. Our prototype implements ScreenPass on top of Android 2.3.4, uses TaintDroid for taint-tracking, and runs on an HTC Nexus S smartphone.

3.5.1 Robustness to Spoofing Attacks

To measure ScreenPass's robustness to various kinds of spoofed keyboards, we created 12 static attacks and four dynamic attacks.

For the static attacks, we applied the following modifications classes to the stock Android keyboard: transformations, color changes, and special effects. Each class can be applied to the keyboard's background, its characters, or both. Figure 3 shows the upper-left corner of each static attack keyboard used in our experiments with the character sequence "q w e r t y".

Transformation: To generate the transformed attack keyboards, we altered the shape of each character in the stock keyboard. These transformations included changing the font, italicizing the font, altering the font width, rotating characters, and warping characters.

Figure 3: Static attack images tested: a. Scripted typeface, b. Italicized typeface, c. Narrowed typeface, d. Widened typeface, e. Rotated characters, f. Warped characters, g. Colored background, h. Colored characters, i. Blurred background, j. Blurred characters, k. Shiny characters, l. Random noise.

Color: Color attacks altered the background or character color of the stock keyboard.

Effects: We also created attack keyboards by applying special effects to parts of the stock keyboard, such as a blurred background, blurred characters, "shiny" characters, and adding random noise.

If ScreenPass only performed OCR on individual frames it would be vulnerable to dynamic attacks in which a malicious app rapidly alternated partially complete frames. ScreenPass's squashed images smooth out visual differences between frames over a period of time to approximate what a user sees.

Figure 4: Sample frame from a dynamic-attack keyboard that ScreenPass could not detect by analyzing individual frames. ScreenPass detected the attack by analyzing a squashed image.

For our dynamic-attack experiments, we were interested in knowing (1) how incomplete frames needed to be before analyzing individual frames became insufficient, and (2) how robust our squashed image approach was to these attacks. Each frame in our dynamic attack keyboards consisted of a checkerboard pattern of alternating empty blocks and keyboard-image blocks. Consecutive frames in an attack sequence swapped empty and keyboard-image blocks, so that when alternating frames were displayed at 30 FPS, a user saw a legitimate keyboard on the screen.

When the alternating blocks were only 1 X 1 pixel, OCR still detected the keyboard on individual frames. However, when blocks were 4 X 4 and 8 X 8 pixels large, OCR failed on individual frames but succeeded on the squashed image. The only dynamic attack that ScreenPass could not defeat was a keyboard with a flowery background. Figure 4 shows an example frame from the attack.

The reason that ScreenPass failed to detect this keyboard is that the characters on the keyboard blended in with the noisy background in our squashed image. Nonetheless, this attack keyboard was clearly non-standard and would be easy for users to detect.

3.5.2 User Studies

To better understand ScreenPass's usability, we designed two insitu user studies. In both studies, participants were asked to (1) replace the software keyboard of their personal Android smartphone with a ScreenPass-like study keyboard, and (2) use their smartphones to complete a series of online surveys.

3.5.2.1 Study Designs

Both study keyboards provided a high-fidelity simulation of using ScreenPass. The keyboards' default behavior was to immediately prompt the user to tag their password when the keyboard received a password-input request from an app. The prompt included a list of alphabetically-ordered pre-loaded tags, as well as an option to

add a new tag. As with ScreenPass, the keyboard allowed users to tag any text input, even in the absence of an explicit prompt. The keyboard also included hooks for remotely turning explicit prompting on and off. The main difference between the keyboards used in our initial study and our follow-up study was that the initial-study keyboard did not include a password manager, whereas the keyboard in the follow-up study did. In both studies, each participant was instructed to disable the "password remembering" feature of their phone's web browser and to set the study keyboard as their phone's default input method. Each time an app requested password input the keyboard uploaded a time-stamped record with the following information:

- whether the user was prompted to tag their password
- whether the user tagged their password
- a cryptographic hash of the tag (i.e., a domain name)
- a timestamp of when the keyboard received the password-input request
- a timestamp of when the keyboard displayed the tag prompt
- a timestamp of when the tag prompt stopped being displayed
- a timestamp of when the keyboard stopped being displayed

This data allowed us to measure how often users tagged their passwords (with and without prompting), how many unique tags a user used, how long users took to tag their passwords, and how long users took to type in their passwords. In the initial study,

the keyboard did not log individual keystrokes or unhashed tags. In the follow-up study, the keyboard stored passwords to a local database so that they could be loaded automatically at a later time.

In addition to logging this data, we also asked participants to complete online surveys on their smartphones, as well an initial demographic and exit questionnaire on a PC. Surveys could only be accessed after logging in with a username and password, which increased the number of study-keyboard logins we could observe. In the initial study, we asked participants to complete three surveys, and in the follow-up study, we asked participants to complete two surveys.

Each of the online surveys (i.e., pre-study, mid-study, and poststudy for the initial study and pre-study and post-study for the follow-up study) consisted of three questions. Each question was presented as a statement, and users were instructed to select their level of agreement with each statement on a scale from one ("Strongly disagree") to seven ("Strongly agree").

The first two survey statements were "I worry that websites and mobile apps may steal my passwords" and "Before I log into a website, I make sure that the connection is secure." These statements were intended to gauge a user's awareness of security threats, and were identical in all pre-study, mid-study, and post-study surveys.

The third statement asked participants to indicate how difficult they felt logging into apps and websites on a smartphone is. For the pre-study surveys, we presented the statement "Before beginning the study, logging into websites and mobile apps on my smartphone was difficult." For the mid-study and post-study surveys, we changed the start of the statement from "Before beginning the study" to "Since beginning the study", while leaving the remaining wording the same. The change in phrasing allowed us to compare the perceived difficulty of inputting passwords on smartphones with and without the study keyboard.

For the first 20 days of the initial study, users were always prompted to tag their passwords. However, we turned off prompting before releasing the post-study survey. Because users were instructed to complete the post-study survey from their smartphones, turning off prompting at the end gave us a sense of how users would react to a silent-widget attack after 20 days of experience with the study keyboard. We similarly turned off prompting after a training phase in our follow-up study.

3.5.2.2 Recruitment and Training

After receiving approval for our initial study from the university Internal Review Board (IRB), we recruited candidate participants by posting a call for participation on Facebook and Google Plus, and by sending emails to several university mailing lists. Interested users were instructed to complete a demographic questionnaire on our study

website. As an incentive to participate, we offered each participant a $20 Amazon Gift Card, to be given after completing all study tasks.

46 individuals registered and completed our demographic questionnaire. We selected 20 participants from these candidates. We rejected candidates who used a non-Android smartphone or lived too far from campus to sign a consent form in person. From our initial set of 20 participants, two did not install the keyboard in a timely manner, and we have excluded them from our results. Thus, our reported results from the initial study are from 18 participants who installed our study keyboard on their personal Android smartphone for a period of three weeks during November and December, 2012.

From the initial demographic questionnaire, the average age of the 18 participants was 24 years old. Participants had owned an Android smartphone for an average of 1.5 years, spent an average of two hours each day browsing the Internet on their phone, and spent an average of five hours each day using the Internet on a PC. 15 of the participants were undergraduate or graduate students, one was a professor, one was a doctor, and one worked for local technology company. Four participants were female. After notifying participants that they had been selected for the study, each signed a consent form in person. After signing the consent form, we installed the study keyboard on their personal Android smartphone and set it as the default keyboard. We

also disabled the remembering-passwords option in their web browser settings. Lastly, we showed each participant how to tag their passwords and how to add a new password tag. The list of pre-loaded tags did not include one for the study website. We repeated this setup procedure in our follow-up study.

3.5.2.3 Results and analysis

In analyzing our study results, we were primarily interested in characterizing the qualitative and quantitative burden of using ScreenPass, and users' willingness to tag passwords with and without explicit prompting.

Tagging burden

Figure 5 shows the results of our pre-study, mid-study, and poststudy surveys for the initial study. These graphs have two noteworthy features. First, the median level of worry about stolen passwords and the median level of diligence about checking for SSL remained constant throughout the study. It is hard to draw any conclusions from the slight fluctuations of the 25th and 75th percentile responses to the first two survey questions. These results show that, in general, our participants thought of themselves as being fairly security conscious, and that this self-perception did not change over the course of the initial study.

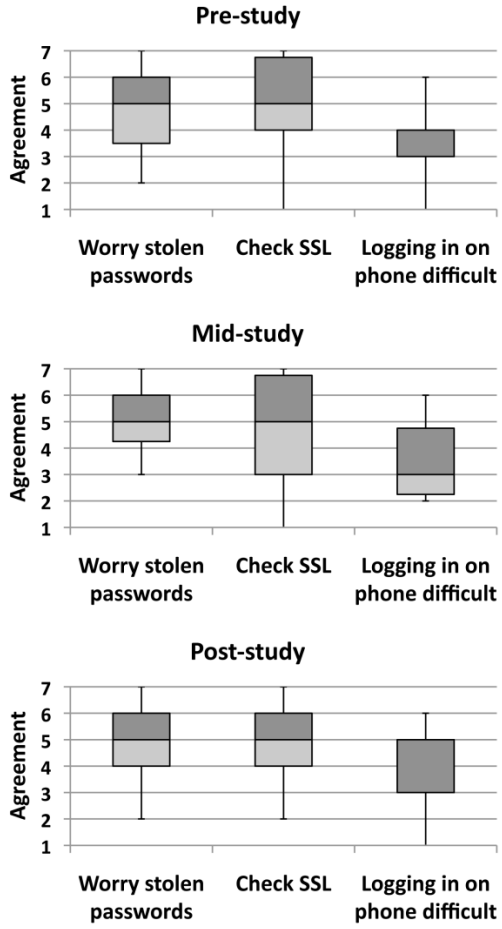

Figure 5: At the beginning, middle, and end of our initial user study, participants were asked to rate how much they agreed with statements that (1) they are worried that malware will steal their passwords, (2) they make sure SSL is enabled before logging into a website, and (3) logging into apps and websites on a smartphone is difficult. Pre-study responses reflect users' experiences before participating in the study. Mid-study and post-study responses reflect users' experiences during the study. The agreement scale ranges from one ("Strongly disagree") to seven ("Strongly agree"). The top edge of each dark-gray box represents the 75th percentile response, the bottom edge of each dark-gray box represents the median response, and the bottom edge of the light-gray box represents the 25th percentile response. The top whisker extends to the maximum response, and the bottom whisker extends to the minimum response. Note that an absent light-gray box indicates that the 25th percentile response was equal to the median response.

The median reported difficulty of logging into apps on a smartphone also remained constant throughout the initial study. However, the 75th percentile response rose slightly in the mid-study and post-study surveys of the initial study, which cover the period when users were asked to tag their passwords with the study keyboard. This indicates that tagging a password imposed only a minor additional burden on users beyond the existing inconveniences of typing in a password on a smartphone.

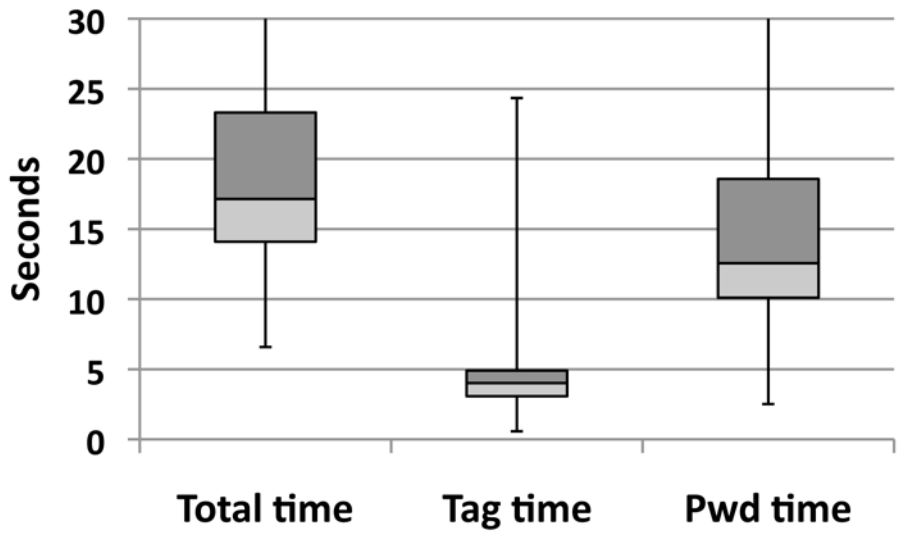

Figure 6: When a participant tapped on a password field, we recorded the total time the keyboard was displayed ("Total time"), the time spent tagging the input ("Tag time"), and the time spent typing in a password ("Pwd time"). The top edge of each dark-gray box represents the 75th percentile time, the bottom edge of each dark-gray box represents the median time, and the bottom edge of the light-gray box represents the 25th percentile time. The top whisker extends to the maximum time, and the bottom whisker extends to the minimum time. Note that the top whiskers for "Total time" and "Pwd time" have been cut off to improve readability. The maximum login time was 117 seconds, and the maximum time to enter a password was 116 seconds. These results are for the initial study only.

The timing numbers collected whenever participants input a password are consistent with our survey results. Figure 6 shows that during the initial study the median time to tag and enter a password was 23.3 seconds, the median time to tag a password was 4.9 seconds, and the median time to type in a password was 18.6 seconds. Note that the median tag and password values do not sum to the median total value because the median total time was taken from a different event than the median tagging time and median password-entry time.

The slowness of inputting passwords on a smartphone has three implications for ScreenPass. The first is that the time to tag and type in a password is dominated by typing. Thus, it is not surprising that users did not report a significant increase in the burden of logging into apps during the study. The second implication is that integrating a system-wide password manager into ScreenPass (as we did in our follow-up study) should make logging in significantly faster. Integration would essentially make entering a password as fast as tagging; a user would only have to choose a tag, and the keyboard would send the password characters to the requesting app.

Finally, even with an integrated password manager, ScreenPass users will occasionally have to input new passwords. However, the long latency of entering a password gives ScreenPass a large window for detecting spoofing attacks. This large

window affords ScreenPass the opportunity to safely throttle the frequency with which it performs OCR. We will return to this issue in Section 3.5.3.

Willingness to tag

During the initial study we recorded data from 134 logins. The average number of logins per user was 7.2 (with a maximum of 20), and the median number of unique tags per user was 2.5 (with a maximum of 6). Despite our instructions, some participants did not complete the surveys on their smartphones, although most users did as we asked.

Figure 7 shows how frequently participants tagged passwords over the course of the initial study. During the first period of this study, beginning from the start of the study until the time when the first mid-study survey had been completed, participants tagged their passwords 83% of the time (i.e., 49 out of 59 passwords were tagged). During the second period of the study, beginning from completion of the first mid-study survey until the first post-study survey was completed, participants tagged their passwords 89% of the time (i.e., 42 out of 47 passwords were tagged). We suspect that the rise in tagging rate is due to users becoming more comfortable with the study keyboard. Thus, the overall tagging rate for prompted logins was 86% (i.e., 91 out of 106 passwords were tagged). However, disabling prompting during the final period of the initial study had a major impact on the observed tagging rate. With prompting off and

without a password manager, only 25% of passwords were tagged (i.e., 7 out of 28 passwords).

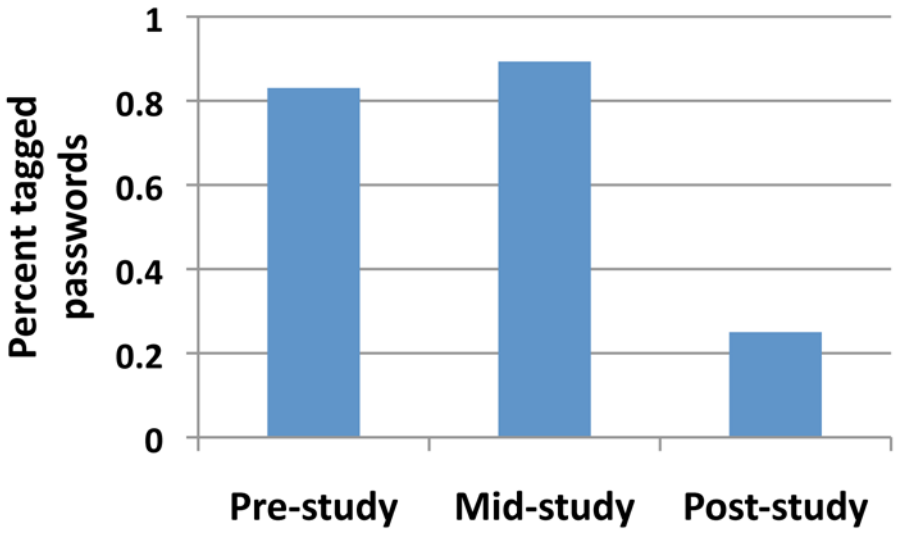

Figure 7: The first bar ("Pre-study") depicts data for the period starting with the beginning of the study and ending before the first mid-study survey was completed. The second bar ("Midstudy") depicts data for the period starting with the completion of the first mid-study survey and ending before the first post-study survey was completed. The third bar ("Post-study") shows data for the period beginning with the completion of the first post-study survey and ending with the end of the study. During the pre-study and mid-study periods, users were immediately prompted to tag their password. During post-study period, users were not prompted to tag their passwords. These results are for the initial study only.

The low tagging rate for unprompted logins demonstrated that ScreenPass must give users a strong reason to tag their passwords without being prompted. If not, users will likely fall victim to silent-widget attacks. Thankfully, as mentioned earlier, integrating a password manager into ScreenPass provides precisely such a reason. In a silent-widget attack, users still interact with the secure keyboard, but are not prompted

to tag their password because a malicious app does not set a password flag in its input request. Telling the secure keyboard to load a password from a ScreenPass maintained database effectively tags the password and prevents the attack.

To test our hypothesis, we ran a small, IRB-approved follow-up study with eight participants from our initial study over one week in April, 2013. We recruited volunteers for the follow-up study by reaching out to the pool of participants from the initial study. Seven of the eight were graduate students, one was a professor, and two were female. Each volunteer received a $20 Amazon gift card for participating.

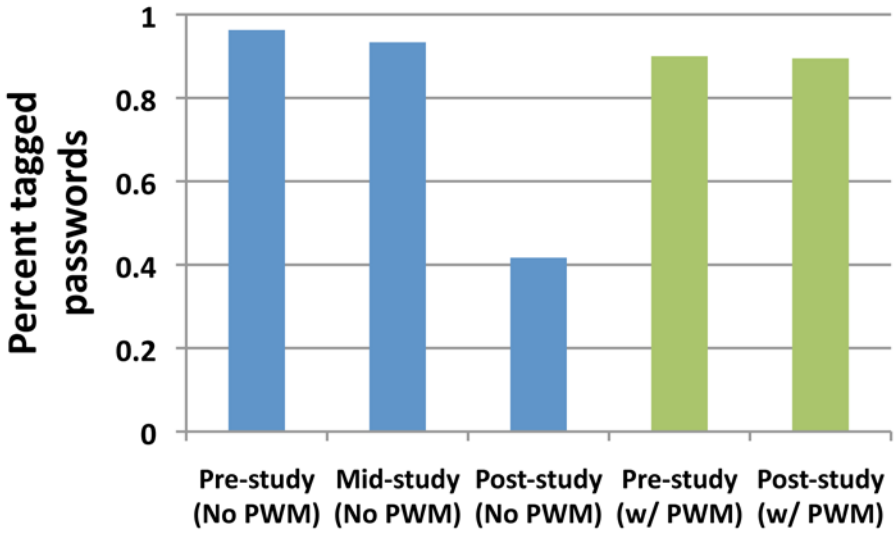

Figure 8: This graph compares the tagging rates for the eight users who participated in our initial user study and our followup user study. The first three bars depict tagging rates for those users during the three phases of the first study, when the study keyboard did not include a password manager ("No PWM"). The last two bars depict the tagging rates for those users during the two phases of the follow-up study, when the study keyboard included a password manager ("w/ PWM"). Explicit prompting was turned off during the "Post-study" phase for both studies.

As before, participants were prompted to tag their passwords during an initial training phase lasting six days, and prompting was turned off during the final phase to simulate a silent-widget attack. However, unlike the initial study, the keyboard in the follow-up study included an integrated password manager that saved passwords the first time they were entered, and automatically loaded a saved password according to the tag specified by the user. In the interest of clarity, we only present the tagging rates from our follow-up study.

Figure 8 shows the tagging rates during the initial and follow-up studies for the eight volunteers who participated in both. First, users who chose to participate in both studies had higher tagging rates than the broader population that only participated in the initial study. For example, these eight participants tagged passwords at rates of 96%, 93%, and 42% during the initial study's pre-study, mid-study, and post-study phases, respectively. Importantly, as with the the larger population in the initial study, the eight who participated in both studies exhibited a sharply lower tagging rate when prompting was turned off.

As Figure 8 shows, the unprompted tagging rate increased dramatically when users were given a keyboard with an integrated password manager. With prompting on during the pre-study phase, users tagged their passwords at a rate of 90% (i.e., 27 out of 30 passwords were tagged). With prompting off during the post-study phase, users

tagged 89% of their passwords (i.e., 17 out of 19 passwords were tagged). This tagging rate is more than double the rate of those same volunteers in the post-study phase of the initial study. Because the sample sizes are relatively small, it is hard to precisely cross-compare the phases of each study. Nonetheless, the trend for unprompted tagging is striking and strongly suggests that integrating a password manager into ScreenPass makes users far less vulnerable to silent-widget attacks.

3.5.3 OCR Performance

Because users react to screen updates slowly, ScreenPass performs OCR periodically rather than on every frame displayed. We expect the system to notify the user of an attack within one second of being drawn on the screen, which should be before she has given the app her password. The time the FrameChecker takes to analyze a frame determines how often it can run. The faster the FrameChecker is, the more frequently it can check for attacks.

To characterize the performance of analyzing a frame, we used a variety of representative app workloads: exploring Android's system-preference menus ("General"), scrolling the App Drawer ("AppDrawer"), playing a YouTube video ("Video"), playing two popular games from the Android Market ("Labyrinth" and "Winds of Steel"), browsing email ("Email"), and reading a PDF document ("PDF Reader"). Table 1 summarizes each app and our interactions with it. We ran each app for

at least one minute and averaged the times to analyze all frames within a run. For these experiments, the FrameChecker did not skip any frames and did not pause between scans.

Table 1: App workloads used for energy and performance experiments.

Workload	App version	Description
General	Android 2.3.4 Settings	Scrolled and browsed through system settings.
App Drawer	Android 2.3.4 App Launcher	Vigorously scrolled through app icons.
Video	YouTube, version 4.4.11	Played the "Fast Five" movie trailer in HD.
Labyrinth	Version 1.5.2	Loaded and started a game.
Winds of Steel	Version 2.2	Loaded and started a game.
Email	Android 2.3.4 email client	Read a sequence of emails from the inbox.
PDF Reader	Adobe Reader, version 10.5.2	Resized and read a research paper (occasionally scrolling to recenter the text).

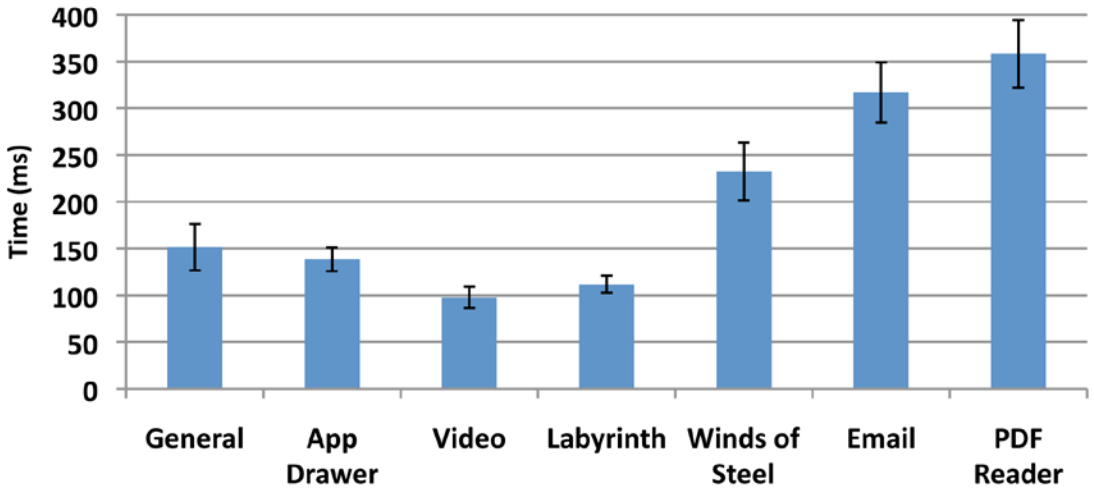

Figure 9: The average time taken by the FrameChecker to scan a single frame.

The results of our experiments are in Figure 9 with error bars representing standard deviations. The high standard deviations we observed while running resource-intensive apps like games were expected, since the FrameChecker thread must contend for the CPU with the app itself. This also explains why analyzing Winds of Steel, which has a high CPU utilization, was relatively slow. None of the other workloads require much CPU time.

The text-heavy workloads of Email and PDF Reader were the slowest to analyze, requiring an average of 317ms and 358ms, respectively. Their high standard deviations are due to the difference between analyzing a frame containing static text, which was slower, and analyzing a (squashed) frame generated while scrolling and zooming, which was faster. For example, during the Email workload, we read a sequence of emails from

Android's stock email client. Scrolling within a message and moving to new message caused text to move and change on the screen, but these events were infrequent relative to the amount of time spent reading static text.

We suspect that the times to scan General and App Drawer frames were also relatively high because of the amount of text displayed on the screen. These numbers indicate that the OCR engine's analysis time is strongly correlated with the amount of text in a frame. Thankfully, as we will see in Section 3.5.4, text-heavy apps require infrequent OCR analysis since they update the screen and are tapped infrequently relative to FrameChecker wake ups.

As noted earlier, experiments with our ScreenPass prototype were run on a Nexus S smartphone, which has a single-core CPU. A multi-core mobile processor would help ease the CPU contention observed with Winds of Steel, and a device with a newer, faster CPU should significantly improve OCR performance. However device screens are also growing in size and pixel density. For example, the Nexus 4's screen has a resolution of 1280 X 768 and contains over two and a half times more pixels than the Nexus S, whose screen resolution is 800 480. It is unclear how well ScreenPass will continue to perform given these competing trends, and we leave an investigation of this question to future work. Finally, we note that for all workloads, our ScreenPass prototype generated zero false positives.

3.5.4 Sampling Rate

Though OCR is fast enough to sample frequently, we would like to choose a sampling rate that best balances the overhead of running ScreenPass and its security guarantees. To characterize the tradeoffs, we configured ScreenPass with three sleep-time ranges: no sleep (0ms), 0 to 500ms (500ms), 0 to 1,000ms (1000ms). Note that the expected rest time for the 500ms configuration is 250ms, and that the expected rest time for the 1000ms configuration is 500ms.

To understand the sampling-rate tradeoff, we used the same apps as before plus another game (PinBall, version 1.3.2), and measured their frame rates and energy consumption. Frame rate captures ScreenPass's impact on device responsiveness. However, because the FrameChecker runs in parallel with an app, ScreenPass could still drain the battery without affecting responsiveness.

3.5.4.1 Frame Rate

UI responsiveness is important for mobile devices and can be captured by measuring an app's frame rate. We should note that Android supports two frame-rendering modes: continuous and render when dirty. In continuous mode, the system continuously renders frames, regardless of whether anything on the screen has changed. In contrast, render when dirty mode only renders frames when on-screen changes occur. Render when dirty mode is not useful for measuring the effect of ScreenPass on frame

rate, and we focused on apps that use Android's continuous rendering mode. For this reason we do not include our frame-rate results for the General, Email, or PDF Reader workloads.

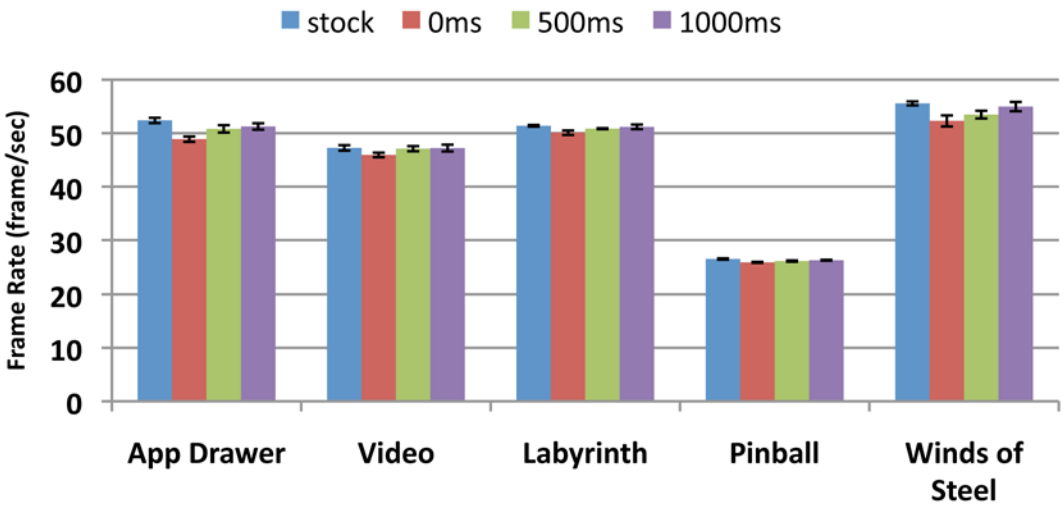

Figure 10: The average frame rates observed while running several applications.

Similar to our OCR-performance experiments, we recorded the frame rates of a variety of apps while running ScreenPass. In particular, we looked at video playback and resource-intensive games, since frame rate is extremely important for these apps. We ran our apps on an unaltered version of Android 2.3 and ScreenPass using three sleep ranges. Each app ran for one minute. We measured the frame rate by periodically logging the frame rate for a small time slice of approximately 250ms throughout the experiment. We then averaged the logged frame rates for that run. Note that for these

experiments, we conservatively analyzed the screen even if the user had not tapped the screen.

Figure 10 shows our results. Continuous analysis has the most significant impact on frame rate due to CPU contention. Most apps experienced little to no slowdown when the FrameChecker slept between 0 and 500ms. Video playback, Pinball, and Labyrinth were all within 1-3 frames/sec of the unaltered system. In the worst cases (App Drawer and Winds of Steel under 0ms), the frame rates dropped by approximately 3.5 frames/sec, but both were close to 50 frames/sec overall. This is a relatively small performance hit that we would not expect to users to notice.

3.5.4.2 Energy

We were also interested in characterizing ScreenPass's effect on energy and battery life. To measure energy overhead we attached a hardware power meter and power source to our Nexus S prototype's battery. We measured current at 5000 Hz and assumed a constant voltage. Using all of the apps from this section under the stock, 0ms, 500ms, and 1000ms configurations, we ran each app continuously for five minutes. To measure energy, we separated the samples into minute-long intervals. Finally, we averaged each interval for a given app and system configuration.

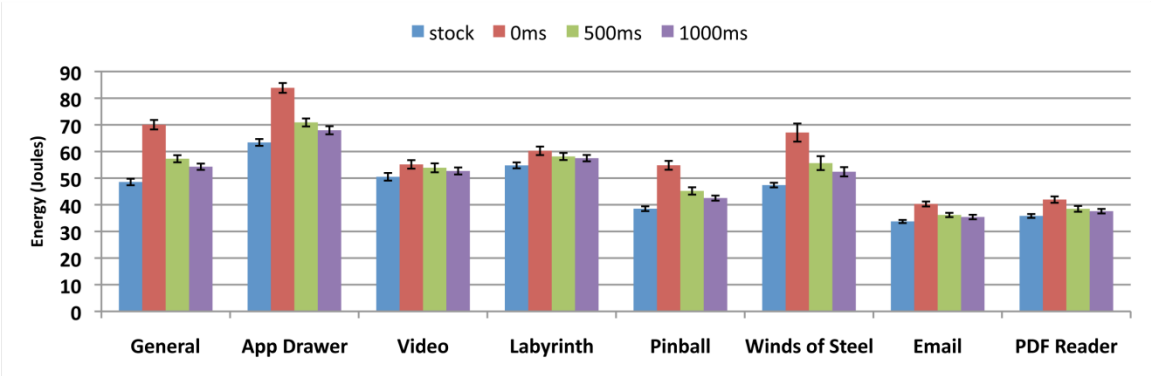

Figure 11: The average energy consumed over one minute while running several applications.

Figure 11 shows our results. The energy overhead of running the FrameChecker continuously is too great. The General workload experienced the greatest energy overhead of 44%, due to its high interactivity and frame-update rate. However, 500ms and 1000ms required far less energy. For 500ms, the maximum overhead was 18% for General, while the interactive games of Pinball and Winds of Steel exhibited overheads of 17%. All other apps were 10% or less under 500ms. For 1000ms, all apps exhibited overheads of less than 12%. Of course, ScreenPass has no impact on either performance or energy when the screen is not updated or when the user does not tap the screen. For example, playing a YouTube video and the game Labyrinth require infrequent user tapping and have low energy overhead across all configurations. This is also true of our two text-heavy workloads, Email and PDF Reader; both imposed 7% overhead for 500ms and 5% overhead for 1000ms.

Recall that in our user study 75% of passwords took 10.1 seconds or longer to enter, and that the minimum time to enter a password was 2.5 seconds. Recall too that under a 500ms configuration, the FrameChecker expects to pause for 250ms between analyses. The FrameChecker expects to pause for 500ms between analyses under a 1000ms configuration. Thus, with a maximum analysis time of around 350ms, either a 500ms or a 1000ms configuration would provide a good balance of safety and energy efficiency; 500ms would provide better safety, and 1000ms would provide better energy efficiency.

Though several in-situ app-usage studies have been published in recent years [23, 28, 42], it is difficult to use these studies to precisely quantify the impact ScreenPass would have on overall battery life. The worst-case workload for ScreenPass is one that is highly interactive and frequently updates the screen, as in the General workload and interactive games.

None of these studies found that users spend much time browsing their phone's settings. However, a study of iPhone users found that social networking apps accounted for 8% of overall app usage, while games accounted for approximately 5% of app usage [42]. A study of Android users found that users have their screen on for an average of 60 minutes each day and that the average game session lasts 114 seconds [23]. An earlier study of Windows Mobile users found a wide range of usage, with screens on between

30 minutes and 500 minutes each day. Among Windows Mobile users, mapping apps and games tended to have the longest sessions, lasting approximately 120 and 110 seconds, respectively, on average [28].

A common conclusion across all three studies is that phone usage can vary significantly from user to user. Thus, we suspect that ScreenPass under a 500ms configuration would scarcely impact light and average phone users, while heavy users may prefer running ScreenPass under a 1000ms configuration. Either configuration would provide strong security against keyboard-spoofing attacks.

3.5.5 App Study

To better understand how existing apps handle passwords, we ran 30 apps from the Android Market and one malicious credentialstealing app from the Android Malware Genome Project [58] under ScreenPass. We are unaware of any malicious apps in the wild that mount keyboard-spoofing or silent-widget attacks.

The non-malicious apps were chosen from a pool of the top 20 free applications in the following categories: finance, communication, education, media and video, shopping, social, comics, and productivity. To broaden our pool we also added the top 20 apps returned after searching for the following keywords: password, auctions, and online games.

From this larger pool, we required apps to be in English, to request a password without requiring the username to be validated first (as most banking apps do), to not use native libraries (since TaintDroid does not support native libraries), and to have been installed at least 100,000 times (most of our apps had more than 250,000 installations). Finally, we tried to create a diverse range of apps by choosing apps in less well-represented categories over apps with more installations in categories that were already represented. All apps in our study either stored a user's password locally in the file system or sent it over the network. Table 2 shows the distribution of apps, organized by how they handle passwords.

Table 2: App types grouped by their password handling (saving to the file system or sending over the network). App categories are shown in parenthesis.

Application Behavior	# of Applications and type
Sent the password through the network	28 (1 business, 1 comics, 1 communication, 1 entertainment, 4 finance, 1 music, 3 productivity, 1 shopping, 13 social, 1 tools, 1 malicious)
Stored the password in the file system	12 (1 entertainment, 1 finance, 3 productivity, 7 social)

Our malicious app was a fake NetFlix app provided by the Android Malware Genome Project. The fake NetFlix app poses as a legitimate NetFlix app and requests a

user's NetFlix login credentials. However, rather than sending a user's password to NetFlix servers, it posts it to the hard-coded URL http://erofolio.no-ip.biz/login.php. We should note that since this host is no longer active, we manually changed the hardcoded URL to one under our control to run the app. The fake NetFlix app is the only app in the Android Malware Genome Project archive that attempts to steal a user's password.

Table 3: Problematic password handling practices in eight of the studied apps. One of the apps stored and sent passwords in plaintext.

Observed Behavior (# of apps)	Details
Password to third party servers (4)	Fake NetFlix, one Financial, and two Social sent the password for the requested services to the application developer's server.
Password through the network in plaintext (4)	Fake NetFlix, one Entertainment, and two Social sent the password in plaintext.
Password stored in plaintext (4)	Two Entertainment and two Social stored the password in the phone's local storage in plaintext.

Our findings are summarized in Table 3. ScreenPass found that four apps sent passwords to a third-party server controlled by the app developer, four apps sent

passwords over the network in plaintext, and four stored passwords in the local file system in plaintext.

Sent to third-party servers: Four of the 28 applications that sent passwords over the network sent them to a server controlled by the app's developer. We have already discussed the fake NetFlix app above. The remaining three non-malicious apps aggregate credentials for the user's convenience: two are multiprotocol IM clients, while the other aggregates a user's financial information from her credit and investment accounts. To the credit of the financial app, its user agreement clearly states that it stores passwords on its servers and that they are used solely to provide functionality. Unfortunately, the IM clients do not discuss how users' passwords are handled in their user agreements.

Sent in plaintext over the network: We found that four of the 28 applications that sent passwords through the network did so in plaintext. ScreenPass tracked both SSL and non-SSL connections. We examined the contents of flagged non-SSL connections and found plaintext passwords in three of them. One of these apps was the fake NetFlix app, two are first-party clients for widely-used dating networks, and another was an online game and instant messaging client.

Stored in plaintext in the file system: Out of the 12 applications that stored passwords in the file system, four saved them in plaintext in the phone's file system. All

kept passwords inside a preferences xml file. We manually inspected all flagged accesses to the file system, and only reported those in which the file persisted after the app was closed.

3.6 Summary

In this chapter we have presented ScreenPass, which allows users to securely tag their passwords before handing them to third-party mobile apps. ScreenPass provides users with a trusted password-entry UI and prevents spoofing of the trusted UI through OCR.

Our evaluation of a ScreenPass prototype demonstrates that ScreenPass is robust to both static and dynamic attacks, and our energy and performance results show that running ScreenPass imposes modest overhead in the common case. Our user studies show that users are willing to tag their passwords when prompted, and that integrating a password manager into ScreenPass gives users a strong incentive to tag their passwords when they are not prompted.

4. VeriUI: Enforce Trusted UI via ARM TrustZone

In this chapter we present VeriUI [33] – a system that enhances the security of sensitive operations on touchscreen mobile devices through ARM TrustZone technique.

Mobile apps increasingly require users to login to remote services such as Facebook and Twitter. Unfortunately, today's mobile platforms provide no protection for login credentials such as passwords. To address this problem, we introduce the idea of an attested login and an embodiment of this idea called VeriUI. Attested login augments user credentials with a certificate describing the software and hardware that handled the credentials. Experiments with a VeriUI prototype found that it avoids the sluggish responsiveness of a thin-client approach, while a small app study indicates that VeriUI would require minor changes to existing apps.

4.1 Introduction

Mobile apps are immensely popular nowadays and people are increasingly access all kinds of cloud services through these mobile apps on their smartphones or tablets. Although these mobile apps can access the cloud service, they may not be developed by the cloud service provider. We call such apps third-party apps. Correspondingly, we call the cloud service accessed by third-party apps first-party cloud service. For example, an individual developer can develop a third-party app to retrieve public articles from first-party services, such as Washington Post, New York

Times, etc., and feed users via mobile devices. There are many reasons why people choose to use third-party apps. Sometimes there are no first-party apps existing in the market, and people have no choice but to use third-party apps. Even if first-party apps are available, third-party apps may still be preferred because they embed augmented functionality and integrated service, and thus provide better user experience.

As third-party apps become ubiquitous, it is almost unavoidable for users to perform sensitive operations on cloud services through these apps. Depending on the information users need to provide, these sensitive operations can be categorized as: 1) Login process which requires users' passwords. For example, before the third-party app can access the private data in the online social networks, it must ask user to first input password to login to the online social networks from the apps. 2) Online payments which require users' financial information. For example, a VoIP app will ask the user to login to the online payment site in order to purchase credits for her VoIP account.

There are several different ways to implement the above sensitive operations in third-party apps. A straightforward method is via the direct query to user by third-party apps. However, users generally do not prefer this kind of queries and the popularity of this method has declined. A more complex, but widely used method is WebUIs. There are two types of web UIs: calling default browser through IPC or embedding WebView inside the app. The IPC method involves opening another browser and loads the

required information. In the previous VoIP example, the app can redirect users to the default browser to complete credit purchasing transactions from online payment website. Differently, WebView is a UI component serving as a mini browser to load any URL. It can be easily embedded in the app and manipulated through its APIs. In the previous online social networks example, the third-party app can embed WebView to load login UI from the online social networks and ask the user to complete inputting password. WebView provides a variety of functions, enabling developers to conveniently render web pages and display contents from cloud services in their mobile apps.

However, a common weakness exists in all these methods, that is, there is no mechanism to ensure the security of sensitive information, such as password or financial account. Ideally, such user information should be only accessible by first-party cloud services, because it is secret between users and these services. However, through the analysis of current implementation methods, we find that there is a high risk that sensitive information can be revealed to third-party apps, which potentially leads to users' loss. We will illustrate the reasons as following.

The most important reason is there lacks an enforcement mechanism to regulate third-party apps so that they securely handle sensitive user interaction. To facilitate the development of third-party apps, the first-party cloud services usually provide a set of

cloud APIs for RPC calls. Some of them even provide client SDKs that third-party apps could conveniently embed. However, the first-party cloud service can neither require the third-party app to use its SDK, nor check the integrity of the sensitive code in the SDK. Put another way, the first-party cloud service has no control over what the third-party apps display to users and how they handle sensitive input from the user.

A number of third party apps choose not to use the SDK provided by the first-party service. For example, the third-party app may ask for the sensitive information directly from the user, and complete the sensitive transactions through RPC calls to cloud APIs by itself, totally ignoring the client SDK. In this way, users' sensitive information, which is supposed to be only accessible by first-party cloud services, is actually revealed to third-party apps. Involving third-party apps in the handling of sensitive information is improper due to security reasons and it is highly desirable to completely isolate the apps from this process.

Even if third-party apps include SDK to handle sensitive operations with cloud services, they can still easily steal the sensitive user input through modifying the original SDK, since currently the first-cloud service has no way to check the integrity of its client SDK used in the third-party app. For example, the third-party app can replace the WebView UI component with a hacked one which will leak the user input to the third-party app. Besides the direct replacement, third-party can also manipulate the

WebView UI component to steal sensitive information. [36, 37] discussed the security vulnerabilities of the WebView component in Andoid system and showed that it is easy to be manipulated by its host mobile app. Therefore, although WebView brings convenience to app developers and equips mobile apps with customizable mini browser, it also raises new security issues to the mobile system.

As seen from above, current mechanisms have no way to guarantee the client side provides a secure environment. As long as third-party apps participate the processing of sensitive information, users are in the danger of having their information stolen by those apps.

Aside from third-party apps, there are still other contributors in damaging the security of users' sensitive information. Phishing attacks and system spywares are two well-known threats.

Phishing attacks, which have been studied for a long time under desktop browser environment, become even worse on mobile devices due to the small screen size. In mobile browsers, URL address lines are usually hidden to make maximum utilization of the screen display, and users have to pull down the screen to see the URL. What are even worse, most embedded VebView components do not have the address lines and users have no way to check what sites they are accessing to. As a result, when the third-party app open the web UI in WebView or mobile bowser to ask users to take

sensitive operations, it is very hard for users to clearly understand the relations between apps and the web contents displayed in the WebView. Most users are even not aware of the context switch between the two different entities. Furthermore, the small size screen displays less web content with the fewer clues provided, which make it much harder for users to differentiate phishing and genuine web UIs before input sensitive information. Under this situation, users are placed in higher risk of phishing attacks.

The other well-known threat is system spywares. Even if the mobile apps are trusted and handle user interactions properly, there are still many other ways that users' sensitive information can be leaked to system spywares. For example, if the input method is malicious or compromised, sensitive input from the users will be stolen. Normal mobile systems do not provide highly secured environment for users to input private information or take sensitive operations.

To solve all the above problems, we propose a solution called VeriUI. VeriUI provides trusted UI for third-party apps to handle sensitive user interactions and guarantee strong isolation by leveraging the TrustZone technique enabled on ARM chips. TrustZone supports running two systems concurrently: a normal mobile system such as Android runs in the normal world, and a trusted light-weighted system runs in the secure world. A secured webkit runs on top of secured OS to provide generalized trusted UI to handle user interactions in sensitive operations. Third-party apps running

in the normal world can call the trusted UI through the TrustZone driver, but they are totally isolated from the trusted UI. To combat social engineering attacks, we capture user intent actively and check it before the webkit load the trusted UI. Secured webkit disables advanced web features to enforce a secure environment on the client side.

However, the trusted UI alone cannot guarantee the security of sensitive operations on mobile devices, so VeriUI also needs mechanism to enforce the usage of the trusted UI by third-party apps. To require third-party apps to call secure service to handle sensitive interactions with the user, the cloud services ask the client to provide attestation of the software stack together with the sensitive transaction request. The attestation proves to the cloud server that it is trusted UI handling interactions with the user. The enforce mechanism prevent the third-party apps from circumventing the secure service in secure world to cheat the user.

In short, we combine the hardware security provided by ARM processors and the attestation checking from trusted first-party cloud services to regulate the behaviors of third-party apps. To combat spyware in Android system, we leverage TrustZone to provide an isolated secure environment for users to input sensitive information. As for phishing attacks, we will describe our anti-phishing design in detail in Section 4.3.3.

VeriUI makes the following contributions:

- Strong isolation: We leverage the security support from TrustZone technique to provide strong isolation between third-party apps and trusted UI service. We do not need to trust rich OS in the normal world. Even if the rich OS is compromised, users' sensitive information is still safe.
- Minimize user burden: We combine the efforts from first-party cloud services and TrustZone-enabled devices to combat malicious or improper third-party apps, in order to minimize the users' attention to attacks. Users' sensitive operations are protected under TrustZone and third-party apps are required to prove to the first-party services that they provide trusted UI to users. As far as we know, this is the first work to protect integrity of sensitive code in client SDKs from first-party cloud services.
- Generalized framework: VeriUI is a generalized framework for both third-party apps and first-party cloud services. In VerUI's design, WebKit in the secure world can load HTML UI from any cloud services, and any third-party apps in the normal world can call the trusted UI service.
- Minimize change to third-party apps: Our solution requires minimum changes on current third-party app ecosystem, especially for third-party

developers. It will befit both users and cloud service providers. For users, the framework protects their passwords and cloud data. For login services, it guarantees the sensitive code integrity on client. For third-party developers, it is easy to migrate to new framework because most changes are made within the client SDK provided by cloud services.

In VeriUI, we focus on third-party apps and concentrate our discussion on how to protect sensitive operations to first-party cloud services against these third-party apps. However, our generalized solution can also be applied to first-party apps which need high security guarantees.

The rest of this chapter is organized as follows: Section 4.2 describes the trust and threat model of VeriUI, Section 4.3 describes VeriUI's design in detail, Section 4.4 presents our prototype implementation of VeriUI, Section 4.5 presents the evaluation results to VeriUI's prototype, and Section 4.6 gives our conclusions.

4.2 Trust and Threat Model

VeriUI's trust model is rooted in the hardware isolation provided by the ARM TrustZone. We do not trust the normalword operating system (e.g., Android), because a mobile user can install malware on their mobile device that compromises the Android system. For the same reason, we cannot trust any apps running in the normal world.

VeriUI assumes that all code in the secure world is trustworthy. Thus, our trusted computing base (TCB) consists of the TrustZone monitor, secure kernel, and all software that runs on top of it. VeriUI provides no guarantees if the TCB becomes compromised. As a result, even if the Android system running in the normal world is compromised, it cannot access sensitive information residing in the secure world.

VeriUI cannot prevent passwords input outside of the secure world from leaking. For example, an app could spoof the UI of the secure world to convince a user that they are interacting with the secure world even if they are not [34]. Login attempts from outside of the secure world will not be properly attested and can be rejected by a remote service. However, if an attacker can convince a user to input her password in the normal world, it could later manually input the stolen password after switching to a secure world (presumably on a device physically controlled by the attacker).

One way to prevent such an attack is to make the secure world more difficult to spoof. For example, Cloud Terminal [38] can be configured to display a user-chosen background image known only to the user and secure operating system. Under this approach, users must be trained to look for the visual secret and avoid inputting their password if it is absent.

Another way to prevent leaked passwords from causing unauthorized logins is to embed additional contextual information in an attested-login certificate. For example,

the secure world operating system could attest to login-time GPS coordinates, WiFi-scan results, or imagery from the front-facing camera. A service could then verify one or more of these data items before authorizing a login request. Our current prototype does not support this use of "trusted sensing", but attested login is general enough to incorporate this additional information. Furthermore, prior work has demonstrated how trusted sensing can be implemented with a small TCB [35].

4.3 VeriUI Design

In this section we describe the design of VeriUI in detail.

4.3.1 System Overview

As we have discussed in Section 4.1, either the WebView UI component or the default browser is vulnerable to attacks and cannot provide secure environment for users to take sensitive operations. In VeriUI, we provided a separate service called SecureWebKit to replace WebView or browser to handle sensitive user operations. SecureWebKit works as a light-weighted webkit engine to parse simple webpages and complete web requests, and it can provides general service to third-party apps. On the other hand, SecureWebKit is completely isolated from the third-party apps which call it, so it can provide trusted UI to handle use interactions securely, such as displaying online payment interfaces or typing in passwords.

Figure 12 shows system architecture of VeriUI. Two operating systems are running simultaneously based on TrustZone support. The rich OS is running in the normal world and the secure OS is running in the secure world.

For untrusted Android system, we expose a TEE API to third-party apps. These apps can invoke the SecureWebKit service running in the TrustZone through secure procedure calls. Once the app requests the SecureWebKit, the TrustZone driver will be called and the system will switch to the secure world. Once in the secure world, the SecureWebKit service will start.

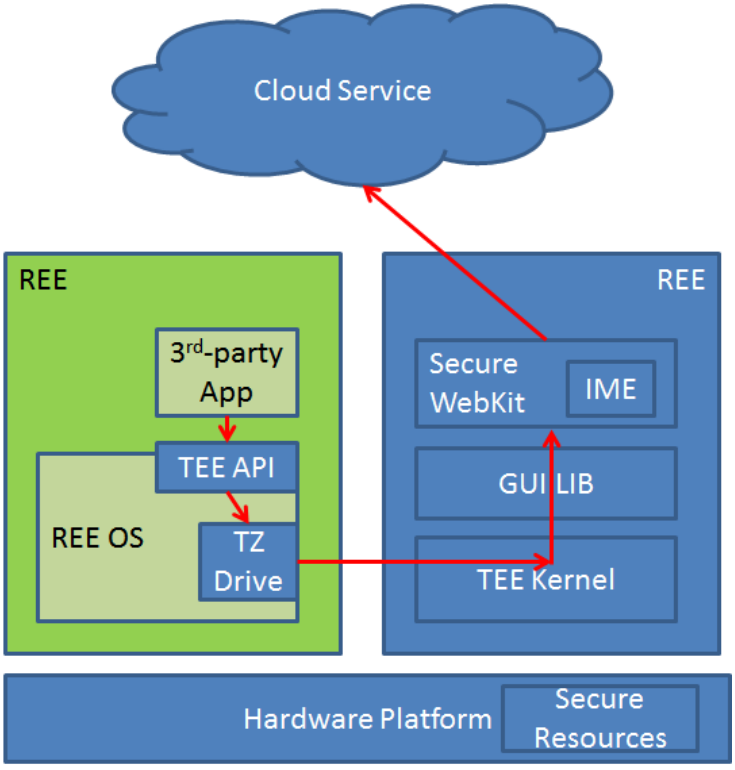

Figure 12: Overview of the VeriUI architecture.

The secure OS is a light-weight kernel running with minimum driver support, such as network, display, and touchscreen drivers. The secure OS includes GUI lib support so that we can develop secure services and apps with a trusted UI for the secure world. SecureWebKit and touchscreen input methods run on top of the software stack and users can use them to securely connect to first-party services.

To isolate SecureWebKit from the third-party apps for which it provides service, we leverage the TrustZone technique to run SecureWebKit in the secure world. Once the SecureWebKit has started, it will be running in isolation and disallowing any manipulation from the third-party app completely. And the third-party app knows nothing about the user input and operations in the whole login process. Third-party apps in the area will be limited and threats will be prevented. The isolation provided by TrustZone guarantees the security of users' sensitive operations.

We would like to provide a secure environment for users to process secure operations. The environment includes not only the SecureWebKit, but all the security facilities in the secure world. For example, in addition to the SecureWebKit, VeriUI also provides an input method engine for text entry. This input method is trusted and secure because it is isolated in the secure world. In the secure environment, we guarantee the UI displayed to handle sensitive user interactions is secure and genuine.

Memory is isolated. Normal system is restricted in the normal memory. But shared memory is allocated to help to pass parameters between normal world and secure world. When the third-party apps call SecureWebKit, they need to pass the URL as the parameter to the secure world. When SecureWebKit finishes the request, it may return data such as OAuth token back to the third-party apps.

It is very important that the trusted UI and secure environment in charge of sensitive user interactions can be remotely verified by the first-party cloud services. To provide such capability, VeriUI loads a pair of keys into the secure world memory from the hardware ROM during the secure boot process. VeriUI uses the AIK to sign secure messages and provide remote attestations. The secure kernel provides a set of APIs for the SecureWebKit to call to generate attestations. We assume a ROM that is pre-installed with a key pair by the manufacturer is accessible from within the secure world during secure boot.

4.3.2 SecureWebKit

SecureWebKit is a WebKit browser that provides secure and basic browser functionality to third-party apps in the normal world. SecureWebKit is designed to provide general service to handle sensitive operations such as authentications and payments. SecureWebKit and third-party apps are completely isolated, but they are bridged by the TrustZone monitor and secure kernel. SecureWebKit can be started by a

third-party app by passing the targeting URL as a parameter through the secure kernel. After the TrustZone monitor completes a system switch, the secure kernel starts SecureWebKit to open the URL and handle sensitive user interactions. After the user finishes interacting with the first-party service, SecureWebKit will exit and return any received data (e.g., an OAuth token) to the third-party app.

SecureWebKit disables advanced web features such as JavaScript and CSS to enhance the security of the browsing environment. SecureWebKit only supports basic HTML parsing and super-link navigation, and only communicates over HTTPS. Most sensitive operations, such as authentication and payment, only collect sensitive information such as passwords or account numbers from users. They are neither interaction-intensive (i.e. game) nor content-intensive (i.e. multimedia), so they do not need complex browser functionalities. This is even true for mobile devices with small-size screens, which cannot display too much web contents. Basic and simple web UI can satisfy the requirements of such sensitive operations.

To prevent phishing attacks, VeriUI provides a domain-selection UI for explicitly capturing user intent. Before SecureWebKit opens a target URL, it first displays a list of web domains and asks the user which domain she would like to access. Figure 13 shows the domain selection UI. After the user chooses the intended domain, SecureWebKit will check whether the SSL certificate of the target URL matches the selected domain. If a

mismatch is detected (indicating a potential phishing attack), SecureWebKit ignores the target URL and alerts the user.

After capturing the user's intent, SecureWebKit loads the target URL and restricts all future communication to the selected domain. SecureWebKit monitors web navigation and URL loading to enforce the secure domain policy until it exits and returns. Because most sensitive operations only involve a single first-party cloud service and usually can be completed in several steps, the secure domain policy is reasonable and practical. This ensures that sensitive data can only be sent to the selected domain.

Figure 13: Domain selection UI of SecureWebKit.

All web requests generated by the SecureWebKit contain a certificate signed by the secure kernel covering any sensitive user input and the software configuration of the secure world. Taking user authentication as an example, after the SecureWebKit loads

the web UI from a first-party cloud service, users type in their password and submit their request to the server. Before sending these requests, SecureWebKit invokes the secure kernel to generate an attestation. The attestation together with the request can prove to the first-party cloud service that user input was handled in a secure environment. First-party cloud service verify the attestation and can decide whether to authorize the user or not.

```
1.  <cert>
2.    <webkit>
3.      <load_url>(URL requested by third-party app)</load_url>
4.      <domain>(domain selected by user )</domain>
5.      <post_url>(URL to send user input)</post_url>
6.      <post_data>(data to be sent to post_url)</post_data>
7.      <timestamp>(timestamp)</timestamp>
8.    </webkit>
9.    <webkit_digest>SHA1(webkit)</webkit_digest>
10.   <platform>
11.     <boot>SHA1(boot partition)</boot>
12.     <system>SHA1(system partition)</system>
13.     <aik_pub>AIK_pub</aik_pub>
14.   </platform>
15.   <platform_digest>SHA1(platform)</platform_digest>
16.   <signature>
17.     <sign>sig{webkit_digest, platform_digest}AIK_priv</sign>
18.   </signature>
19. </cert>
```

Figure 14: Format of a remote attestation certificate.

Figure 3 shows the format of an attestation certificate. It contains sensitive data received by the SecureWebKit, and the configurations of the software stack in the secure world. The attestation is signed by the secure kernel using an AIK. In the WebKit part, the attestation states the initial targeting URL and the secure domain selected by the

user. The entries of post URL and post data should match the request to the server. It also includes request timestamp to prevent replay attacks. In the system part, the attention contains hashes of the software configuration (i.e., measurements) which can help first-party could service to verify the integrity of the client.

4.3.3 Prevent Phishing Attacks

Phishing attacks are very hard to prevent. Malicious third-party apps may leverage human faults or misunderstanding to cheat users to input account information or complete online payments. We consider the following two types of phishing attack scenarios:

- Malicious apps send request to SecureWebKit to open a phishing URL in the web UI.
- Malicious apps open a phishing URL in the web UI in the normal world.

For the first type of phishing attacks, we have described several methods to prevent them in the SecureWebKit design in Section 4.3.2. The most important method is to introduce the new domain selection UI to capture user intent before the SecureWebKit starts. The SecureWebKit will open the URL only when the targeting URL passed from the third-party app matches the selected domain, which means the user is not fooled by the app and really aiming to that website.

After SecureWebKit start, it restricts all user actions within the intended domain by checking the URLs in super link navigations. Since SecureWebKit has disabled all advanced web features such as JavaScript and CSS, and only supports basic HTML webpages, it prevents the web application from communicating other servers outside the secure domain or manipulating the web content and layouts dynamically to cheat users. User operations will be restricted within the same secure domain until they exit SecureWebKit and return to the app. We believe most sensitive operations can be completed under simple HTML UI in the single domain within several steps.

Finally, like many existing anti-phishing systems had already adopted, SecureWebKit could install white lists or black lists to accept access from only a limit number of trusted websites. The black lists can help filter out suspicious phishing sites and the white lists can enumerate all the popular and trust websites which provide services to third-party apps and support VeriUI verification framework. If SecureWebKit restricts access only to the servers on white list, it will jeopardize the generality of SecureWebKit, but enhance the system security greatly against phishing attacks. A trusted directory service could be introduced to update the white list dynamically and alleviate the problem of generality.

For the second type of phishing attacks, we use the following methods to prevent them:

Firstly, TrustZone LED will be turned on when the device is running under secure OS, in order to notify users about the system switch. This is an explicit sign for users to check the security environment when they take sensitive operations. This can also be treated as reverse key [15] of the SecureWebKit. We have to educate people about this and they have to pay extra attentions on it. However, since we cannot fully rely on user efforts against phishing attacks and not all mobile devices support hardware signs like this, we have other mechanisms to prevent this type of phishing attacks, or reduce the losses to the minimum.

Secondly, the first-party cloud service requires and verifies the attestation together with sensitive requests from the third-party apps. This mechanism requires the third party app to use VeriUI to interact with users for sensitive operations. Any sensitive operations such as OAuth authorizations and payment transactions will be failed without verified attestations. As a result, the third-party app will not receive the OAuth token to access the user's data in the cloud, or it will not get paid or money transferred from the online bank. And the user will be notified by the first-party cloud services regarding the abnormal failure through email or other methods. Even the first phishing attacks succeed in cheating the users, their private data in the cloud and the financial properties are still under protected.

Finally, attestation key will be used as the second authentication factor to detect login or payment operations from new devices. First-party could services will send notification to the user through email once they find new devices. Even if the attacker gets the user's sensitive information anyway, the attacker has to complete sensitive transactions to the first-party service from another VeriUI-enabled device. Then the user will be notified and alerted. The attestation key becomes the last defense firewall to reduce the user's loss.

In general, phishing attack is a hard problem and very difficult to prevent completely, because the social-engineering attacks work on human's faults. For example, a malicious app could even ask the user to input account information directly, and the user may really give her password to the app. However, VeriUI guarantees that no sensitive operation will take effect if they are sent from an untrusted environment. It reduced the users' losses to the minimum because the malicious app can do nothing to users' cloud data or financial properties without verified attestations.

4.4 Implementation

We built a VeriUI prototype based on the design described in Section 4.3. We implemented the prototype on an ARM SOC equipped with ARM Cortex A8 cores. The following subsections describe how we developed the prototype in detail.

4.4.1 Duo System Boot

We cut a stock Linux Kernel to reduce the TCB as much as possible. We only left the kernel and minimum drivers in the secure system, including the display driver, touchscreen driver, and network driver. Due to the limitation of the development board, we can only use a wired Ethernet network to connect to the servers.

We use U-Boot [21] to start the two systems on the ARM SOC, and developed a monitor to control the world switches. During system start, we separate the memory for secure world and normal world. We reserve part of memory as shared memory to bridge the data communication between the two worlds.

We modify the Linux kernel under Android to add the TrustZone driver into it. It exposes an API to Android apps to start a secure procedure call into TEE.

4.4.2 Port GUI in TEE

We use Qt [13] as our GUI lib in the secure world, since Qt provides better support for WebKit than other simple GUI libs in Linux.

We implemented our SecureWebKit based on QtWeb [14]. We ported QtWeb onto the secure OS and implemented all the security and anti-phishing features described in Section 4.3.2. We removed all advanced features in QtWeb, and added the user-intent selection interface when SecureWebKit starts. We add attestations when

sending request to the servers. We also developed a touchscreen input method for the secure world to help user input data.

4.4.3 Attestation

We used OpenSSL [10] for generating attestations. The lib can provide attestations about the image of the secure OS. It provides an API for apps in the secure word to call. SecureWebKit will call the lib to generate attestations when it send request to the servers.

4.4.4 Demo App using VeriUI

To verify the usage of VeriUI system and evaluate the performance of it, we developed a simple Android app that supports OAuth login to Twitter. The app is built on top of standard SDKs provided by Twitter, and we modified these libraries to make a secure procedure calls to the SecureWebKit in TEE through TrustZone driver in REE.

4.5 Evaluation

VeriUI should support legitimate apps and require little or no modifications to existing apps. To better understand the feasibility of this goal, we performed a small survey of third-party Android apps and SDKs provided by first-party services. We characterize how current third-party apps handle sensitive operations to first-party could services, and how easily for app developers to switch to VeriUI framework.

For VeriUI to be practical, it must have acceptable performance. To understand its performance we compared its UI responsiveness to the thin-client approach embodied by Cloud Terminal [38].

To understand the overhead to complete sensitive operations through VeriUI, we measure the time for a third-party app to call VeriUI and compare it with the time to call VebView or mobile browser. We also measure the overhead to generate and verify an attestation certificate.

4.5.1 App and SDK Study

Table 4: Popular third-party apps for Twitter and Facebook in the study.

Cloud Services	Third-party Apps
Twitter	TweetCaster, UberSocial, TweetDeck, Plume, Seesmic, HootSuite, Slices, Janetter, Scope, Echofon, TwitPal, Tuippuru, Twidere, Feel on!, and Tweedle
Facebook	Go!Chat, TweetDeck, Video to Facebook, Seesmic, Friendcaster, HootSuite, BeejiveIM, Facebook Pages Manager, Sync.ME, and Contact Sync

To better understand how current mobile apps request OAuth tokens, we performed a survey of existing Android apps from Google Play for most popular cloud

services. We examined the 15 popular third-party Twitter apps and 10 popular third-party Facebook apps, which are listed in Table 4. We did not consider the first-party apps developed by either Facebook or Twitter since they remain under the services control. Although our study was limited to Facebook and Twitter apps, we believe that these are representative of third-party apps written for other cloud services. The Facebook and Twitter Platforms are the most popular platform for third-party app developers.

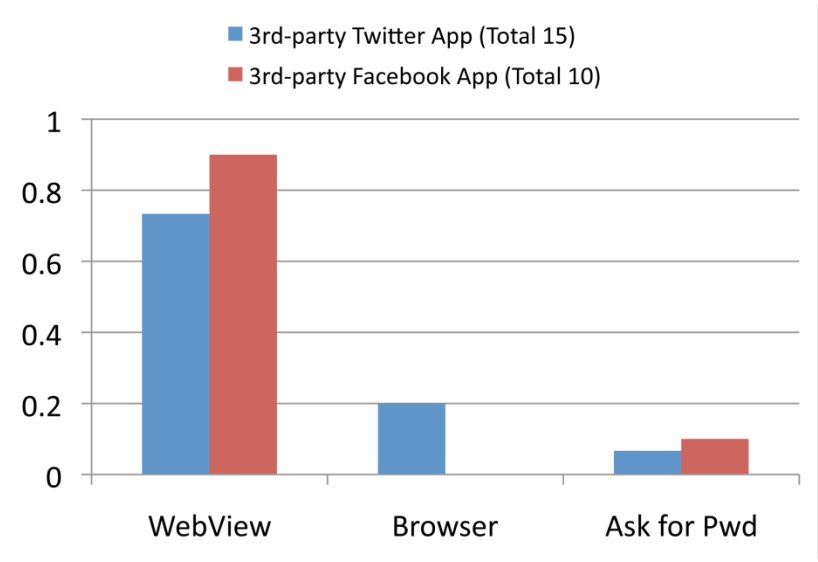

Figure 15: Third-party app study for Facebook and Twitter - they use one of the three methods to handle user authentications: embedded WebView, mobile browser, or ask for user accounts directly.

We were most interested in whether apps used Android WebView widgets, the default browser, or handled OAuth internally. Our results are in Figure 15. Among the 25 apps in our study, we found that can see that most apps use WebView for OAuth: 11

out of 15 Twitter apps and nine out of 10 Facebook apps. No Facebook apps in our survey invoked the web browser, although three Twitter apps did. And one app from each category handled OAuth themselves. This indicates that for most apps, password requests could be redirected to VeriUI without many changes.

We also study the SDKs provided by the two most popular cloud services. For Twitter, Twitter4j [20] is the recommended SDK for Android platform. It does not contain the login UI part because it is developed for Java in general, not for Android specifically. Third-party apps usually adopts WebView or default browser to complete the login process, and use IPC call to redirect and pass authorization code. For Facebook, Official SDK [1] is well encapsulated. The WebView-based login UI is packed into a prompt dialog.

From third-party app and SDK study, we can see for most apps, the login UI can be easily split from the code. Modifications of SDKs can be done by the first-party cloud service providers so the effects on third-party app developers will be minimized. Our implementation to modify the Twitter SDK and app also indicates that the efforts to switch from WebView or browser to VeriUI are very low.

4.5.2 UI responsiveness

As mentioned previously, Cloud Terminal [38] addresses similar problems as VeriUI, but adopts a thin-client approach. This results in a smaller TCB, but leaves it

vulnerable to the high network latencies that are common for cellular data networks. To test our hypothesis that VeriUI would provide a better user experience than a thin client, we measured the time for VeriUI's SecureWebKit and a web browser accessed via VNC client to complete keystrokes and window scrolls on under various network latencies. Both clients ran in the secure world on our prototype board.

We setup the experimental environment using the development board and two desktop servers: the VNC server and the proxy server. We put them in the same network and direct all the traffic from the development board through the proxy server, so that we could inject latency at the proxy server to explore different latencies between the client and the VNC server. The average RTT between the client and the VNC server without any injected latency is 29ms.

Under these network configurations, we used VNC client and SecureWebKit to open the same login webpage. When we use VNC client to open the webpage, it was actually opened from the browser in VNC server but viewed and manipulated on the client. After the login page loaded, we pressed keys to see how much time the character took to be displayed on the client. For SecureWebKit, we instrumented the code to capture these times; for VNC client, we modified its input method to capture the starting time for pressing a key and the finishing time was when the client reads the update from

the server and displays it. We injected different latency from 0ms to 250ms for all the traffic in the proxy server to test the UI responsiveness.

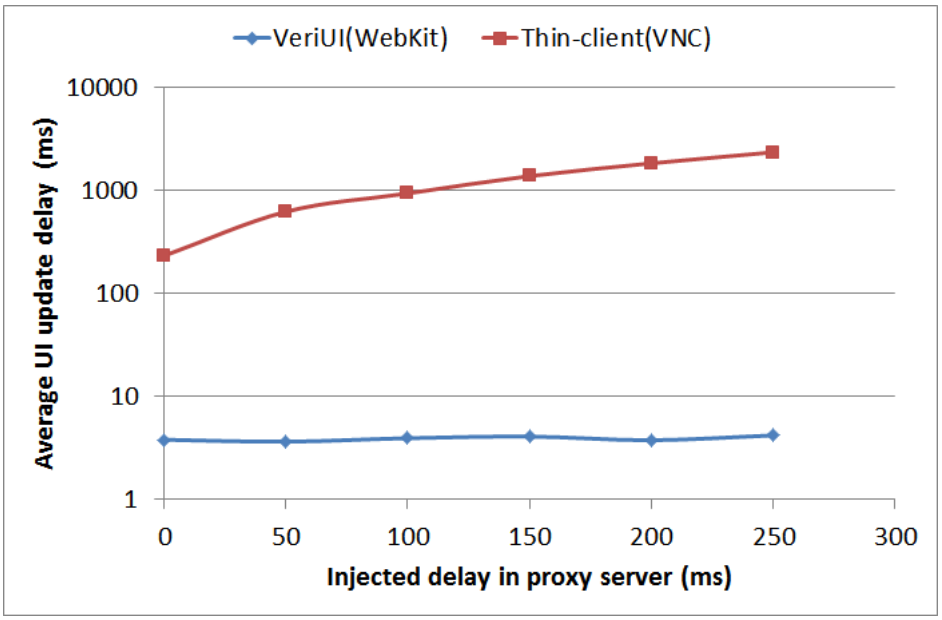

Figure 16: Comparison of UI responsiveness for VeriUI and thin-client.

Our results are in Figure 16. Unsurprisingly VeriUI's approach provides a much more responsive UI than the thin client approach. Even for an injected delay of 250ms in the proxy server, the VNC client took over 2 seconds to respond to a keystroke, whereas VeriUI's responsiveness was independent of network latency.

4.5.3 Performance overhead

In this section we evaluation the performance overhead of running VeriUI.

We first want to know the performance of transitioning from third-party apps in the normal world to SecureWebKit in the secure world, and load a URL. To understand

the cost of system-switching and starting SecureWebKit, we measures the time from the beginning of SMC call to the end of URL loading in SecureWebKit. We also would like to compare the run time performance of VeriUI to the other two methods: embedded WebKit and mobile browser. So we developed two Android apps: the first app embedding a WebView widget and the second app to call the first app. We can measure the run time to load URL in an embedded WebKit through the first app. And we use the second app to send intent to the first app which serves as a mini mobile browser to open the URL from the received intent. Then we can measure the run time of starting a mobile browser to load URL. We use the three methods to load the Twitter login webpage under the same network condition.

Figure 17 shows the run time performance results. VeriUI, with an average delay of 3305 milliseconds, has the best performance among the three methods. Because delay for system switch and starting SecureWebKit is shorter than starting an embedded WebView widget. Mobile browser is the slowest because it has to complete inter-app context switch before it start to load the URL.

We also interested in the overhead of generating and verifying attestation. According to our measurements, the average cost to generate a remote attestation certificate from the secure kernel on the development board is 216 milliseconds. And the average cost to verify an attestation certificate from a commodity server (Intel i5 2.8GHz

CPU, 2G memory) is 15 milliseconds. Our results show that the cost of remote attestation is very low.

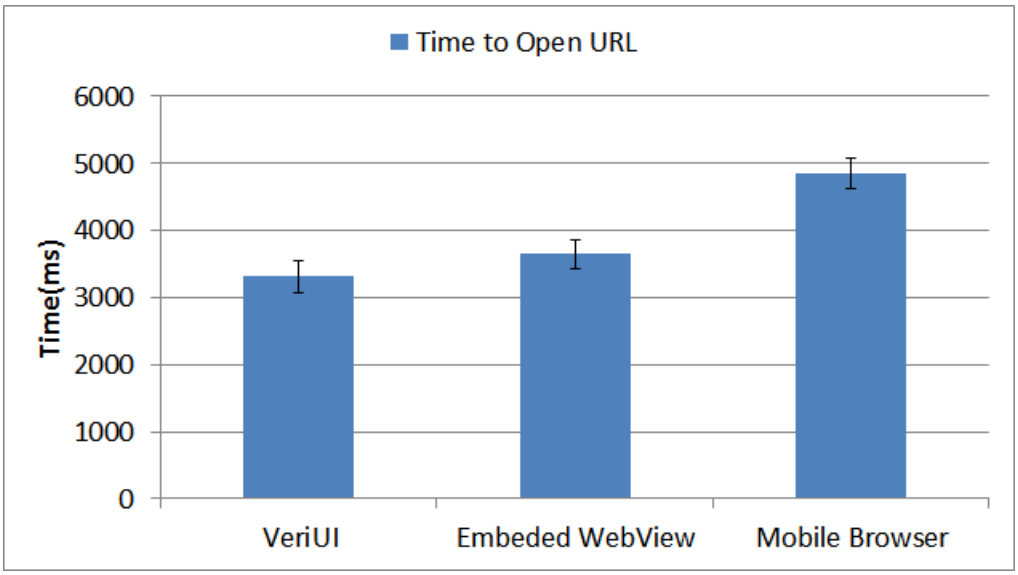

Figure 17: Comparison of run time performance for VeriUI, embedded WebView, and mobile browser to start and load URL.

We also measured the time to rebooting the secure world which is 7232 milliseconds in average. Plus the time to switch system and start SecureWebKit, the total delay should be 10.5 seconds in average. For highly sensitive operations, we can also consider not only restarting SecureWebKit, but also rebooting the whole secure world to give users the highest security guarantee. Our evaluation shows the delay is still acceptable in this case.

4.6 Summary

This paper has presented VeriUI, which helps thwart phishing attacks by mobile apps through attested login. Attested login augments a user's credentials with information about the hardware and software that handled those credentials. By separating credential handling from the rest of an app and executing this code in a secure environment, users and services can be given greater assurance that passwords and other sensitive data has been handled properly. A small app study indicates that our architecture would require modest modifications to third-party apps, and experiments with a VeriUI prototype demonstrate that it provides better UI responsiveness than a thin client approach.

5. Related Work

Due to the importance of client security on user authentication and online payment, it has attracted significant attention from the research community in general. In this chapter we discuss prior work that is related to the systems proposed in this dissertation. Section 5.1 compares our work to the most relevant systems on password security. In Section 5.2, we discuss prior work that involves trust computing techniques.

5.1 Password Security and Anti-spoofing Techniques

The problem of securely delivering sensitive data from a user to trusted software is not new. Similar problems on PCs have been addressed through a secure attention sequences (SAS), such as "@@" or the F2 in Bumpy [40] and PwdHash [47], respectively. The wellknown ctrl-alt-delete command on Windows machines is also an SAS.

Relying on an SAS requires on a secure path from a user to the OS, usually through a trusted keyboard driver. Unfortunately, recognizing an SAS on a modern mobile platform is more challenging than on a PC, since nearly all popular devices lack a physical keyboard. Mobile operating systems can interpret taps on a touchscreen only as a set of coordinates with opaque semantics. Nontouchscreen inputs are scarce by design and typically have wellestablished functionality (e.g., the home button of the iPhone or the menu button on Android devices).

Previous attempts to create secure user interfaces on desktop machines [29, 30] have restricted where applications may write on the screen rather than what they may write. The general approach of these systems is to partition the display into regions that are writable by untrusted application and regions that are writable only by the trusted computing base. This approach is not appropriate on small-screen mobile devices, where screen real estate is at a premium. For example, mobile games would be much less usable if regions of the display were off limits.

Using visual similarity to prevent spoofing has been previously used to identify phishing attacks [22, 30]. These projects compile training databases of well-known websites and then analyze unknown email and web sites to determine how visually similar they are to entries in the training database. The primary difference between our proposal and this prior work is that OCR can detect spoofed keyboards more accurately and efficiently than general purpose computer vision can identify logos and other iconography. In addition, performing similarity analysis continuously on the display of a mobile device imposes additional performance constraints not faced by the network proxies used to detect phishing emails and web sites.

As mentioned previously, many studies have shown that security indicators are ineffective on web users [25, 26, 49, 57]. However, our user study results suggest that

integrating a password manager into ScreenPass provides a strong incentive for users to tag their passwords.

Android and iOS provide integrated account services for various services such as Google, Twitter, and Facebook. However, these account services do not prevent malicious apps from asking a user for her login credentials.

[46] proposes a new way for users to assign and monitor permissions to their apps through ACGs (access control gadgets). If a user grants an app permission to access a resource, such as a camera, the system embeds the appropriate ACG within its UI. ACGs are a useful way to make apps' permissions explicit and visible to users. Unfortunately, ACGs cannot solve the problem of spoofed keyboards on mobile devices, because mobile platforms must support full-screen mode.

Finally, LayerCake [44, 45] finds that allowing web and smartphone applications to embed user interfaces from other parties comes with security implications, both for the embedded interfaces and the host page or application. It explores the requirements for a system to support secure embedded user interfaces by systematically analyzing existing systems like browsers, smartphones, and research systems. It modifies Android to support secure interface embedding and evaluate the implementation using case studies that rely on embedded interfaces, such as advertisement libraries, Facebook social plugins (e.g., the "Like" button), and access control gadgets.

5.2 Trust Computing Related

Cloud Terminal [38] proposed a framework to provide secure client to access sensitive application on the cloud. It provides strong isolation from compromised OS, and attestable to both server and user. Cloud Terminal moves the general client – browser to the cloud and handles sensitive applications. The client side only handles user input from keyboard and mouse, and displays the remote desktop of cloud servers through VNC. The client uses mutual attestations and encryption to setup secure channel with the cloud server. Cloud Terminal relies on light-weighted hypervisor to capture all user input and isolate its display from untrusted OS. It uses secure path to bring up the Cloud Terminal.

TLR [48] provides a framework to run trusted applications on smartphones by leveraging ARM TrustZone technology and porting .NET MicroFramework to the TEE. A secure application can package the code handling sensitive data into TrustLet and run it in the TrustBox in TEE. The integrity of TrustLet can be verified by secure hash and bound with the sealed sensitive data. Communications through the boundary is via Secure Procedure Call.

[35] proposes two software abstractions to develop trusted sensor applications: sensor attestation and sensor seal. They are based on the two important primitives provided by trusted computing: software attestation and sealed storage. They

implement the two abstractions on both x86 and ARM platforms. TrustZone technology makes the implementation on ARM much easier than its counterpart on x86.

SIMlet [43] presents a new trustworthy computing abstraction for trust billing through ARM TrustZone. It allows content providers to pay for the traffic generated by mobile users visiting their websites or using their services. To implement split billing securely on a mobile platform, a SIMlet can be bound to a network socket to monitor and account all the traffic exchanged over the network socket. SIMlets provide trustworthy proofs of a device's mobile traffic, and such proofs can be redeemed at a content provider involved in split billing.

Flicker [39] is an infrastructure for executing security-sensitive code in complete isolation while trusting as few as 250 lines of additional code. Flicker can also provide meaningful, fine-grained attestation of the code executed (as well as its inputs and outputs) to a remote party. Given the correlation between code size and bugs in the code, Flicker significantly improves the security and reliability of the code it executes. Flicker guarantees these properties even if the BIOS, OS and DMA-enabled devices are all malicious. Flicker leverages trust-computing support from commodity processors.

Nexus [50-52] implements logical attestation in a new operating system which executes natively on x86 platforms equipped with secure processors. It provides a general-purpose and flexible attestation mechanism to establish statements about the

current state of a computation, and a strong, high-performance isolation mechanism to enable reasoning about future behavior based on statements about the present. When deployed on a trustworthy cloud-computing stack, logical attestation is efficient, achieves high-performance, and can run applications that provide qualitative guarantees.

6. Conclusions

Mobile devices such as smartphones and tablets are becoming one of the essential parts in people's daily life, but they also introduce new security challenges and problems. Users access all kinds of cloud services through mobile apps, but their sensitive operations with cloud servers, such as user authentication and online payment, are under the threats from malwares. This dissertation presents two systems, ScreenPass and VeriUI, which protect users' sensitive input such as credentials against buggy and malicious apps in their mobile devices. ScreenPass tracks and monitors the usage of sensitive input by mobile apps, while VeriUI completely isolate sensitive operations from untrusted code. They both use trusted UI to handle sensitive user interactions, which forms the base for the two security solutions.

This dissertation makes contributions in three major areas. The first area is conceptual – it consists of the novel ideas generated by this work. The second area is a set of artifact system that we developed to validate this dissertation. The final area of contribution is the experimental evaluation of the systems that validate the feasibility of this dissertation's statement.

6.1 Conceptual contributions

This dissertation makes the following conceptual contributions:

- I showed that it is necessary to regulate the behaviors of mobile apps when people are using them to access cloud services, because improper and malicious apps do exist. I propose two methods to protect sensitive input: tagging and monitoring the usage of it, or isolating it from untrusted apps.
- I proposed to use trusted UI to handle sensitive user interactions in mobile apps. I proposed to use two methods to enforce the usage of trusted UI: computer vision analysis and verifiable remote attestation.
- I proposed a new UI to capture user intent. I showed it's usability in preventing social-engineering attacks.
- I showed that it is possible to enhance the security of software keyboards by OCR analysis in touchscreen devices.
- I showed that by leveraging ARM TrustZone technique, it is possible to isolate mobile apps from sensitive information while keep the original functionality with a few minor changes in code.

6.2 Artifacts

In the course of this dissertation, I have developed two major artifacts to validate the thesis: ScreenPass, and VeriUI. I built prototypes of these systems to confirm the feasibility of systems' implementations and their usability.

- I built ScreenPass, an extension to Android system enhances password security on touchscreen devices through OCR and taint-tracking techniques. ScreenPass is composed of several related components: a special input method with domain-selection UI, a framebuffer checker with ORC analysis, a taint-tracking extension with warning facility.
- I implemented an input method app and a data-collection server for the usability study of ScreenPass. The input method has the identical UI with that in ScreenPass, and report usage data regarding the domain-selection UI to the data-collection server.
- I built VriUI, a system leverages ARM TrustZone to enhance security of sensitive operation on mobile devices. VeriUI supports to run two systems currently, an Android in the normal world and a minimized Linux kernel in the secure world. The secure world includes a secure web client to provide generalized web service, and the normal world includes a TrustZone driver to provide API for mobile apps.
- I implemented a demo app to verify the usability of VeriUI.

Table 4 compares of ScreenPass and VeriUI on how to practice the three principles of designing and implementing trusted UI presented in Section 1.3: secure environment, usage enforcement, and anti-spoofing campaign.

Table 5: Summary and comparison of ScreenPass and VeriUI.

Principle	ScreenPass	VeriUI
Secure environment	Tags and tracks sensitive data.	Isolates sensitive input from untrusted code.
Usage enforcement	OCR to enforce the usage of secure software keyboard.	Remote attestation to enforce the usage of VeriUI.
Anti-spoofing campaign	OCR to prevent spoofed software keyboards. Capture user intent to check domain.	Capture user intent to check domain. Public key in attestation as second factor.

6.3 Evaluation results

The evaluation results in this dissertation present the most important insight: When users access cloud services from touchscreen mobile devices, it is feasible to enhance the security for sensitive operations by enforcing trusted UI in mobile apps.

- The app study on ScreenPass does find a malicious app stealing users' passwords, and a number of unsecure or problematic apps which do not handle user credentials properly. It is absolutely necessary to enhance security for sensitive operations through mobile apps.

- The evaluation of ScreenPass shows acceptable performance and energy overhead. It shows good performance against both static and dynamic spoofing-keyboard attacks.
- The evaluation of VeriUI shows very low overhead. It shows current apps can easily fit to use VeriUI framework by changing only small amount of code in SDKs.
- The usability study for the new UI to capture user intent proves its usability.

Bibliography

[1] *Android - Facebook Developers*. Available: https://developers.facebook.com/docs/android/

[2] *Apple - Press Info - Apple Updates iOS to 6.1*. Available: http://www.apple.com/pr/library/2013/01/28Apple-Updates-iOS-to-6-1.html

[3] *comScore: 4 Out Of 5 Smartphone Owners Use Device To Shop; Amazon Is The Most Popular Mobile Retailer | TechCrunch*. Available: http://techcrunch.com/2012/09/19/comscore-4-out-of-5-smartphone-owners-use-device-to-shop-amazon-most-popular-mobile-retailer/

[4] *Facebook statistics | Facebook*. Available: https://www.facebook.com/pages/Facebook-statistics/119768528069029

[5] *Facebook Statistics | Statistic Brain*. Available: http://www.statisticbrain.com/facebook-statistics/

[6] *Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time*. Available: http://www.gartner.com/newsroom/id/2573415

[7] *Google: 900 million Android activations, 48 billion app installs*. Available: http://www.androidauthority.com/google-io-android-activations-210036/

[8] *The magic moment: Smartphones now half of all U.S. mobiles | VentureBeat*. Available: http://venturebeat.com/2012/03/29/the-magic-moment-smartphones-now-half-of-all-u-s-mobiles/

[9] *OAuth 2.0 — OAuth*. Available: http://oauth.net/2/

[10] *OpenSSL: The Open Source toolkit for SSL/TLS*. Available: http://www.openssl.org/

[11] *People use smartphones nearly an hour a day, study says - CNN.com*. Available: http://www.cnn.com/2013/05/29/tech/mobile/smartphone-time-study/index.html

[12] *Press Center - Instagram*. Available: http://instagram.com/press/#

[13] *Qt Project*. Available: http://qt-project.org/

[14] *QtWeb - Portable Web Browser*. Available: http://qtweb.net/

[15] *SiteKey Security from Bank of America*. Available: https://www.bankofamerica.com/privacy/online-mobile-banking-privacy/sitekey.go

[16] *Smartphones killing point-and-shoots, now take almost 1/3 of photos — Tech News and Analysis*. Available: http://gigaom.com/2011/12/22/smartphones-killing-point-and-shoots-now-take-almost-13-of-photos/

[17] *Statistics - YouTube*. Available: http://www.youtube.com/yt/press/statistics.html

[18] *Trusted Computing Group - Home*. Available: https://www.trustedcomputinggroup.org/

[19] *TrustZone - ARM*. Available: http://www.arm.com/products/processors/technologies/trustzone.php

[20] *Twitter Libraries | Twitter Developers*. Available: https://dev.twitter.com/docs/twitter-libraries

[21] *U-Boot - the Universal Boot Loader*. Available: http://www.denx.de/wiki/U-Boot

[22] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker, "Spamscatter: characterizing internet scam hosting infrastructure," in *Proceedings of 16th USENIX Conference on Security Symposium*, Boston, MA, 2007, pp. 1-14.

[23] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, Stockholm, Sweden, 2011, pp. 47-56.

[24] L. Cai and H. Chen, "TouchLogger: inferring keystrokes on touch screen from smartphone motion," in *Proceedings of the 6th USENIX conference on Hot topics in security*, San Francisco, CA, 2011, pp. 9-9.

[25] R. Dhamija, J. D. Tygar, and M. Hearst, "Why phishing works," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Montréal, Québec, Canada, 2006, pp. 581-590.

[26] S. Egelman, L. F. Cranor, and J. Hong, "You've been warned: an empirical study of the effectiveness of web browser phishing warnings," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Florence, Italy, 2008, pp. 1065-1074.

[27] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, *et al.*, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, Vancouver, BC, Canada, 2010.

[28] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, San Francisco, California, USA, 2010, pp. 179-194.

[29] N. Feske and C. Helmuth, "A Nitpicker's guide to a minimal-complexity secure GUI," in *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005, pp. 85-94.

[30] A. Y. Fu, L. Wenyin, and X. Deng, "Detecting Phishing Web Pages with Visual Similarity Assessment Based on Earth Mover's Distance (EMD)," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, pp. 301-311, 2006.

[31] X. Jiang. *Smishing Vulnerability in Multiple Android Platforms*. Available: http://www.csc.ncsu.edu/faculty/jiang/smishing.html

[32] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation," in *Proceeding of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[33] D. Liu and L. P. Cox, "VeriUI: Enforce Trusted UI on Touchscreen Mobile Devices," in *submission*, 2013.

[34] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. P. Cox, "ScreenPass: secure password entry on touchscreen devices," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, Taipei, Taiwan, 2013, pp. 291-304.

[35] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, Low Wood Bay, Lake District, UK, 2012, pp. 365-378.

[36] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Proceedings of the 27th Annual Computer Security Applications Conference*, Orlando, Florida, 2011, pp. 343-352.

[37] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, iOS, and windows phone," in *Proceedings of the 5th international conference on Foundations and Practice of Security*, Montreal, QC, Canada, 2013, pp. 227-243.

[38] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, *et al.*, "Cloud terminal: secure access to sensitive applications from untrusted systems," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, Boston, MA, 2012, pp. 14-14.

[39] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for tcb minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Glasgow, Scotland UK, 2008, pp. 315-328.

[40] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "ACCessory: password inference using accelerometers on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, San Diego, California, 2012, pp. 1-6.

[41] J. M. M. A. Perrig and M. K. Reiter, "Safe Passage for Passwords and Other Sensitive Data," in *Proceeding of the 16th Annual Network and Distributed System Security Symposium*.

[42] A. Rahmati, C. Tossell, C. Shepard, P. Kortum, and L. Zhong, "Exploring iPhone usage: the influence of socioeconomic differences on smartphone adoption, usage and usability," in *Proceedings of the 14th international conference on Human-computer interaction with mobile devices and services*, San Francisco, California, USA, 2012, pp. 11-20.

[43] H. Raj, S. Saroiu, A. Wolman, and J. Padhye, "Splitting the bill for mobile data with SIMlets," in *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, Jekyll Island, Georgia, 2013, pp. 1-6.

[44] F. Roesner, J. Fogarty, and T. Kohno, "User interface toolkit mechanisms for securing interface elements," presented at the Proceedings of the 25th annual ACM symposium on User interface software and technology, Cambridge, Massachusetts, USA, 2012.

[45] F. Roesner and T. Kohno, "Securing Embedded User Interfaces: Android and Beyond," in *Proceedings of the 22nd USENIX conference on Security Symposium*.

[46] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 224-238.

[47] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger password authentication using browser extensions," in *Proceedings of the 14th USENIX conference on Security Symposium - Volume 14*, Baltimore, MD, 2005, pp. 2-2.

[48] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Trusted language runtime (TLR): enabling trusted applications on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, Phoenix, Arizona, 2011, pp. 21-26.

[49] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The Emperor's New Security Indicators," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007, pp. 51-65.

[50] F. B. Schneider, K. Walsh, and E. G. Sirer, "Nexus authorization logic (NAL): Design rationale and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 14, pp. 1-28, 2011.

[51] A. Shieh, D. Williams, E. G. Sirer, and F. B. Schneider, "Nexus: a new operating system for trustworthy computing," presented at the Proceedings of the twentieth ACM symposium on Operating systems principles, Brighton, United Kingdom, 2005.

[52] E. G. Sirer, W. d. Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, *et al.*, "Logical attestation: an authorization architecture for trustworthy computing," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, 2011, pp. 249-264.

[53] R. Smith, "An overview of the Tesseract OCR engine," in *Proceedings of the 9th Conference on Document Analysis and Recognition*, 2007, pp. 629-633.

[54] R. Smith, D. Antonova, and D.-S. Lee, "Adapting the Tesseract open source OCR engine for multilingual OCR," presented at the Proceedings of the International Workshop on Multilingual OCR, Barcelona, Spain, 2009.

[55] S. Tang, H. Mai, and S. T. King, "Trust and protection in the Illinois browser operating system," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, Vancouver, BC, Canada, 2010, pp. 1-8.

[56] R. Unnikrishnan and R. Smith, "Combined script and page orientation estimation using the Tesseract OCR engine," presented at the Proceedings of the International Workshop on Multilingual OCR, Barcelona, Spain, 2009.

[57] M. Wu, R. C. Miller, and S. L. Garfinkel, "Do security toolbars actually prevent phishing attacks?," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Montreal, Quebec, Canada, 2006, pp. 601-610.

[58] Z. Yajin and J. Xuxian, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 95-109.

Biography

Dongtao Liu was born on March 1st, 1983 in Shijiazhuang, Hebei Province, China. He received his B.S. in Automation and M.S. in Computer Science from Tsinghua University in Beijing, China in 2005 and 2008 respectively. From fall 2008, he joined the Department of Computer Science at Duke University as a PhD student working with Dr. Landon P. Cox. His research interests include security on mobile computing and decentralized online social networks. He has published three papers and submitted another one in these areas. He received his second M.S. in Computer Science from Duke University in 2012. He defended his PhD dissertation at Duke University in Nov. 2013.