VeriUI: Attested Login for Mobile Devices

Dongtao Liu
Duke University
dliu@cs.duke.edu

Landon P. Cox
Duke University
lpcox@cs.duke.edu

ABSTRACT

Mobile apps increasingly require users to login to remote services such as Facebook and Twitter. Unfortunately, today's mobile platforms provide weak 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.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems

General Terms

Security

Keywords

Phishing, Trusted Computing, Mobile Computing

1. INTRODUCTION

Mobile apps often augment their functionality by accessing user data maintained by services such as Dropbox, Facebook, and Twitter. Before an app can access a user's data, she must authenticate herself by providing a username and password. Sadly, today's mobile platforms provide weak password protections, leaving users vulnerable to increasingly sophisticated mobile phishing attacks [9, 15, 18].

Two-factor authentication (TFA) has been embraced by numerous services to combat phishing attacks. Under TFA a user must provide both her password (i.e., "something the user knows") and a one-time token that can only be accessed from secure hardware (i.e., "something the user possesses"). For example, after a login attempt, Google sends a token to a TFA user's mobile phone via SMS, and the user can only access her account after typing in the token. TFA prevents unauthorized logins because an attacker is highly unlikely to simultaneously learn a user's password and gain possession of her secure hardware.

The primary drawback of TFA is poor usability [8]. If a user does not possess her secure hardware, she cannot login. Worse, recovering an account after *losing* one's secure hardware is exceptionally painful. For example, Google's advice for TFA users who have lost their mobile phone includes requesting a new phone with the old number (presumably a non-Android device or an Android device setup under another account), and, failing that, initiating a multi-day account-recovery process. Due to these issues, TFA remains opt-in for all large-scale services of which we are aware, leaving the vast majority of users vulnerable to attack.

In this paper, we present a system called VeriUI that provides strong password protection for mobile devices. VeriUI is designed to prevent phishing attacks by mobile apps through a secure, hardware-isolated environment for password input and transmission. An app can invoke a web browser running in the secure environment to retrieve an OAuth token after the user successfully authenticates. Crucially, malicious code cannot access a user's password data, even if the operating system is compromised.

VeriUI embodies a login mechanism we call *attested login* that protects user-provided login credentials from untrusted code and cryptographically binds this data to a trustworthy execution environment. VeriUI's secure environment provides a trustworthy setting for handling passwords through hardware isolation, while its trusted software only reveals password data to administrative domains explicitly specified by a user. When a user logs in via a VeriUI device, VeriUI augments the user's credentials with a certificate describing the hardware and software state of her device and other contextual meta-data.

An attested login is only trustworthy if the code that handles password data has a small attack surface. Prior work on CloudTerminal [12] addressed this problem by providing a secure thin-client terminal. The thin-client terminal securely connects to a trusted server, which proxies interactions with other remote services. While CloudTerminal has a small client-side trusted computing base, a thin-client approach leads to poor UI responsiveness when network latency is high (as is commonly the case for cellular networks). Thus, instead of a thin-client, VeriUI's secure environment provides a web browser and limits its communication to servers in a user-specified domain. A secure web browser also gives existing apps a straightforward way to transition from obtaining OAuth tokens through Android's mobile web browser and WebView widgets to doing so via VeriUI.

ACM HotMobile'14, February 26–27, 2014, Santa Barbara, CA, USA.

This paper makes the following contributions:

- We introduce the notion of an attested login and describe the design and implementation of a system, VeriUI, that embodies it.
- We describe experiments with a VeriUI prototype and demonstrate that code with a relatively small attack surface can provide a responsive UI.

The rest of this paper is organized as follows: Section 2 describes background on technologies underlying VeriUI, Section 3 describes our trust and threat model, Section 4 describes VeriUI's design in detail, Section 5 describes our prototype implementation of VeriUI, Section 6 describes an evaluation of our VeriUI prototype, Section 7 discusses related work, and Section 8 gives our conclusions.

2. BACKGROUND

In this section, we provide background information on technologies underlying VeriUI's motivation, design, and implementation.

2.1 OAuth 2.0

OAuth 2.0 [1] is an authorization protocol widely used by mobile apps and web services. It allows a user to grant a third party access to resources without directly exposing long-term credentials such as a password to the third party. Instead, services issue OAuth tokens to third parties, which can use the tokens as capabilities for accessing a user's data. Each token has an expiration date and corresponds to a list of resources maintained by the service that issued it.

The typical procedure for obtaining an OAuth token is as follows: when a mobile app initially wants to access user data maintained by a service, the app redirects the user to the OAuth login web page of the service, passing a list of requested resources as parameters. The user authenticates herself through the web UI with her username and password and approves the app's resource requests. The service then generates an OAuth token and returns a client secret to the app (potentially indirectly through a server coordinating with the app). The app uses the client secret to retrieve its OAuth token from the service. Once it possesses an OAuth token, an app use the token to access user data via a service's API. The mobile app usually stores the OAuth token on the client for future use, and it may also send the token to the app's own server.

2.2 Web UIs on Android

There are two types of web UI facilities on Android that apps commonly use to retrieve an OAuth token: the mobile browser and WebView widgets. Although OAuth is meant to prevent apps from directly accessing users' password data, neither approach protects passwords very well.

A WebView is a system UI widget that serves as an embedded browser for loading URLs within an app. WebViews are widely used in mobile apps to render all manner of web content. WebViews support basic browser functionality such as navigation and JavaScript execution. They also support an API that allows an app to interact with and customize the WebView's output.

There are several ways that a malicious app can control or monitor its WebViews. First, the app can register event listeners to hijack web content or snoop on all interactions with a page. Second, the app can inject JavaScript code into the WebView that is capable of passing information back to the host app. As a result, passwords input through an app's WebView are not protected from the app itself.

On the other hand, a mobile browser is a stand-alone app that can be invoked by other apps using IPC. Unlike a WebView, third-party apps have limited control over a mobile browser. To invoke the mobile browser, an app simply issues a Binder IPC request containing the URL the browser should open. The app yields control to the browser and waits in the background for the browser session to complete. After finishing its work, the mobile browser can be redirected back to the app. The problem with retrieving an OAuth token using the mobile browser is that an app can direct the browser to open any URL, including malicious web pages designed to steal a user's password.

Since many apps use the mobile browser and WebViews to retrieve OAuth tokens, one of VeriUI's primary goals is to interoperate with these interfaces as much as possible. Ideally, non-malicious apps could run on a VeriUI device without changing any app code at all.

2.3 ARM TrustZone

TrustZone [16] is a set of extensions for trusted computing supported by ARM processors since ARMv6, including ARM Cortex A8, A9, and A15 processors. TrustZone's security architecture is based on a "Two Worlds" approach. One world is called the Normal World or rich execution environment (REE), and the other is called the Secure World or trusted execution environment (TEE).

TrustZone allows a processor to efficiently run two operating systems at the same time. A rich operating system, such as Android, and all user-installed apps run in the Normal World, while a secure operating system and secure applications run in the Secure World. Software stacks in the two worlds can be bridged through a secure monitor call (SMC). A TrustZone monitor that executes in the Secure World manages switching between worlds. When an SMC executes, the hardware jumps to the TrustZone monitor to perform a secure context switch in the Secure World. The TrustZone monitor can also be entered by configuring it to handle certain hardware interrupts.

TrustZone supports secure boot, ensuring that the system always boots into the Secure World. After the secure operating system boots, the Secure World yields to the Normal World by loading the bootloader for the rich operating system. When the boot process for the entire system finishes, the Normal World can use an SMC to invoke services in the Secure World.

Unlike secure hardware such as an external TPM [5] or MTM module TrustZone does not provide simple primitives for binding, sealing, and remote attestation. Instead, vendors must install keys and other sensitive data in memory that is only addressable by code in the Secure World. In particular, the two worlds are separated by a special thirty-third address line on the system bus. Untrusted code from the Normal World cannot access protected memory pages or peripheral registers of the Secure World when this bit is set. As a result, even if the rich operating system in the Normal World becomes compromised, it cannot read or tamper with data and code in the Secure World.

VeriUI uses the TrustZone to create a tamper-proof environment for inputting and transmitting users' password data. In addition, the secure kernel can use its protected resources (i.e., a vendor-installed public-key pair) to generate a signed attestation that includes a cryptographic hash of the Secure World's system software as well as information about the user's login request (e.g., a username-password pair and the intended domain).

3. TRUST AND THREAT MODEL

The goal of attested login is to prevent an attacker from logging into a service with a stolen password. It does this in three ways.

First, a device will only generate a signed attestation when a user inputs a password within the Secure World. Second, the Secure World only reveals password data to trusted servers. Third, a service that supports attested login will only authorize a login request if it is accompanied by an appropriate attestation.

Our trust model is rooted in the hardware isolation provided by the ARM TrustZone. We do not trust the apps or the rich operating system, but assume that all code in the Secure World is trustworthy. Thus, our trusted computing base (TCB) consists of a TrustZone monitor, secure kernel, as well as any software that runs on top of the secure kernel. VeriUI provides no guarantees if the TCB becomes compromised. If the rich operating system becomes compromised, the TrustZone ensures the integrity and confidentiality of code and data residing in the Secure World. For example, the TrustZone ensures that any keys used to generate an attestation cannot be tampered with and do not leak. However, VeriUI cannot prevent a compromised Normal World from launching a denial-of-service attack against the Secure World.

The TCB is responsible for generating attestations that describe the hardware and software state of the device, a user's login credentials (e.g., username and password), where the user intended their credentials to be sent (e.g., an administrative domain such as Facebook), and potentially other contextual information. The TCB is also responsible for enforcing the user intent described by the attestation (i.e., ensuring that passwords are only sent to the intended servers). The TCB is meant to be a relatively small code base that can be certified by services themselves or by trusted third parties.

Unfortunately, VeriUI cannot force a user to input passwords only within the Secure World. For example, a malicious app in the Normal World could try to trick a user by spoofing the UI of the Secure World [10]. Of course, login attempts from outside of the Secure World cannot be properly attested and could be rejected by a service. However, once in possession of a user's password, an attacker could manually login as the victim via the Secure World on another device.

One way to prevent such an attack is to make the Secure World more difficult to spoof. For example, CloudTerminal [12] 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 approach is to utilize additional contextual information in the attestation. For example, the secure operating system could attest to login-time GPS coordinates, WiFi-scan results, or an image from the front-facing camera. A service could then check one or more of these data items before authorizing a request.

Furthermore, a service could combine contextual information with data from past logins. For example, the first time a user tries to login to a service from a device, the service might require that the device be located within a specific geographic area (e.g., the user's home country, state, or city); perhaps subsequent attested logins from that device could be performed anywhere. Our current prototype does not support this use of "trusted sensing" [11], but attested login is general enough to allow a rich set of authorization policies.

4. DESIGN

As we discussed in Section 2, neither WebView widgets nor the default web browser can protect passwords from malicious apps. In VeriUI, we provide a separate service called a SecureWebKit that handles sensitive user input in place of WebViews and the browser. SecureWebKit is a lightweight WebKit engine that parses simple web pages and completes web requests. VeriUI provides it as a general service for third-party apps. A SecureWebKit is isolated from any apps that call it, and it provides a trustworthy UI for handling sensitive data such as passwords.

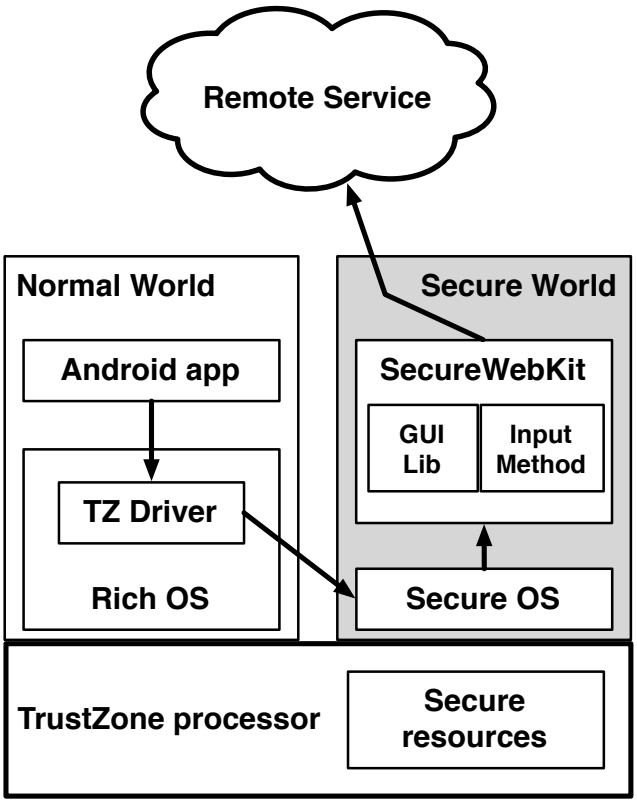

Figure 1: VeriUI architecture

Figure 1 shows VeriUI's architecture, with the rich operating system running in the Normal World and the secure operating system running in the Secure World. VeriUI exposes a narrow API to untrusted apps for communicating with the SecureWebKit service running in the Secure World. To submit a request to the SecureWebKit, an app passes a message to a TrustZone driver in the rich operating system, which switches to the Secure World and forwards the message to the secure operating system.

The secure operating system is a lightweight kernel with minimum driver support (e.g., network, display, and touchscreen). The Secure World provides a simple GUI library and input methods for building trusted apps in the Secure World. The SecureWebKit runs on top of this software stack and manages interactions with remote services.

VeriUI's SecureWebKit is a stripped-down web browser providing basic functionality for handling sensitive operations such as acquiring OAuth tokens. For example, SecureWebKit does not support JavaScript or CSS. Rather, SecureWebKit only supports basic HTML parsing, link navigation, and only communicates over HTTPS. Unlike Android WebViews, the SecureWebKit is completely isolated from untrusted apps by the TrustZone.

Untrusted apps invoke the SecureWebKit by passing a target URL to the secure kernel; this interface is very similar to the one currently used by Android apps to invoke the mobile browser. After the TrustZone driver completes a switch to the Secure World, the secure kernel forwards the target URL to the SecureWebKit.

Figure 2: Domain selection UI

To prevent phishing attacks, VeriUI provides a domain-selection UI for explicitly capturing user intent. Before SecureWebKit opens the target URL, it displays a list of web domains and asks the user which domain she is trying to access. Figure 2 shows the domain selection UI of our current prototype.

After the user chooses a domain, SecureWebKit checks whether the SSL certificate of the target URL matches the domain selected by the user. If there is a mismatch, SecureWebKit ends the session by ignoring the target URL and alerting the user. Ensuring that the user's intent matches the target URL prevents malicious apps from directing the SecureWebKit to dangerous URLs as can be done with the Android mobile browser.

If the user's intent matches the target URL, the SecureWebKit loads the target URL and restricts all future communication to the selected domain. Because most sensitive operations only involve a single service and can be completed in several steps, a single domain policy is reasonable. Furthermore, restricting communication to the intended domain ensures that only trusted servers receive sensitive data.

A key benefit of attested login is that it allows services to verify the code and environment that handles sensitive data. To provide this feature, VeriUI loads an attestation identify key (AIK) pair into the Secure World memory from the hardware ROM when the secure operating system boots. VeriUI uses the AIK to sign secure messages and generate remote attestations. The secure kernel provides an API for the SecureWebKit to generate attestations. We assume the ROM is pre-installed with a key pair by the manufacturer and is only accessible from within the Secure World during secure boot.

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. After the SecureWebKit loads a web page from a service, users type in their password and submit their request to the server. Before sending these requests, SecureWebKit asks the secure kernel to generate an attestation. The attestation proves to the service that user input was handled in a secure environment. Services verify the attestation and decide whether to authorize the request or not.

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 describes the initial target URL and the secure domain selected by the user. The post URL and post data should match the request to the server. The certificate also includes a timestamp to prevent

```
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 3: Format of attestation

replay attacks. The system part of the attention contains hashes of the system software.

The SecureWebKit exits when the user is done interacting with the remote service, and returns any data it receives (e.g., an OAuth token) to the third-party app.

5. IMPLEMENTATION

We built a VeriUI prototype for an ARM SOC equipped with ARM Cortex A8 cores. For the secure kernel, we eliminated functionality from a stock Linux kernel to reduce the TCB as much as possible. We left only the kernel, the display driver, the touchscreen driver, and a network driver. Due to the limitations of the development board, we currently use a wired Ethernet network interface.

We used u-boot [6] to start the two systems on the ARM SOC, and developed a monitor to control world switches. During system start, the secure kernel separates memory for the Secure World and Normal World. It also reserves a portion of memory as shared to bridge communication between the two worlds. We modified the Android Linux kernel to support our TrustZone driver. To initiate an attested login, an app writes the target URL to /dev/tz, and the TrustZone driver passes the URL to the secure kernel. After the user authenticates through the SecureWebKit, the app can then read the client token from /dev/tz.

We used Qt [3] as our GUI lib in the Secure World, since QT has good support for WebKit. The SecureWebKit was based on QtWeb [4]. We ported QtWeb to the secure operating system and implemented all the security and anti-phishing features described in Section 4. We removed all advanced features in QtWeb, added the user-intent selection interface when SecureWebKit starts, and added attestations when sending requests to remote servers. We also developed a touchscreen input method for the Secure World to help users input data. We used OpenSSL [2] for generating attestations.

Finally, as a proof of concept we developed a simple Android app that supports OAuth login to Facebook and Twitter. The app is built on top of the standard SDKs provided by Facebook and Twitter, and we modified these libraries to make secure procedure calls to the SecureWebKit. This app was straightforward to build and re-

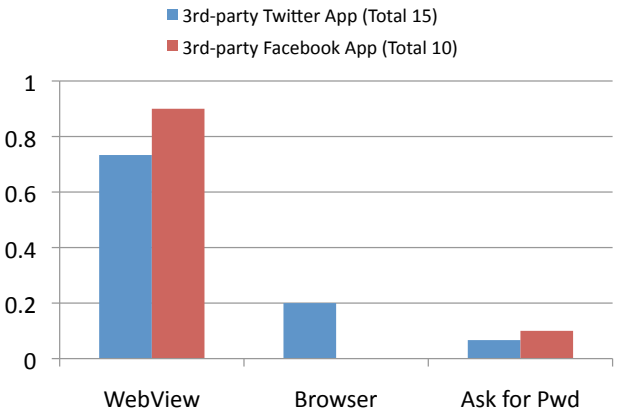

Figure 4: OAuth app study

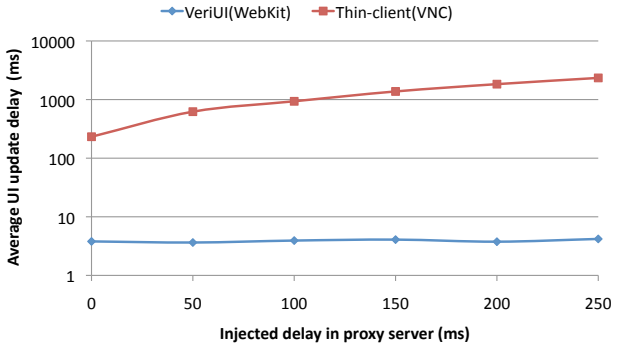

Figure 5: UI responsiveness

inforced our belief that apps would not require much modification to take advantage of attested login.

6. EVALUATION

VeriUI should 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. In addition, VeriUI must have acceptable performance. To understand its performance, we compared its UI responsiveness to the thin-client approach embodied by CloudTerminal [12].

6.1 App study

To better understand how current mobile apps request OAuth tokens, we performed a survey of existing Android apps from Google Play. We examined the 15 popular third-party Twitter apps [1] and 10 popular third-party Facebook apps [2]. 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 services since Facebook and Twitter are among the most popular platforms for third-party app developers.

We were most interested in whether apps used Android WebView widgets, the default browser, or handled OAuth internally. Our results are in Figure 4. Among the 25 apps in our study, we found 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.

6.2 UI responsiveness

As mentioned previously, CloudTerminal addresses similar problems as VeriUI, but adopts a thin-client approach. This results in a small client-side TCB, but leaves users 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 a 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 placed our client, VNC server, and a proxy server on the same campus network and redirected all traffic from the development board through the proxy. This allowed us to inject latency at the proxy server and explore the effect of network latency on user interactions. The average round-trip time between the client and VNC server without any injected latency was 29ms.

We then used the VNC client and SecureWebKit to open the same login web page. When we used the VNC client to open the web page, a browser in the VNC server fetched and rendered the web content, but it was viewed and manipulated on the client. After the login page loaded, we used the Secure World keyboard to see how slowly typed-in characters were 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 when the client read the update from the server and displayed it. We injected latencies ranging from 0ms to 250ms on all the traffic through the proxy server to test responsiveness.

Figure 5 shows the average keystroke latency from 10 trials for each amount of injected network latency; standard deviations were within 12% for each group of trials. Unsurprisingly, VeriUI's approach provides a much more responsive UI than the thin client approach. For an injected delay of 250ms, the VNC client took over two seconds to respond to a keystroke, whereas VeriUI responded within 5ms regardless of the network latency.

7. RELATED WORK

TLR [14] provides a framework for running trusted applications on smartphones by splitting the app's functionality between the Normal World and Secure World. A secure application can package the code handling sensitive data into a software component called a TrustLet and run it in a secure environment called a TrustBox in the Secure World. VeriUI's approach to protecting passwords is similar; VeriUI effectively uses a web browser for its trusted runtime rather than the .NET MicroFramework. However, the primary difference between VeriUI and TLR is that VeriUI must securely handle user inputs (i.e., passwords) and communication with remote servers, whereas TLR does not support direct I/O within the Secure World.

[1] TweetCaster, UberSocial, TweetDeck, Plume, Seesmic, HootSuite, Slices, Janetter, Scope, Echofon, TwitPal, Tuippuru, Twidere, Feel on!, and Tweedle

[2] Go!Chat, TweetDeck, Video to Facebook, Seesmic, Friendcaster, HootSuite, BeejiveIM, Facebook Pages Manager, Sync.ME, and Contact Sync

[11] uses ARM TrustZone to implement two software abstractions for trusted sensor applications: sensor attestation and sensor seal. VeriUI's attestations are very similar to a sensor attestation; both provide evidence of a trustworthy chain of custody for data collected on a mobile device. However, trusted sensing alone is insufficient for securing user passwords, since VeriUI must also ensure that attestations are shared only with the intended domain.

CloudTerminal [12] provides a secure client for accessing sensitive applications in the cloud. Like VeriUI, CloudTerminal provides strong isolation from a compromised operating system. CloudTerminal's approach is to move the entire client application (e.g., a web browser) to a trusted server and to connect to the server using a thin-client in the Secure World. The client side handles user input from a keyboard and mouse, forwards those inputs to the server, and receives and renders display updates for the user. As we have discussed, CloudTerminal has a smaller TCB than VeriUI, but suffers from poor interactivity when network latency is high.

Flicker [13] describes performing SSH authentication using a small piece of application logic (PAL) that runs in a secure environment on a server. However, a compromised client could still leak a user's password. Ideally, a service would combine Flicker's server-side protections with VeriUI's client-side protections.

ScreenPass [10] 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. The primary advantage of VeriUI over ScreenPass is that VeriUI has a much smaller TCB.

Like VeriUI, PhoneAuth [7] tries to navigate the inherent tradeoffs between TFA and conventional password-based authentication. PhoneAuth relies on a user's mobile phone to generate an identity assertion for a paired web browser whenever the user tries to login to a service through the browser. An identity assertion binds the user's identity (represented as a public key stored on the phone) to the web browser (also represented as a public key). The primary difference between PhoneAuth and VeriUI is that attested login provides services with evidence that a password was handled within a trustworthy environment, whereas identity assertions prove that a user's mobile phone was nearby when they input their password.

Finally, a paper recently compared the trade-offs between two-factor authentication and trusted execution environments like ARM TrustZone [17]. The paper articulates many of the same insights that motivated our work. Furthermore, the paper's models predicted that using a trusted execution environment in lieu of a second factor would present many of the challenges that arose while designing and building VeriUI.

8. CONCLUSION

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 have 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.

Acknowledgements

We would like to thank our shepherd, Michael Piatek, and the anonymous reviewers for their helpful comments. Our work was supported by Intel through the ISTC for Secure Computing at UC-Berkeley as well as the National Science Foundation under NSF awards CNS-0747283 and CNS-0916649.

9. REFERENCES

[1] Oauth 2.0.
[2] Openssl: The open source toolkit for ssl/tls.
[3] Qt project.
[4] Qt web - portable web browser.
[5] Tpm.
[6] U-boot - the universal boot loader.
[7] A. Czeskis, M. Dietz, T. Kohno, D. Wallach, and D. Balfanz. Strengthening user authentication through opportunistic cryptographic identity assertions. In *CCS*, 2012.
[8] C. Herley and P. C. van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10(1), 2012.
[9] X. Jiang. Smishing vulnerability in multiple android platforms. `http://csc.ncsu.edu/~jiang/smishing.html`.
[10] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. P. Cox. Screenpass: Secure password entry on touchscreen devices. In *MobiSys*, 2013.
[11] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *MobiSys*, 2012.
[12] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica. Cloud terminal: secure access to sensitive applications from untrusted systems. In *USENIX ATC*, 2012.
[13] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Operating Systems Review*, volume 42. ACM, 2008.
[14] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Trusted language runtime (tlr): enabling trusted applications on smartphones. In *HotMobile*. ACM, 2011.
[15] I. Security. New android trojan can thwart two-factor authentication, May 2013. `http://www.infosecurity-magazine.com/`.
[16] A. S. Technology. Building a secure system using trustzone technology.
[17] R. van Rijswijk-Deij and E. Poll. Using Trusted Execution Environments in Two-factor Authentication: comparing approaches. In *Open Identity Summit*, volume 223 of *Lecture Notes in Informatics, LNI*, pages 20–31. Springer, 2013.
[18] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Security and Privacy*, 2012.