Secure Input Overlays: Increasing Security for Sensitive Data on Android

by

Louis Sobel

S.B. CS MIT, 2014

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2015

Author: ______________________________
Department of Electrical Engineering and Computer Science
May 20, 2015

Certified by: ______________________________
Srini Devadas
Edwin S. Webster Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: ______________________________
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Secure Input Overlays: Increasing Security for Sensitive Data on Android

Louis Sobel

Submitted to the Department of Electrical Engineering and
Computer Science on May 20, 2015 in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering in
Electrical Engineering and Computer Science

Abstract

Mobile devices and the applications that run on them are an important part of people's lives. Often, an untrusted mobile application will need to obtain sensitive inputs, such as credit card information or passwords, from the user. The application needs these sensitive inputs in order to send them to a trusted service provider that enables the application to implement some useful functionality such as authentication or payment. In order for the inputs to be secure, there needs to be a *trusted path* from the user, through a *trusted base* on the mobile device, and to the remote service provider. In addition, remote attestation is necessary to convince the service provider that the inputs it receives traveled through the trusted path.

There are two orthogonal parts to establishing the trusted path: local attestation and data protection. Local attestation is the user being convinced that they are interacting with the trusted base. Data protection is ensuring that inputs remain isolated from untrusted applications until they reach the trusted service provider. This paper categorizes previous research solutions to these two components of a trusted path.

I then introduce a new solution addressing data protection: Secure Input Overlays. They keep input safe from untrusted applications by completely isolating the inputs from the untrusted mobile application. However, the untrusted application is still able to perform a limited set of queries for validation purposes. These queries are logged. When the application wants to send the inputs to a remote service provider, it declaratively describes the request. The trusted base sends the contents and the log of queries. An attestation generated by trusted hardware verifies that the request is coming from an Android device. The remote service provider can use this attestation to verify the request, then check the log of queries against a whitelist to make a trust decision about the supplied data.

Thesis Supervisor:	Srini Devadas
Title:	Edwin S. Webster Professor of Electrical Engineering and Computer Science

1 Introduction

Mobile devices and the applications that run on them are an increasingly prevalent way that people use computers. Many applications use the Application Programming Interface (API) of some other service to provide useful functionality such as authentication or payments. The application's use of these APIs often requires the user to enter sensitive inputs into an untrusted application, which will then send them to a trusted service provider.

This paper presents Secure Input Overlays, a mechanism for applications—without requiring the user's trust—to obtain and pass on sensitive information such as passwords and credit cards. Additionally, the system provides the trusted service provider with an attestation proving that the information was handled securely.

Applications that need to send sensitive inputs to a remote service provider include those that support external authentication providers and those need to pass payment information to a payment processor. There are security issues that arise with these applications. A user might fall victim to a phishing attack, where an application masquerades as another application. A malicious application could steal a user's sensitive inputs or accidentally leak them.

Establishing a *trusted path* from the user to a remote server requires some trusted base on the mobile device that can protect data from untrusted applications. Establishing the trusted base is just part of the problem. The other part is *local attestation*: the user must correctly identify the trusted base. These two parts are orthogonal. In addition, is important for applications to be able to perform computations on sensitive inputs, such as client-side validation, in order to provide a rich user experience. Furthermore, it is useful for the service provider to know how inputs it receives were handled.

There are existing research solutions for local attestation and data protection on mobile devices. In this paper, I categorize and describe these projects. Local attestation solutions either use a secure attestation sequence, personalization, or a dedicated interface element. Data protection schemes rely on a trusted execution environment, taint tracking, or shadowing. Local attestation and data protection are two orthogonal legs of a trusted path, so providing a solution to just one is useful.

Secure Input Overlays are a new method of protecting data that enables applications to send sensitive inputs to a remote service provider without viewing them while still being able to perform whitelisted computations. Secure Input Overlays do not provide local attestation. A Secure Input Overlay is an interface element in the address space of the trusted operating system that appears on top of an untrusted application. This is not noticeable to the user, who enters sensitive inputs directly into the overlaid window. Inputs are stored in a *hidden buffer* that is completely isolated from the untrusted application. The application can query the contents of hidden buffers. These queries are logged, and this log is included when an application exfiltrates the contents of the hidden buffer to a remote server. The message going to a remote server is attested using trusted hardware and a remote attestation protocol.

To demonstrate the feasibility of this approach, I implemented Secure Input Overlays as an extension to the Android mobile operating system. I then implemented two example applications, a login form and a payment form, that use Secure Input Overlays for sensitive inputs while still providing an experience comparable to that provided by existing applications. These case studies show the feasibility and simplicity of integrating Secure Input Overlays into applications.

The rest of this paper is as follows. Section 2 expands on the problem of sensitive mobile inputs and the authentication and payment examples given. Section 3 gives a taxonomy of existing solutions to local attestation and data protection on mobile devices with comparisons to similar desktop solutions. Section 4 details the design and implementation of Secure Input Overlays. Section 5 gives two case studies of applications built using Secure Input Overlays. Section 6 evaluates the design and implementation. Section 7 discusses other relevant previous research, and Section 8 suggests future work. Section 9 concludes.

2 Sensitive Mobile Inputs

The fact that untrusted mobile applications need to handle sensitive inputs introduces security challenges. To provide motivation for Secure Input Overlays, this section gives examples of applications that need sensitive inputs and the consequent security issues. Applications on mobile devices allow users to do all sorts of useful things—order food, date, pay their bills, shop. It is often the case that an application will use an external service that requires sensitive information from the user in order to provide some functionality, such as authentication or payments. The application needs to obtain the sensitive inputs and should not do anything more to them than validation before sending them to the service provider's Application Programming Interface (API). Establishing a trusted path from the user to the service provider has two security problems: *local attestation* and *data protection*.

2.1 Examples

Two types of services that applications can use are authentication services and payment processing services. In order to use these services, the application needs to collect sensitive data, such as a password or credit card number, validate it, then send it to the service provider. The user trusts the service provider with this data. The application is not trusted.

2.1.1 Authentication

Authentication providers allow applications to access a service on a user's behalf. This is good for end-users, as they can use multiple applications with fewer credentials. It is also good for applications, as it lowers the barrier to new users, who do not have to set up a new account for each application they use. Facebook, Google, and Twitter [1, 2, 3] are three of the many service providers that have publicly available APIs for this purpose.

It is important for applications to be able to validate the credentials a user enters. An application might check that the contents of an email field look like a valid email address or that an entered password conforms to a policy. As described in [4], many mobile applications enforce constraints on the composition and length of a password.

A straightforward implementation that applications could use to have a user authenticate with a service provider would be to have the user directly enter their credentials. The application could then use them to access the service provider on behalf of the user. This causes two problems. First, the untrusted application would need to store a plaintext copy of the credentials. With every new application, this would increase the likelihood that they leak. This also causes the second problem: it would not be possible to individually revoke one application's access. The only way to revoke access would be for the user to change their password, which would have the effect of revoking access for all applications.

A protocol called OAuth [5] is a different implementation that service providers use. It solves the above problems by isolating credentials from untrusted applications. OAuth is a protocol through which applications do not need to store usernames and passwords but can still authenticate and take actions on a user's behalf. It is a multi-step flow that relies on a trustworthy user-agent to provide isolation. At the end of the process, applications receive a token that can be used to act on behalf of a user and is individually revocable. It was designed for use on web browsers, which provide interdomain isolation through the use of the Same-Origin Policy [6]. On mobile devices, however, OAuth implementations lose their security properties because there

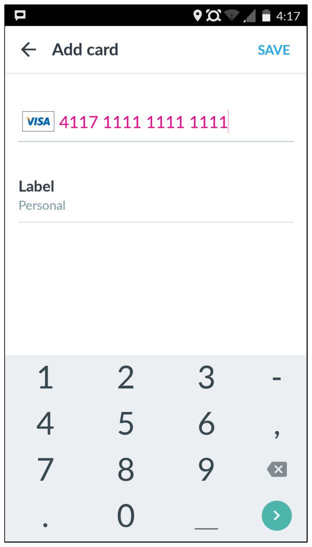


Figure 1: Payment information form from Lyft's [9] Android application.

is no sure way to provide isolation. Many service providers publish OAuth libraries for application developers that use a mobile browser for isolation [1, 2, 3], but an untrusted application could still inject code into this browser and circumvent the isolation [7, 8].

This means that the user, ultimately, must trust the application to which they are providing credentials. The application has access to the user's sensitive inputs and has no constraints on what it can do with them. The service provider also, without the isolation properties web browsers provide, is unable to be certain that the inputs it receives were handled securely. It is in the best interest of both the service provider and the user that the sensitive inputs be handled securely. The untrusted applications should perform only the processing necessary to validate inputs and send them to the service provider. If the authentication provider knew with certainty that an application did not store credentials, the OAuth protocol could be replaced with a simple one where the application exchanged provably not-leaked credentials for a revocable token.

2.1.2 Payments

An application using a payment processing service is another example of a case when a user needs to enter sensitive input into an untrusted application that should just be validated and sent to a remote service provider. Mobile applications need to collect payment information in order to collect money from users. The application might only need to make a one-time charge, or may need to somehow store the payment information for future use. There are a number of services that allow applications to do this, including Braintree [10] and Stripe [11].

One of the benefits for an application of using a payment service is that the application then does not need the infrastructure to securely store credit card information. The Payment Card Industry Data Security Standard [12] (PCI DSS) describes a strict set of rules that applications that store credit cards must follow. Complying with these rules is arduous and expensive. Applications can avoid storing payment information by using the tokenization features provided by payment services. Tokenization allows an application to exchange payment information for an opaque reference to it.

An application that uses a service provider to tokenize payment information only needs to access the information itself in order to validate it and send it to the service provider. Figure 1 is a screenshot from the mobile application Lyft's [9] credit card adding screen. The entered credit card number is invalid and highlighted in red. The application further improves its usability by giving feedback on the brand of card being entered.

The user must trust that applications properly handle the payment information they enter. It is in the user's best interest that the application does not do anything with the information other than validating it and sending it to the payment service. This security is also a goal of payments services; credit card fraud is an issue that they must deal with. Without proof that the payment information they received was properly handled, payment services could deny a transaction or take further administrative action.

2.2 Security Issues

As the above examples suggest, there are security issues that arise when a user needs to trust that an application will not mishandle sensitive inputs. The structure of the *trusted base* is important when considering these issues. The trusted base is what software and hardware on the mobile device is trusted. It is an intermediary between the user (who is trusted) and the service provider (who is as well). The application using the service provider is not trusted. The fundamental security problem is establishing a *trusted path* from the user to the service provider through this trusted base.

There are two orthogonal parts to establishing an end-to-end trusted path. The first is local attestation, which is establishing a trusted path from the user to the trusted base. The second is protecting data from untrusted applications and getting it securely to the remote service provider. The orthogonality of these two components is also discussed in [13]. Additionally, remote attestation is necessary for the service provider to be convinced that the path some data took was trusted. The following three sections describe these issues.

2.2.1 Local Attestation

Local attestation is the user being convinced that the software they are interacting with is part of the trusted base. With the exception of the occasional savant, most humans cannot mentally perform asymmetric cryptographic operations, making local attestation fundamentally a Human–Computer Interaction problem. Existing research into providing local attestation is discussed in Section 3.1. Without local attestation, the user is vulnerable to *phishing*.

Phishing is when a malicious application masquerades as a trusted one in order to manipulate the user. It is a widespread problem on the web and mobile devices. In the first half of 2014, APWG, a nonprofit organization tracking phishing on the internet, received reports of 87,901 unique domain names hosting phishing websites [14]. It is a problem for mobile device users as well. Mobile devices present particular challenges to preventing phishing, due to their small screen sizes, software keyboards, and full screen, immersive applications. The many different attack vectors on mobile devices and examples of applications that exploit them are described in [15, 16, 17].

2.2.2 Data Protection

Even if a user correctly identifies the application they are interacting with, it is still possible for that application to insecurely handle their data. The second leg of establishing a trusted path from the user to a remote service provider is structuring the trusted base so that untrusted applications can neither accidentally nor maliciously misuse sensitive data. Existing research into protecting data on mobile devices is discussed in Section 3.2. Applications have access to sensitive data both from user inputs and also from mobile operating system APIs that reveal sensitive information like GPS coordinates or contact lists.

Without adequate data protection, users cannot be sure their data will be handled securely. Their sensitive inputs may accidentally be logged, stored, or leaked. They might also be *intentionally* exfiltrated from a device and used in a way not intended by the user. The principle of least privilege [18] suggests that the components of a system should have access to the minimum necessary amount of information. In the use cases described above, the applications do not themselves need access to sensitive inputs, but only need to validate them and pass them on. The possible need for validation makes choosing how much access to allow a tradeoff between isolating data from applications and giving the applications computational freedom over that data.

2.2.3 Remote Attestation

Local attestation and data protection together establish a trusted path, but the service provider who receives inputs might still need proof that those inputs travelled through the trusted path. Remote attestation provides this proof. The ability to distinguish insecure applications from those that can be trusted could increase the security features of service providers and simplify the interface they expose. Examples include removing the need for the OAuth protocol and decreasing the incidence of credit card fraud. For authentication, this process has been referred to as *attested login* [7, 19].

3 Existing Solutions

This section describes existing systems on mobile devices for providing local attestation and protecting users' data. The purpose of this discussion is to give context for Secure Input Overlays and to provide a taxonomy of existing solutions. As previously discussed, establishing a trusted path is a two-step process. First, the user must establish a secure channel to the trusted base. Second, the trusted base must keep inputs secure from untrusted applications.

There have been a number of previous research solutions. Some establish an end-to-end trusted path, handling both local attestation and data protection, while others deal only with one or the other. This section focuses on solutions for mobile devices, but also discusses some relevant examples from other computing environments. A summary of the information in this section is presented in Table 1.

3.1 Local Attestation

Local attestation is the user establishing a secure channel to the trusted base. No matter how well the trusted base protects data, the user still has to correctly identify the trusted base in the presence of malicious applications that can spoof the interface of trusted components. Without local attestation, a user is vulnerable to phishing attacks. Solutions break down into three categories: those that use a secure attestation sequence, those that use personalization, and those that use a dedicated interface element.

3.1.1 Secure Attestation Sequence

A secure attestation sequence is an action the user must take to securely communicate to the trusted base that sensitive inputs are about to be entered. In order to work, the input channel needs to be part of the trusted

	Local Attestation			**Data Protection**		
	Secure Attestation Sequence	*Personalization*	*Dedicated Interface Element*	*Trusted Execution Environment*	*Taint Tracking*	*Shadowing*
SpoofKiller [20]	◐	-	-	-	-	-
ScreenPass [21]	◐	-	-	-	-	-
Marforio et al. 2015 [16]	-	◐	-	-	-	-
TIVOs [22]	-	◐	-	-	-	-
Bianchi et al. 2015 [17]	-	-	◐	-	-	-
US Patent 8,479,022 [23]	-	-	◐	-	-	-
TrustUI [24]	-	-	●	●	-	-
GuarDroid [4]	-	●	-	-	-	●
VeriUI [7]	-	-	-	◑	-	-
OAuth Manager [8]	-	-	-	◑	-	-
TaintDroid [25]	-	-	-	-	◑	-
SpanDex [26]	-	-	-	-	◑	-
PiOS [27]	-	-	-	-	◑	-
AppFence [28]	-	-	-	-	-	◑

Table 1: Summary of existing solutions for mobile devices discussed in Section 3. Local attestation and data protection are orthogonal problems in creating a trusted path. Existing works marked with ● implement a full trusted path. Those marked with ◐ only implement local attestation, and those marked with ◑ only data protection. Local attestation solutions either use a secure attestation sequence, personalization, or a dedicated interface element. Data protection schemes use either a trusted execution environment, taint tracking, or shadowing.

base. A secure attestation sequence convinces the user that they are only communicating with the trusted base. They are actions that a user must deliberately remember to make for every sensitive input, which limits their effectiveness.

There are examples of secure attestation sequences on desktop computers. Using the CTRL+ALT+DEL key combination to bring up a login screen in Microsoft Windows is one example [29]. Linux has a corresponding key combination [30]. PwdHash [31] and Bumpy [19] provide local attestation on desktop web browsers. In these designs, keyboard input handlers are part of the trusted base, and the user must prefix sensitive inputs with "@@". Examples of secure attestation sequences on mobile devices are SpoofKiller [20] and ScreenPass [21].

Even though many mobile devices use touchscreens, most still have a physical power button. SpoofKiller uses the power button as a secure attestation sequence. Hardware buttons are under the control of the operating system, which is part of the trusted base, so this attestation sequence cannot be intercepted. On most phones, pressing the power button causes the screen to go blank. SpoofKiller modifies this behavior. When a whitelisted application is receiving input, clicking the power button does *not* cause the screen to go blank. This provides obvious feedback to the user: if they are trained to expect password boxes not to disappear when clicking the power button, they can readily identify an untrusted application when a click of the power button causes the application to disappear.

ScreenPass uses a secure attestation sequence in the form of a button integrated into a secure software keyboard. Users press this button to identify a particular input element as one that requires a password. To help train the user, ScreenPass includes an integrated password manager. They ensure that their software keyboard is part of the trusted base by performing image recognition on the screen to prevent applications from spoofing the keyboard and its associated secure attestation button.

3.1.2 Personalization

Personalization uses a human-understandable secret shared between the user and the trusted base. A canonical example of personalization on the web is SiteKey, used by Bank of America and Vanguard [32]. Users set up an image to appear when logging in and are trained to abort a login process if this image is not shown. The goal of this is to decrease phishing—an attacker cannot possibly guess what image the user chose and so cannot accurately masquerade as the trusted website. SiteKey's effectiveness relies on the user remembering to check for the image. Some studies have shown that this is not a resilient method of local attestation, especially when the attacker substitutes a blank image or, more nefariously, a message stating something like "This image cannot be displayed due to technical difficulties" [33, 34]. However, a recent paper [16] tested a similar scheme, specifically for mobile applications, that decreased the frequency of users becoming phishing victims by almost 50%. Two other local attestation projects that use personalization on mobile devices are TIVOs [22] and GuarDroid [4].

In TIVOs, on a user's first use of their phone, they take a picture of something personal to them. This image is stored so that only the trusted base (the Android operating system) can access it. TIVOs modifies the software keyboard so that when a user goes to type input into an application, the keyboard shows a badge containing the name of the application combined with the previously taken picture. This badge is unforgeable without the picture. This enables a user to know with certainty what application they are interacting with.

GuarDroid uses a similar approach. Rather than using images, it has the user select a phrase, consisting of a pair of words, each time the phone is booted. It uses a minimal recovery operating system for choosing the phrase to ensure security during that process. Like the picture in TIVOs, this phrase is a secret shared between the user and the trusted base. It is displayed on a modified keyboard during password entry. The user is expected to check for the presence of the phrase to ensure they are interacting with the trusted base.

3.1.3 Dedicated Interface Element

Using a dedicated interface element for local attestation is having some unforgeable way for the trusted base to communicate to the user. An example on desktop computers is Qubes [35]. The trusted base of Qubes maintains exclusive control over the color of a window's frame. Based on the displayed color, the user can make a trust decision about the window's contents. Another example on desktop computers is how web browsers securely communicate the trust status of a website's SSL certificate [36, 37]. At least one paper [38] has found these SSL status indicators ineffective.

It is feasible to dedicate screen space for the trusted base on mobile devices. Bianchi et al. implement such a scheme in [17]. But, as pointed out in [15], this is not an ideal approach on mobile devices. Screen space is very limited. Dedicating some of it to the trusted base would preclude the use of full screen applications, which would inhibit the immersive experience provided by applications like video players and games.

Another possible implementation of the dedicated interface element is through hardware. Two research projects into local attestation of public computers [39, 40] use the user's trusted mobile device as a dedicated interface element for the trusted base. Bumpy [19] takes the same approach. Clearly, a mobile device cannot expect to use another mobile device as a dedicated interface element. Another option, as patented in [23], is to use an additional hardware interface element such as an indicator light (LED) controlled by the trusted base. TrustUI [24] furthers this idea. They propose using a multicolor LED whose color is synchronized with the background color of the trusted interface. This would allow users to detect the presence of a malicious overlay.

3.2 Data Protection

Local attestation is establishing a secure channel between the user and the trusted base. The orthogonal second step of creating an end-to-end trusted path is protecting data from untrusted applications once it is in the trusted base. Protecting data from applications results in a tradeoff with their functionality. Some solutions completely isolate sensitive data. Others allow the application limited interaction with it. This section categorizes existing trusted bases that protect data on mobile devices. There are three types of solutions: using a trusted execution environment, taint tracking, and shadowing.

3.2.1 Trusted Execution Environment

Trusted execution environments protect data from untrusted applications by completely isolating the data and handling it entirely within the trusted base. The Same-Origin Policy [6] that web browsers enforce, where a trusted web page can be isolated from all others, is what makes OAuth secure on the web. Mobile operating systems do provide interapplication isolation, which enables a trusted application to be used instead of OAuth as long as it is already installed. For example, Facebook, when possible, enables users to authenticate to applications using its installed mobile application rather than a remote API [1]. The OAuth Manager for Android prototyped in [8] acts as a central trusted base for authentication. Rather than using an application or web browser as the user-agent for the OAuth flow, users instead use a fully isolated client built into the operating system.

VeriUI [7] and TrustUI [24] go a step further by not trusting the operating system at all. They both use ARM TrustZone [41] technology to start a new, trusted operating system when an application wants a user to enter sensitive inputs. TrustZone completely isolates the trusted operating system from the untrusted one. This isolation enables the secure entering of inputs even when the main operating system has been compromised. VeriUI's trusted base is an entire Linux kernel and associated drivers, with a simplified browser running on top. TrustUI achieves a much smaller trusted base by reusing the device drivers from the untrusted operating system. It has a number of features that enable security under these conditions. One is the external-LED-color synchronization technique described above. Another is randomizing the layout of the on-screen keyboard in the trusted operating system. This prevents the untrusted drivers from using the coordinates of touch inputs to learn what a user is typing.

3.2.2 Taint Tracking

A trusted execution environment completely isolates data from applications. Taint tracking is at the other extreme, allowing applications complete access to data. However, the trusted base tracks the flow of sensitive data through the application. This tracking allows fine-grained control over data use and immediate detection of any misuse. For instance, an application may be permitted to freely use a sensitive input as long as it remains on the device, but any attempts to exfiltrate that input would be blocked. Taint tracking can be performed through static analysis or via dynamic labeling and monitoring. There has been extensive research into its use. HiStar [42] is an operating system that uses dynamic taint tracking to enforce security policies around how data is shared between processes. On Android, TaintDroid [25] and SpanDex [26] perform dynamic taint tracking. PiOS [27] provides static taint analysis for iOS binaries.

TaintDroid modifies the Android Java runtime to support dynamic taint tracking, from *sources* to *sinks*. When an application obtains data from a source, such as the microphone, the GPS sensor, or the user's contact list, that data is labeled. This label is propagated as the data is used throughout the program. TaintDroid can detect when the data reaches a sink, such as the network interface, and take appropriate action. One flaw in TaintDroid is that it does not track data through *implicit flows*, such as when the branch that a conditional statement takes depends on sensitive data. SpanDex adds this missing implicit flow tracking to TaintDroid. SpanDex focuses on passwords in order to make the constraint-satisfaction problem that it solves tractable by forcing common validation procedures to use established and trusted libraries.

PiOS, unlike SpanDex and TaintDroid, performs *static* taint tracking on *iOS* binaries. It parses the binary to build a control-flow graph, then explores possible flows of data. The authors propose running this when applications are submitted to centralized application distributors. A benefit of static taint tracking is that it requires no operating system modifications or runtime overhead.

3.2.3 Shadowing

Using a trusted execution environment, data is completely isolated from applications. With taint tracking, data is freely accessible, but its use is tracked. Shadowing is in between. With shadowing, untrusted applications do not receive actual sensitive data, but rather something that *shadows* it. This can be fake data or an indirect reference to the data. The tokenization used in OAuth and by payment processors is a form of shadowing. Projects using shadowing on mobile devices are AppFence [28] and GuarDroid [4].

AppFence protects data from mobile applications by modifying Android to not always actually provide the data that an application asks for. Instead, depending on configuration, it provides some fake, shadow version of the data. This allows the user to specify fine-grained controls over what data different applications can access. Providing shadow data (rather than treating disallowed accesses as errors) allows many applications to continue working without modification.

GuarDroid is a system for protecting passwords. Rather than giving applications the actual password a user enters, it provides a token that shadows the actual password. This token is the password encrypted with a key known only to the trusted operating system. The ciphertext is encoded such that it seems just like a password to the application. The application is expected to treat the password as an opaque value and to not notice the difference. GuarDroid uses a man-in-the-middle web proxy that intercepts all outgoing network traffic, scanning for transmission of encrypted passwords. If it finds any, it replaces them with the corresponding plaintext. One problem with this, as discovered by the GuarDroid authors, is that many applications *do* depend on password values satisfying some policy, such as a having a minimum or maximum length or meeting constraints on their composition.

4 Secure Input Overlays

The solution I present in this paper is the use of *Secure Input Overlays*. It addresses the data protection part of the problem of establishing a trusted path from the user to a remote server for sensitive inputs on mobile devices. Local attestation is not covered. I prototyped it by modifying the Android Open Source Project. Users' inputs go straight to the Android operating system, bypassing untrusted applications. Using a shadowing technique, the applications instead get opaque references to sensitive data. The applications can use these references to perform validations on the inputs, and when ready, send them each to exactly one remote server. A remote server receives, in addition to the data, a log of what computations the sending application ran. The inputs and log are verified using an attestation generated by secure hardware. Based on the log and attestation, the server decides whether or not to accept the inputs. The rest of this section gives background on technologies used, then describes the design and implementation in further detail, and finally provides an argument for the security properties that Secure Input Overlays provide.

4.1 Background

Before diving into the design, I first provide background information on some of the technologies on which it relies. Secure Input Overlays are implemented on top of the Android Open Source Project. A remote attestation protocol, relying on simulated trusted hardware, is used to provide a proof to the service provider of how inputs were handled. Extended BSD Packet Filters (eBPF) is a limited machine language used to support flexible computation over sensitive inputs.

4.1.1 Android Open Source Project

The Android Open Source Project (AOSP) [43] is the open-source core of all Android mobile operating systems. It consists of layers of abstraction built on top of the Linux kernel. It provides a good security model and Binder, an interprocess communication (IPC) primitive with useful security attributes.

Structure and Framework Android is built of multiple layers [44]. Near the bottom of the stack is the Linux kernel. The topmost layer is the Android Framework, which application developers use as the API to the system. For example, the framework has APIs for building user interfaces and accessing the network. The framework is implemented in Java. It includes many *system services* that mediate access to resources and manage the operating system. Example services are the `WindowManager`, which provides a windowing system, and the `LocationService`, which provides mediated access to GPS and other location features.

Security Model Android has strong security properties [45]. One of the primary ones is isolation between applications. This is accomplished by giving each application its own Linux user id[1], which enforces privilege-separated access to the system. System services run as a special system user. Another security property is capability-based limits on the resources a particular application can access, such as GPS coordinates, the camera, or the network. Each application requests a whitelist of capabilities, known as *permissions*, at the time it is installed. The possession of a required permission can be checked at runtime when an application attempts to access a particular resource [46].

Binder IPC The technology that enables this cross-process permission checking is Binder interprocess communication (IPC) [47]. Binder's foundation is a modification to the kernel; applications access it through Java bindings in the Android Framework. It enables a process to perform a remote procedure call into another process. These calls include trustworthy attribution: the callee knows which process is making the request and what user id it is running as. The callee can use this information to limit certain methods to only the system user or to decide whether to allow the request based on the permissions of the calling application. Another feature that Binder provides is the ability to generate globally unique identifiers—known as Binder *tokens*—that can be used across processes [48].

4.1.2 Remote Attestation

Remote attestation uses trusted hardware to prove a system's state to a remote party. The trusted hardware used in this paper is a Trusted Platform Module (TPM). A TPM is a secure coprocessor capable of performing cryptographic operations as specified by the Trusted Computing Group [49]. They have widespread use on enterprise-grade laptops [50] and some consumer devices, such as the Google Chromebook [51]. They are

[1] Although applications distributed by the same developer can choose to share an id.

not currently built into most mobile devices, so, as described in Section 4.5.3, I use a software simulation of one for this prototype. TPMs can generate an asymmetric attestation identity key that uniquely identifies the device. There are two other relevant aspects of the TPM: the Platform Configuration Registers (PCRs) and its *quote* operation.

The TPM has 24 or more PCRs. These are twenty-byte registers which start at some known value and can only modified through an *extend* operation. Given a twenty-byte value v, and a PCR number i, the extend operation uses the SHA256 cryptographic hash function to modify a PCR as follows:

$$\mathrm{PCR}_i \leftarrow \mathsf{SHA256}(\mathrm{PCR}_i \parallel v)$$

where $\parallel$ denotes concatenation. This particular way of modifying PCRs means that for given PCR value, there is a specific chain of extensions that led up to that value. Some of the TPM's PCRs, in particular PCR_{23}, can be reset by software to a known value (0^{20}: twenty zero-bytes) while others can only be reset when the device goes through a physical power cycle.

The other important aspect of the TPM is its `TPM_QUOTE` command [49]. To perform this operation, it signs the values of some subset of PCRs using an attestation identity key: an asymmetric key pair only used for this operation that uniquely identifies the TPM generating the signature. Someone who receives this signature and has access to the corresponding public key can verify that some TPM had the specific PCR values quoted. A cryptographic nonce can be included in the quote in order to ensure freshness.

`TPM_QUOTE` is an important part of the remote attestation protocol. Remote attestation is a way for a remote party to verify the state of a TPM [52]. There are two entities in the protocol: the challenger and the attester. The protocol starts with the challenger providing a nonce. The attester includes this nonce in a call to `TPM_QUOTE`, obtaining a unique signature over the PCR values of the TPM. The attester returns this signature, along with the PCR values, to the challenger. The challenger uses the public component of the attestation identity key to verify the attestation signature. The challenger then confirms that the provided PCR values match the attestation. A resettable PCR can be used to include arbitrary data as part of the attestation, by first resetting that PCR then extending it with the digest of the data that is to be included [53]. This protocol can prove that some data was generated by software with access to a particular TPM.

4.1.3 eBPF

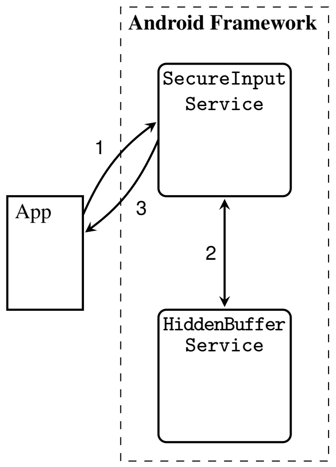
Extended BSD Packet Filters (eBPF) is a restricted language that application developers can use to run flexible computations on sensitive inputs. It is an evolution of the BPF language first introduced for custom in-kernel packet filters in BSD Unix [54]. Since then, eBPF was created for more flexibility. It has an expanded role in the Linux kernel with some seeing it as the "universal in-kernel virtual machine" [55]. Its current uses include packet filtering, system call filtering, and custom kernel tracing [56].

The specification for eBPF is available in the Linux kernel documentation [57]. It is a bytecode for a simple virtual machine. There are ten general purpose registers, ALU operations, conditional branches, and the ability to call external procedures. There are no loops, and all branch targets must be forward, so termination is guaranteed. The input to an eBPF program is an arbitrary byte array. I implemented a basic Java eBPF interpreter[2], which I integrated into the Android Framework and made available to Android applications.

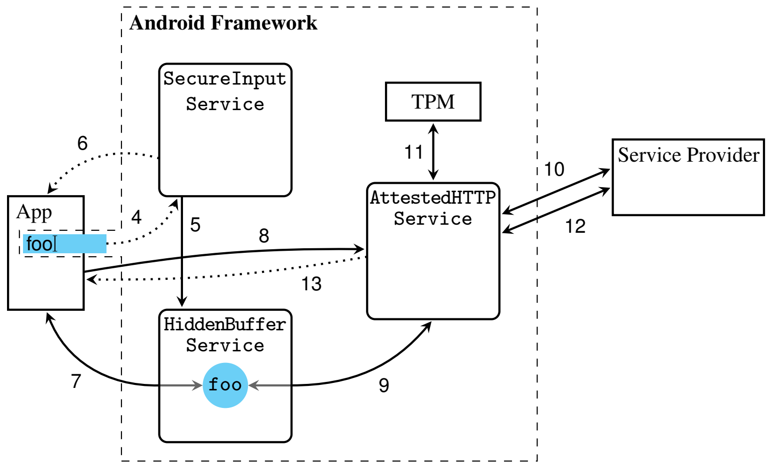
4.2 Design Overview

With some of the necessary background information out of the way, I turn discussion to an overview of the design. The design for Secure Input Overlays builds on the Android platform to provide a way for application developers to increase security for users through strong and attested isolation. For sensitive inputs, applications do not use a standard input element that resides in the application's address space. Instead, the application requests the trusted Android Framework to create a Secure Input Overlay. A Secure Input Overlay includes an input element that lives in a trusted system process and whose contents are inaccessible to the untrusted application. However, the untrusted application is still able to perform limited and logged queries on the contents, which reside in a *hidden buffer*. In order to send the contents of a buffer to another server, the application declaratively describes the request that it wants to happen, and another trusted process performs the request. This request includes an attestation signature generated by trusted hardware. A remote server can verify the attestation, then decide, based on the provided log of queries that were run on the sensitive inputs, whether or not to accept the request. The contents of a hidden buffer cannot be sent to more than one server, and new queries cannot be performed after the contents have been exfiltrated. I implemented this design by modifying the Android Open Source Project, forked from revision 5.1.0_r1. Source code and instructions for

[2] https://github.com/louissobel/jebpf



(a) Phase one—setup. The dashed lines represent the boundaries of the trusted base.



(b) Phases two and three—updating, querying, and exfiltration. The box over the application is a Secure Input Overlay. It has a corresponding hidden buffer in the Hidden Buffer Service. Dotted lines represent asynchronous callbacks.

Figure 2: The three phases of using Secure Input Overlays. Figure (a) shows the setup phase. Figure (b) shows the querying and exfiltration phases. (1) The application asks the Secure Input Service to create an overlay. (2) The Secure Input Service creates the overlay and backing hidden buffer and (3) returns a reference to it. (4) The overlay sends events to the Secure Input Service, which (5) updates the Hidden Buffer Service and (6) forwards some events to the application, which (7) can query the contents. (8) The application can send a description of an HTTP request to the Attested HTTP Service, which (9) assembles a web request, (10, 11, 12) performs the remote attestation protocol, and (13) returns the result to the application.

setting up are available[3].

For this design, the Android operating system and framework are the trusted base. It is also assumed that there is local attestation so that the user is able to identify when they are entering inputs into the trusted base. Ramifications of these assumptions are explored in Section 6.3. The availability of a secure communication channel to a remote server, such as the kind provided by SSL and HTTPS, is assumed as well.

The process of using Secure Input Overlays has three phases, shown in Figure 2. Three new Android Framework services constitute the bulk of the implementation: the Secure Input Service, the Hidden Buffer Service, and the Attested HTTP Service. The design and implementation of these three services is detailed in Sections 4.3, 4.4, and 4.5 respectively. The first phase is setup, the second is updating and querying, and the third is exfiltration. Phase one is shown in Figure 2a. Phases two and three are shown in Figure 2b. In the description that follows, I reference the numbered arrows in the figures.

In the setup phase, the application requests the Secure Input Service to create an overlay (1). The Secure Input Service then has the Hidden Buffer Service create a hidden buffer in which to store the value of the

[3]https://bitbucket.org/secureinputoverlays/manifest

sensitive input (2). Only the Secure Input Service is able to modify the contents of this buffer. The Secure Input Service then creates a new window overlaid on top of the untrusted application's window, and returns to the untrusted application a Binder token identifying the buffer just created (3).

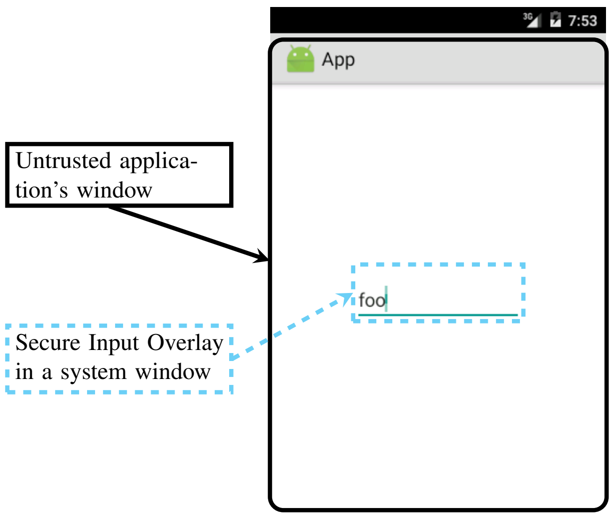
In the second phase, the hidden buffer is updated and the application queries its contents. The Secure Input Service receives callbacks from the overlay when the user starts editing, makes a change, or stops editing (4). When the user updates the contents, the Secure Input Service forwards the change to the Hidden Buffer Service (5), which updates the corresponding hidden buffer. When the user stops or starts editing, the Secure Input Service forwards this event to the application (6). The application can query the contents of the buffer using a limited set of queries (7). It can learn the length of the content, compare it to a regular expression, or run an arbitrary eBPF program. These queries happen against an immutable snapshot of the contents, so that validation code does not have to deal its view of a hidden buffer's contents changing unexpectedly. Every query is logged.

In the third phase, the untrusted application sends the contents of the hidden buffer to a remote server. The application declaratively describes an HTTP request, which includes the target URL and a list of key–value pairs to send in the request. A key is always a string, but a value can either be a reference to a hidden buffer snapshot or a string. The application sends this declarative description to the Attested HTTP Service (8). This service assembles an HTTP POST request, obtaining the contents of any snapshots and the corresponding query logs from the Hidden Buffer Service (9). When it accesses the value of a hidden buffer, it checks that the buffer has not previously been sent to a URL other than the target of the current request. It then performs the remote attestation protocol with the specified server. First, it requests a nonce (10). Second, it gets an attestation of the request body from the TPM (11). Finally, it submits the request (12). Any errors or responses from the remote server are delivered to the application using callbacks (13).

The following five sections first go into further detail on the design and implementation of the three components described: isolated overlays, restricted string queries, and declarative attested HTTP requests. Then I explain the actions that the remote server needs to take upon receiving a request. Finally, I lay out the security properties that the design provides.

4.3 Isolated Overlays

Address space isolated overlays allow embedded user interfaces that completely isolate secrets from untrusted applications. The Secure Input Service is an Android system service that creates and manages these



(a) Annotated screenshot of a Secure Input Overlay in use.

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    >

    <SecureInputView
        android:layout_width="160dp"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
    />

</RelativeLayout>
```

(b) Portion of Android XML layout code creating the Secure Input Overlay in (a).

Figure 3: A Secure Input Overlay in use alongside the Android XML layout code that generated it.

overlays. The Secure Input Service runs in a system process; the overlays reside in the same process. This straightforward isolation model ensures only system processes can access overlay contents. This is possible because the Android window manager is part of the trusted base. Figure 3a shows an application using a Secure Input Overlay. The difference is not perceptible to the user.

To ensure isolation, secrets must never enter the address space of the untrusted application. Another way of doing this might be to intercept input events before the system delivers them to an untrusted process, record the value, and substitute it with a placeholder, as in [19]. The downside to this is that there is no feedback to the user. This would make editing the contents very difficult.

The overlay itself consists of a text input element in an Android system window. There are issues that come up around handling input focus and keeping the window correctly positioned. The window's dimensions are those of the text box it contains. Although only one window can have focus at once, each overlay can detect when there is a click outside of itself and then make itself not able to receive focus so that a window behind it can get focus. An option is set on the window to ensure that its contents will not appear in a screenshot (this option was circumvented to generate Figure 3a and others in this paper).

The position of the window on the screen is determined by the application developer. Their interaction with the service happens through a `SecureInputView` view class which automatically determines proper size and positioning using an invisible element in the untrusted application's view hierarchy that is the same size as the overlay window. The developer can use Android's XML layout description language to create overlays, as shown in Figure 3b. The current prototype is not very robust. For example, it does not automatically adjust its position when the software keyboard appears. These issues are not intractable. LayerCake [58] is a project that also implemented embedded windows within Android (for a different purpose than input security). It solved the keyboard issue and many others and is discussed in further detail in Section 7.

The application developer also uses the wrapping `SecureInputView` to receive events when the user starts or stops editing. From a security standpoint, this is preferable to events for each change because it avoids unintentional and unquantifiable information leakage. The Secure Input Service receives all events from the window, so can forward them to the application as needed. The service updates the hidden buffer whenever the content updates. The application can also use the `SecureInputView` to perform queries on the content of the overlay.

4.4 Restricted String Queries

The Hidden Buffer Service stores the actual sensitive inputs that users enter and allows applications to query the values. These queries are logged so that a list of them can be provided to remote servers. The service maintains a set of buffers that only their creator can update. Applications can access the contents through a limited query interface. Other system services can access the contents of a hidden buffer, obtain its query log, and modify its exfiltration state.

4.4.1 Hidden Buffers

A hidden buffer holds the value of a sensitive input, hidden from untrusted applications. A hidden buffer consists of the actual string contents of a Secure Input Overlay, a log of all queries that have been run on its contents, and the content's exfiltration state, used to ensure that the contents of a Secure Input Overlay can only be sent to one remote server and that new queries cannot take place after the contents have been exfiltrated.

When a hidden buffer is created, the Hidden Buffer Service creates two universally unique Binder tokens associated with it. Only a system service can create buffers. Both tokens are returned to the creator. The tokens are an *update token* and a *query token*. Either can be used by the Hidden Buffer Service to find a particular buffer. The contents of a hidden buffer can only be modified using the update token. This is what the Secure Input Service uses to propagate edits in the overlay back to the Hidden Buffer Service. The Secure Input Service passes the query token to applications and keeps the update token a secret. The query token can be used to create snapshots.

A snapshot of a hidden buffer is an immutable copy of its contents at a particular point in time. They are created by passing a query token to the Hidden Buffer Service, which then looks up the corresponding buffer, creates a snapshot, associates that snapshot with a new Binder *snapshot token*, and returns that new token to the caller. Applications can run queries against snapshots and can declaratively request that the Attested HTTP Service send them to a remote server. The immutability of snapshots allows for a programming model where a particular input cannot change between validation checks or in the time between validation and when the data is accessed by the Attested HTTP Service. The developer-facing API to snapshots is through a `HiddenContentHandle` which is created by calling a method of a `SecureInputView`. Although there can be multiple snapshots belonging to a particular buffer, each buffer has only one query log and exfiltration state.

The role of the exfiltration state is to ensure that a remote service learns everything that ever happened to a buffer. There are two parts to this. First, ensuring that a buffer's contents can only ever be sent to one URL. Second, ensuring that no new queries can occur after a buffer's contents may have been exfiltrated. The exfiltration state consists of three fields: a string `exfiltrationTarget`, a Boolean `exfiltrationInProgress`, and a Boolean `maybeExfiltrated`, which are initially `null`, `false`, and `false`, respectively. The exfiltration state has to deal with uncertainty in whether or not the value has actually been exfiltrated due to the possibility of network errors, hence the "maybe". There are two methods that modify the exfiltration state.

The first is `testAndSetExfiltrationTarget`, which takes a snapshot token and string URL as arguments. It looks up the buffer corresponding to the provided snapshot token and atomically checks the buffer's `exfiltrationTarget`. If it is currently `null`, it is set to the provided URL and the method returns `true`. If if it not `null`, the method returns `true` if it matches the provided URL and `false` otherwise. If this method returns `true`, `exfiltrationInProgress` is set to `true`.

The other method is `exfiltrationAttemptCompleted`, which takes a snapshot token and a Boolean `possiblySuccessful` as arguments. It must only be called after marking an exfiltration in progress using `testAndSetExfiltrationTarget`. The Boolean argument `possiblySuccessful` should be `true` if the exfiltration was possibly successful and `false` otherwise. The Hidden Buffer Service looks up the buffer corresponding to the provided snapshot token and atomically does two things. First, it sets `exfiltrationInProgress` to `false`. Second, it sets:

```
maybeExfiltrated = (maybeExfiltrated or possiblySuccessful)
```

This ensures that `maybeExfiltrated` cannot transition from `true` to `false`. New queries are not allowed if either `maybeExfiltrated` or `exfiltrationInProgress` is `true`. The Attested HTTP Service uses this state machine to ensure that if there is some possibility that the contents of a hidden buffer have been or will be exfiltrated, then new queries are not allowed.

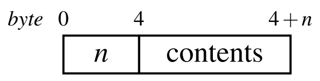
4.4.2 Logged Queries

An important role of the Hidden Buffer Service is to allow untrusted applications to query the contents of a snapshot. This allows applications to perform client-side validation of sensitive inputs. There are three factors to take into account in choosing what query interface to expose. A primary one is serializability: it must be possible to serialize the query in order to log it in a way that can be transmitted across the network and to pass it as a Binder IPC argument. The second is simplicity: common queries should be simple and easy for applications to perform. The final one is expressiveness: uncommon and complicated queries should be possible. Because the Hidden Buffer Service has unfettered access to the actual contents of a sensitive input, protection and computation are separate, so any query interface is possible. It is acceptable for queries to leak the contents of a buffer as long as the service provider can learn that there was a leak. The query log ensures this. The current design exposes three types of queries: length, regular expression, and eBPF. Application developers use these queries by calling methods of a `HiddenContentHandle`.

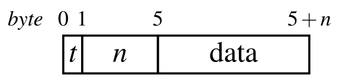
Length Length queries return the length of the contents of the snapshot. This is useful in the case when the length is the only information an application needs. Regular expressions cannot determine the length (in one query).

Regular Expression Regular expression queries allow an application to validate the contents of a snapshot against a regular expression. The Hidden Buffer Service returns a Boolean value: whether or not the contents match the supplied pattern. The use of flags is supported, but some other useful features of regular expressions, such as captures, are not. Regular expressions can answer many questions about inputs for validation purposes, such as whether an email looks valid, a date is formatted properly, or that a password meets some policy-imposed constraints.

eBPF When a finite automaton is not enough, eBPF queries provide the needed expressiveness. They allow running an eBPF program with the snapshot contents as input. The "input packet" to the eBPF program is a byte array laid out as shown in Figure 4a. The string is encoded to bytes using `UTF-8`. The Java



(a) Layout of input to eBPF program for contents of length *n*.



(b) Layout of serialized query log entry for query of type *t*, whose data has length *n*.

Figure 4: Byte-level serialization formats used by the Hidden Buffer Service.

eBPF interpreter previously mentioned deserializes and runs the given program on the constructed packet. eBPF is is a good manifestation for the arbitrary computation that the query interface could expose. It is guaranteed to terminate and is easy to serialize.

Both regular expression and eBPF queries enable the application to eventually learn the contents of a hidden buffer. The query log ensures that a remote server can learn about all queries that were performed on a buffer, so it can reject inputs that had over-informative queries run against it. The query log records the unique queries run against each buffer. This log can be serialized. Figure 4b shows the byte layout of a serialized log entry. Each type of query has an associated one-byte-integer type t as well as optional data d. Length queries have no associated data. The data for regular expressions is the `UTF-8` encoded string that made up the pattern. The data for eBPF queries is the serialized list of instructions. These are the same serializations used to pass queries as IPC arguments. There is one log per buffer, rather than per snapshot, because the remote service provider needs to know all queries that were performed against all values of a hidden buffer, not just the queries performed against one particular value. If the hidden buffer to which a snapshot belongs has already possibly been successfully exfiltrated, or if there is an exfiltration in progress, no queries are permitted that are not already in the log. System services can access the contents of a snapshot and the query log of its associated hidden buffer by passing in a snapshot token. This is how the Attested HTTP Service builds a request containing sensitive inputs.

4.5 Declarative Attested HTTP

Once the user has entered sensitive input into a Secure Input Overlay and the untrusted application has performed any needed validation, the final step is for the application to send the data to the trusted service provider. The Attested HTTP Service performs this task. The application provides a declarative description of an HTTP POST request. The service performs this request, communicating back to the application through Binder IPC callbacks.

The declarative request description includes a destination URL and parameters. The parameters are a set of key–value pairs. The keys are strings, but the values can either be strings or snapshot tokens which refer to a hidden buffer. This enables any combination of strings and sensitive inputs to be sent in a single request. The developer interface to the Attested HTTP Service is through an `AttestedPost` request object.

After receiving the declarative description of the web request, the Attested HTTP Service first asserts that the calling application has the Android `INTERNET` permission, which it must in order to able to access the network. If it does, the service then performs a remote attestation protocol to send an attested HTTP POST request to the provided URL. There are four steps to this:

1. Obtaining the nonce from the destination server,
2. Constructing the request body,
3. Getting a attestation from the TPM, and
4. Sending the request, including the attestation, to the specified server.

Throughout this process, the service carefully interacts with the exfiltration state of any of the hidden buffers whose content it is sending to ensure that the security properties are maintained.

4.5.1 Obtaining a Nonce

The first step of the remote attestation protocol is obtaining a nonce from the destination server. This needs to be performed as a separate HTTP request. The service makes a GET request to the provided URL, with "`/nonce`" appended to the request path. If the response code is not `200`, it is interpreted as an error. Otherwise, the raw bytes of the response body are interpreted as the nonce.

4.5.2 Constructing the Request Body

After getting the nonce, the next step is to build the request body. The parameter values are gathered and then encoded. Parameter values can be either strings or snapshot tokens which are references to a hidden buffer. The Attested HTTP Service accesses the Hidden Buffer Service to resolve any snapshot token parameters by getting the actual string contents to which they refer. For each snapshot token parameter value, it uses the `testAndSetExfiltrationTarget` method of the Hidden Buffer Service to check whether or not that hidden buffer can be sent to the requested URL. If successful, this has the side effect of preventing any new queries from occurring at least until the service calls `exfiltrationAttemptCompleted`. The service then obtains the contents of the snapshots and the query logs of the corresponding hidden buffers.

It encodes the parameters using a form-encoded HTTP Multipart request [59]. This was chosen because, as part of the HTTP specification, many HTTP servers will be able to parse it without modification. The choice of encoding has no fundamental bearing on security properties, so anything could be chosen. For each parameter whose value is a string, it adds the parameter name and value pair. For parameters whose value is a snapshot token, it adds the parameter name with the value obtained from the Hidden Buffer Service. The Attested HTTP Service then adds the serialized query log for that parameter with a name obtained by appending "`-query-log`" to the original name. To ensure that applications cannot send fake query logs, no application-provided parameters are permitted whose name ends with "`-query-log`". Finally, the nonce is added with parameter name `nonce`, and the destination URL is added with parameter name `exfiltration-url`.

4.5.3 Getting the Attestation

After preparing the request body, the Attested HTTP Service uses trusted hardware to get an attestation that verifies it. As described above, most mobile devices do not currently include a TPM. A software simulation of one is used instead. The TPM simulation is encapsulated behind a fourth new system service, the Signing Service. The Signing Service exposes one method.

This method is only accessible to system services. It accepts a nonce and a byte array containing data to sign and returns an attestation signature in the same format that a TPM would. It computes the $\mathsf{SHA256}$ of the input data then simulates resetting PCR_{23} and extending it with that digest:

$$\text{PCR}_{23} \leftarrow \mathsf{SHA256}(0^{20} \parallel \mathsf{SHA256}(\text{data}))$$

where 0^{20} denotes twenty zero-bytes. The Signing Service then simulates a `TPM_QUOTE` operation. It assembles a `TPM_QUOTE_INFO` data structure as specified in [49]. This data structure includes the value of PCR_{23} and the nonce value. The Signing Service signs this structure using the private component of a hard-coded attestation identity key and returns the result. The Attested HTTP Service uses this method of the Signing Service and the nonce obtained in Step 1 to obtain an attestation signature of the request body.

4.5.4 Sending the Request

The final step the Attested HTTP Service needs to perform is actually sending the request. It starts assembling it using the body constructed in Step 2. The attestation signature obtained in Step 3 is added as the header `X-Attestation-Signature`. The request is then sent via an HTTP POST to the URL that the untrusted application provided.

Based on what happens next, the exfiltration state of any hidden buffers whose content was included in the request needs to be updated. Prior to actually sending the POST request, any errors that the Attested HTTP Service encounters are handled by calling `exfiltrationAttemptCompleted` with the parameter `possiblySuccessful` having the value `false`. If the sensitive inputs never leave the device, there is no way they could have reached the remote service provider. However, once the secure inputs leave the device, the exact opposite is true. The default, for either a transport-level error (`IOException`) or application-level error (non-200 response code), must be to assume that the server did receive the inputs, and `possiblySuccessful` should always be `true`. This is because of the unreliable network. No matter what the response code from the POST request, the referenced hidden buffers must be marked as possibly exfiltrated. While no new queries will be possible after this point, the application can repeat already used queries and resend the buffer to the same URL.

4.6 Server-Side Validation

The final step of the design is what the server does upon receiving a POST request sent by the Attested HTTP Service. It needs to verify the attestation then evaluate the request contents to make a trust decision.

The first part is verifying the request's attestation signature. First, the server should confirm that the provided nonce is valid: it was in fact issued by the server and has never before been used. This prevents replay attacks. Second, the server needs to verify the signature provided in the `X-Attestation-Signature` header. To do this, it assembles a structure similar to what the TPM would sign by computing the SHA256 digest of the request body, simulating extending a blank PCR with that digest, and constructing the correct `TPM_QUOTE_INFO` structure. It then uses the public component of the TPM's attestation identity key to verify the signature.

If the request is valid, then based on the attestation identity key, the server knows that it came from an Android device. The server can assume that it came from an authentic and trusted Android operating system. Section 6.3 defends this assumption. The server then needs to check that the URL specified in the request's `exfiltration-url` is that of the server itself. This ensures that the request was sent directly to the service provider. The server then evaluates the query logs supplied for each input coming from a Secure Input Overlay. The contents of a query log can be checked against a whitelist of allowed queries. If every query was allowed, the server knows that the sensitive inputs were handled according to its specification, and that the untrusted application has only the access to them that the service provider allows and knows about.

4.7 Security

The security property that Secure Input Overlays provide is that the request and attestation that the remote server receives mean that the supplied inputs were entered into a Secure Input Overlay and that all computations the untrusted application ran against them are included in the query logs. The server can use this information to be sure of the level of isolation the sensitive inputs received and make its trust decisions accordingly. Under the threat model given this holds.

The first part of that property is that the server knows that the request it receives was generated by the Attested HTTP Service. The Signing Service uses Binder's trusted IPC attribution to ensure that only system services can use the TPM, so the signature in the request must have been generated by the Attested HTTP Service. The nonce prevents replay attacks.

The second part of the security property is that any input that includes a query log was actually entered into a Secure Input Overlay. The Attested HTTP Service ensures that a user cannot use the suffix "`-query-log`" for any parameter names. So a parameter with such a name will only be included if it actually is coming from a snapshot. The Hidden Buffer Service limits creation of hidden buffers, also by using Binder's attributed IPC, to system services. The Secure Input Service is the only service that creates hidden buffers and does not reveal the update token that the Hidden Buffer Service gives it, so it is the only thing that can update a hidden buffer. It only does that when a user types into a Secure Input Overlay.

The final part of the security property is that the query log is enough information to know exactly how a sensitive input was handled. The address space isolation between the untrusted application and the Android system services ensures that an application cannot access the contents of a Secure Input Overlay without going through the query interface. All queries are logged. The Hidden Buffer Service only allows system services access to the contents of a hidden buffer. Screenshots are prevented. The exfiltration state that the Hidden Buffer Service stores ensures that no queries can happen after the Attested HTTP Service obtains the copy of the query log that it sends to the remote server and that each sensitive input is sent to exactly one URL—the one listed in the request—which matches the server's URL. All this ensures that the received inputs were entered into a Secure Input Overlay, did not leak, and were sent directly to the service provider.

5 Case Studies

This section describes the implementation of two example applications that use this system: an attested login application[4] and a payments application[5]. The point of these case studies is to show the feasibility of using Secure Input Overlays instead of regular text inputs, to demonstrate the range of possible client-side validation, and to exercise the exposed APIs.

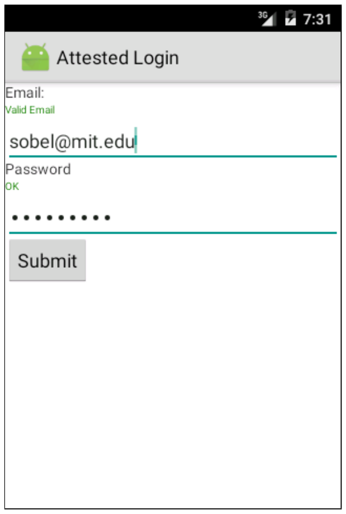
5.1 Authentication

The first case study is an attested login form. The user can submit an email address and password without the untrusted application being able to see either. This is proven to the remote authentication provider. This removes the need for OAuth, as authentication providers would accept the user directly entering their credentials if they knew that the application did not access them beyond whitelisted validation queries. Figure 5a is a screenshot of the application. Both of the input elements are Secure Input Overlays.

Rich validation is still possible. A regular expression is used to validate the email address. A combination of regular expressions and length checks are used to enforce an asinine password policy: that the length is 6–16 characters and that there is at least one number, at least one lowercase letter, at least one uppercase letter, and at least one "special character". Figure 5b shows the code used for validation, which uses the `HiddenContentHandle` API. Validation code for strings would not be any simpler.

On submission, the application uses the Attested HTTP Service to send both the email address and password to a remote server. The server verifies the request and checks that the performed queries are whitelisted. The server then checks that the username and password are valid. If they are not, it can inform the application. The `exfiltrationState` of a hidden buffer is set up so that the application can repeat the validation queries if a user needs to correct their password or email address and resend the request to the same URL—an important feature for login forms.

[4] https://bitbucket.org/secureinputoverlays/android_packages_apps_attestedlogin
[5] https://bitbucket.org/secureinputoverlays/android_packages_apps_paymentform


Attested Login

Email:
Valid Email
sobel@mit.edu
Password
OK
•••••••••

(a) Screenshot of attested login example application. It shows support for multiple overlapping Secure Input Overlays and a variety of input types, including a password.

```
Pattern emailRegex = Pattern.compile(
    // Regular Expression from [60]
    "^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}$",
    Pattern.CASE_INSENSITIVE
);

boolean validateEmail(HiddenContentHandle email) {
    return email.matchesPattern(emailRegex);
}

Pattern hasUpper = Pattern.compile("^.*?[A-Z].*$");
Pattern hasLower = Pattern.compile("^.*?[a-z].*$");
Pattern hasNumber = Pattern.compile("^.*?[0-9].*$");
Pattern hasSpecial = Pattern.compile("^.*?[!^$&%*#@].*$");

boolean validatePassword(HiddenContentHandle password) {
    int length = password.getLength();
    if (length < 6 || length > 16) {
        return false;
    }
    boolean u = password.matchesPattern(hasUpper);
    boolean l = password.matchesPattern(hasLower);
    boolean n = password.matchesPattern(hasNumber);
    boolean s = password.matchesPattern(hasSpecial);

    return (u && l && n && s);
}
```

(b) Section of code from the attested login example application. It demonstrates how a `HiddenContentHandle` can be used to validate the contents of a Secure Input Overlay.

Figure 5: Artifacts from the attested login case study.

5.2 Credit Cards

The second case study is a payment form that is able to validate entered credit card information without seeing any of the sensitive inputs. Figure 6a shows a screenshot of the payments application. There are four fields: `name`, `number`, `expiration`, and `cvv` (a card verification value used to assert physical card presence). The `name` field is not a sensitive input, so uses an ordinary Android `EditText` view. The remaining three inputs *are* sensitive, so each uses a Secure Input Overlay created using a `SecureInputView`. The user can submit their payment information to a remote server, which receives the content and an attested list of the computations run on each of the three sensitive fields.

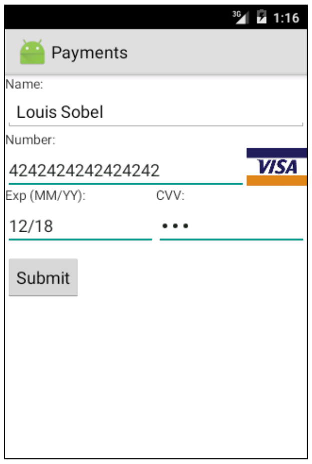
There are various validations that can occur against these sensitive inputs. The validations improve the user experience. Figure 6b shows a screenshot of the payments application with some of these validations failing. First, the brand of the credit card can be determined using the first few digits. In Figure 6a, the card is identified as a Visa due to the prefix `42`. The allowed length of a credit card number varies based on brand, but ranges from 15–19. The prefix of the hidden credit card number can be checked with a regular expression and the length with a length query.

A second layer of validation of the number is the Luhn check [61], a checksum over the digits that can detect transposed and mis-entered digits. The Luhn check can be implemented in eBPF. Figure 6b shows an invalid American Express card number that fails the Luhn check. The Luhn check algorithm needs to iterate over all the digits. eBPF does not have support for loops, so the code needs to use an *unrolled* loop, which is not straightforward to implement by hand. One option would be to dynamically generate the eBPF code, including an unrolled loop of the correct size, based on the length of the credit card number. This would require the server to whitelist many different eBPF programs as allowable computations. Another option, which I use in this example, is to use a pre-generated eBPF program that has an unrolled loop with a fixed maximum number of digits and skips loops iterations if all digits have been consumed. This is possible because—as explained in Section 4.4.2—the length of the string is part of the input to the eBPF program. Having only one allowable eBPF query makes it easier for the server to assess a list of queries, but it causes the eBPF program itself to have an awkward implementation.

The payments application also performs validation of the expiration date and CVV. The expiration date is validated against a regular expression to make sure that it matches the desired format. The expiration date in Figure 6b does not. A useful validation that this application cannot perform is checking that the expiration date is in the future. It would be possible to use dynamically generated eBPF to check for this, but this would cause whitelist complexities similar to those described above for the Luhn check. Alternatively, the Hidden Buffer Service eBPF interpreter could expose a function for getting the current date. The CVV is the final hidden buffer that is validated. The payments application checks that it matches a length that depends on the type of the card. American Express cards have a CVV that is four digits, so the three digit one in Figure 6b causes an error.

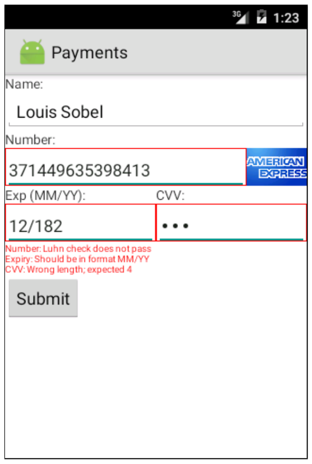
If the entered credit card is valid, the user can send their information to a remote server. Figure 6c shows the code that the application uses to do this. It uses the `AttestedPost` API to construct a declarative description of a request that includes all four fields. It can add parameters to this request using the same method, `addParam`, regardless of whether the parameter is a `String` or a `HiddenContentHandle`. When it performs the request, it specifies two callbacks, exactly one of which will be called. If there is an error in building the request or running the remote attestation protocol, the `ErrorListener` will be called. If the request succeeds, then the `ResponseListener` will be called.

This case study and the attested login one demonstrate that it is possible to build useful applications



Name:
Louis Sobel
Number:
4242424242424242
Exp (MM/YY):
12/18
CVV:
•••

(a) Screenshot of payment form with valid information. The type of credit card can be detected using a regular expression that looks at the first few digits.



Name:
Louis Sobel
Number:
371449635398413
Exp (MM/YY):
12/182
CVV:
•••

Number: Luhn check does not pass
Expiry: Should be in format MM/YY
CVV: Wrong length; expected 4

(b) Payment form with errors. The Luhn check fails for this incorrect American Express card number. A regular expression detects that the expiration date has the wrong format. CVVs for American Express cards are expected to be four digits, not three (as entered).

```
void doSubmit() {
    String name = mNameInput.getText().toString();
    HiddenContentHandle number = mNumberInput.getHiddenContentHandle();
    HiddenContentHandle expiry = mExpiryInput.getHiddenContentHandle();
    HiddenContentHandle cvv = mCvvInput.getHiddenContentHandle();
    CreditCard c = new CreditCard(name, number, expiry, cvv);

    if (!c.isValid()) {
        handleSubmitInvalidCard(c);
        return;
    }

    AttestedPost req = new AttestedPost.Builder()
        .url(PAYMENT_URL)
        .addParam("name", name)
        .addParam("number", number)
        .addParam("expiry", expiry)
        .addParam("cvv", cvv)
        .build();

    req.perform(new AttestedPost.ResponseListener() {
        public void onResponse(AttestedPostResponse r) {
            handleSubmitResponse(r);
        }
    }, new AttestedPost.ErrorListener() {
        public void onError(String msg) {
            handleSubmitError(msg);
        }
    });
}
```

(c) Code from the payments application that uses the Attested HTTP Service to send a POST request. `Strings` and `HiddenContentHandles` can both be easily added to the request. The application uses callbacks to handle a successful response or any errors.

Figure 6: Artifacts from the payment form case study

that can send sensitive inputs to a remote server using Secure Input Overlays. The overlays provide strong isolation: the sensitive inputs are never accessed directly and all queries are logged. The queries enable client-side validation and a user experience that is comparable to existing applications.

6 Evaluation

This section evaluates the design and implementation of Secure Input Overlays. First, I evaluate the performance, looking at the impact of the Hidden Buffer Service IPC on query latency and the effect of the remote attestation protocol on usability. Then, I qualitatively discuss the developer effort required. Last, I analyze the usefulness of the attestation that the remote server receives by reexamining some of the threat model assumptions.

6.1 Performance

I evaluate performance from two angles. First, the overhead introduced by applications needing to go through the Hidden Buffer Service to query the contents of user input. Second, the effects that the use of the TPM and the remote attestation protocol have on the performance and usability of making a POST request.

6.1.1 Query Overhead

Instead of just running methods on objects in its address space, the untrusted application is required to make a Binder IPC call to the Hidden Buffer Service to learn about sensitive inputs. This introduces overhead. Regular expression and length queries take just as long inside the remote Hidden Buffer Service as they do in a local process, so any overhead is only from the IPC call itself. A regular expression query has a slightly larger IPC overhead than a length query does because the regular expression needs to be serialized and deserialized. In a length query, there is no associated data, so there is no serialization overhead. The overhead of eBPF queries is variable. Like that of regular expression queries, the eBPF overhead includes serialization penalties. But the total overhead depends on the actual computation happening—the relative performance of native Java code and the same procedure implemented as interpreted eBPF.

To quantify the query overhead, I performed three benchmarks, one for each query type. Results are shown in Table 2. The results show that while the relative overhead of querying using the Hidden Buffer Service is quite large, the absolute value is still small. Given the relative infrequency at which these queries

	Local		**Remote**	
Query	*Average*	*SD*	*Average*	*SD*
Length	52µs	8µs	669µs	674µs
Regex	74µs	14µs	626µs	345µs
eBPF	70µs	10µs	2,912µs	2,723µs

Table 2: Results of benchmarks showing the overhead of the Hidden Buffer Service on queries. The same query was run both in a local process and remotely using an IPC call to the Hidden Buffer Service. Although the overhead is large, the absolute latency is still reasonable. The latency using IPC calls is far more variable than the same query in a local process.

need to run, the overhead ultimately is acceptable. The benchmarks were performed on an Android Emulator with HAX enabled on Mac OS X. The code for the Android application used to perform the benchmarks is available[6].

The benchmarks were measured by performing 1000 local method calls on the string contents of an Android `TextView` and 1000 iterations using the corresponding call on a hidden buffer populated with the same string. The regular expression benchmark used the email validation regular expression from Figure 5b. The eBPF benchmark used the Luhn checking code from the payments application case study. The data shows that using the Hidden Buffer Service introduces an order of magnitude of overhead and a large amount of variability. However, for length and regular expression queries, the mean latency is still less than a millisecond. The interpreted eBPF query has an even higher overhead. A intraprocess Luhn check implemented as native Java code is able to far outperform an interpreted eBPF version of the same algorithm. However, even with the complex eBPF program used, the mean latency is comfortably under ten milliseconds. Validation queries need to happen relatively infrequently, on the order of every 100 milliseconds to seem instantaneous [62]. The query overheads introduced are small relative to this duration, so would not have a noticeable effect on user experience.

6.1.2 Remote Attestation

A more significant source of overhead comes from the remote attestation protocol being used in place of a single unattested POST request. The overhead of remote attestation has two sources. The first is that the protocol requires an additional HTTP request in order to fetch the nonce. This potentially more than doubles the latency of the process.

The other source of latency is the `TPM_QUOTE` operation. Parno et al., in [63], benchmarked various TPMs

[6]`https://bitbucket.org/secureinputoverlays/android_packages_apps_querybenchmark`

and found that `TPM_QUOTE` latency varies, by manufacturer, from around 350ms to over 900ms. Even using the TPM with the best quote performance, this is significant latency to tack on.

However, given the particular use case, the combined overhead of the extra web request and the TPM, though high, is tolerable. Submitting sensitive inputs is not an common operation. Logins, for example, will happen at most once for each use of an application. Payment is also a relatively infrequent operation. There is also already the expectation of some latency during these actions. Although using Secure Input Overlays might double or triple this delay, it would not cause applications to become unusable or severely impact user experience.

These overheads are not specific to Secure Input Overlays; any similar remote attestation protocol must deal with them. The longer round trip latency further reinforces the need to perform rich client-side validation. This minimizes instances of requests being rejected for errors the client could have detected.

6.2 Developer Effort

This section qualitatively discusses the developer effort required to build applications that use Secure Input Overlays. The changes I propose are not backwards compatible; they would not suddenly make existing applications more secure. But at the same time, nothing breaks; existing applications would neither suddenly become less secure nor stop functioning. Lack of backwards compatibility is acceptable. Server and operating system changes would have to happen, so the timeline is such that backwards compatibility is not an immediate need. However, because the design requires changes to applications, ease of integration is an important consideration. There are four aspects to look at for developer effort: creating a Secure Input Overlay, interacting with it, making an HTTP request, and modifying the server.

Creating Creating a Secure Input Overlay is, from a developer's perspective, no different than creating an ordinary input element. Figure 3b showed how a `SecureInputView` could be created as part of the XML description of an Android layout. Because Secure Input Overlays are designed to act just like a regular input element, the application developer does not have to work around them, but rather can just use them where a regular input would go. There are issues with how the current prototype overlays interact with the rest of an application's interface, but the API would not change as these issues were resolved.

Querying The developer-facing concept of a `HiddenContentHandle` simplifies interacting with the contents of hidden buffers. The fact that a `HiddenContentHandle` reflects an immutable snapshot makes handling them just like handling a string. One difference is that unlike with a standard `TextView`, the only times an application can receive events from a `SecureInputView` are when the user starts editing and when the user stops editing. Only receiving these particular events is an issue if the application needs to do something every time the user changes the text. However, especially given the low query latency numbers previous reported, if the perception of real time updates are necessary, the application developer can repeatedly poll.

The length and regular expression queries present similar APIs to those that a string does. Anecdotally, writing eBPF queries is trickier. The lack of loops in eBPF forces the developer to perform contortions and use unrolled loops to adapt a single program to multiple string lengths. I experienced this while writing the eBPF Luhn check. The LLVM C compiler has an eBPF backend [64] which could potentially simplify this process by automatically unrolling loops.

HTTP Request The API used to perform an attested HTTP request was designed to match existing Android HTTP libraries. Figure 6c shows an example of how a request is constructed. The use of a `Builder` matches the OkHttp API [65], used internally in Android. Parameters can be added using an overloaded method `addParam` that works whether the value is a string or a `HiddenContentHandle`. One benefit of being able to use multiple parameters per request is that there is still just one external request from the developer's point of view. The request callback interface was designed to be similar to the one used by Volley, an Android HTTP library [66]. The API for the Attested HTTP Service is limited. It is not a generic HTTP client, which makes it a more focused API and hence a simpler one to design.

Server Modifications Developers would also have to modify application servers to support the remote attestation protocol. An architectural change that would be needed would be the addition of support for nonces. Supporting nonces can be difficult, especially in a setting where a load balancer is in use and subsequent requests might end up at different servers. This would mean that the nonce would have to be stored in some global, shared state, increasing the coupling of an application's infrastructure. However, most load balancers have some option for stickiness, where subsequent requests can be routed to the same server. If this feature were used, nonce generation could be a per-server task. Another issue for the server is managing the public key infrastructure for checking attestation identity keys. This project completely sidesteps this issue by having one hardcoded attestation identity key. In general, key infrastructure for TPM attestation is an open question.

6.3 Usefulness of the Received Attestation

The attestation that the server receives and verifies does not provide complete assurance that the received inputs passed through a trusted path. The security properties described in Section 4.7 relied on two assumptions. First, that some local attestation scheme existed. Second, that a trusted Android operating system generated the received request.

Because Secure Input Overlays do not provide local attestation, the service provider cannot be entirely sure that the inputs they receive were not misused. Consider the following scenario. A malicious application performs a *phishing attack* by masquerading as another application's payment form. The user mistakenly enters their credit card information, which the malicious application exfiltrates to an attacker. The malicious application then displays a notice that an error occurred and forwards the user to the actual application, where the user proceeds with payment, unaware that their information was stolen. The attacker then uses the stolen payment information on another mobile device using the actual application. The service provider would receive two requests, one from the user and the other from the attacker using the stolen credentials. The query logs in neither of these requests would contain enough information to detect the theft.

The proper solution to this is ensuring local attestation. But even without it, the attestation may still be useful. An attestation identity key identifies a device. If the attacker were doing this often, the service provider would notice a high number of attested requests, with different inputs, coming from the same device. The attacker would have to have a different Android device for almost every set of stolen sensitive inputs in order to match the usage patterns of non-malicious users. That would be very expensive. Service providers already have monitoring around authentication and payments. The source of the attestation would be a useful signal for aggregate detection of malicious activity. Although this inherently invites concerns about anonymity and the tracking of devices using attestation identity keys.

The second issue is that the Android operating system is trusted to provide critical components of the security guarantee, but users are able to install custom operating systems on their devices. Cyanogenmod [67] is one that is widely used. The modified operating system would still be able to access the TPM and so could look like an authentic Android operating system to the service provider. One well-researched solution would be to use the TPM to record a full chain of static trust measurements, starting from the bootloader, to ensure that the operating system is trusted. However, because protecting sensitive inputs is a use of remote attestation where the user and the remote party have aligned interests, this would not be necessary. If the modified operating system that the user installs does not properly handle secure inputs but still claims to do so, the user is willfully giving up the security guarantees. This is like the Same-Origin Policy and other browser-enforced security conventions on the web. The user is welcome to use a browser that is not as secure, but they do so at their own risk. This is in contrast to other uses of remote attestation where the user's interests are not necessarily aligned with those of the remote party, such as Digital Rights Management [68] or requiring a certain device configuration in order to access a network [69]. In these cases, a full static chain of trust is usually necessary.

Even though the received attestation is not ironclad, it is still a good source of security that service providers could use to make advanced trust decisions. Integration with local attestation would make the trusted path fully end-to-end.

7 Related Work

This section discusses related work, elaborating on selected projects already mentioned in Section 3 and throughout the paper. LayerCake [58] also uses embedded windows to achieve Android security properties. Bumpy [19] provides an end-to-end trusted path and attested login on desktops. VeriUI [7] provides attested login on Android devices without trusting the operating system. GuarDroid [4] is a project that also uses a shadowing technique for data protection and enables untrusted applications to send HTTP requests containing unseen sensitive inputs.

The LayerCake project also uses the ideas of overlays and windows within windows. LayerCake does this in order to allow applications to securely embed other applications. This is useful, for example, to isolate advertisements from the applications embedding them in order to prevent fraudulent clicks. The authors of LayerCake also modified the Android Open Source Project to build a prototype. Their implementation of a view class that wraps an embedded window as the developer facing API was a guide for my design. They dealt with some of the issues around brittleness of embedded windows that I ignore in this prototype. A hybrid project might use LayerCake's embedded window infrastructure to support Secure Input Overlays.

Bumpy has a lot in common with Secure Input Overlays. Its goal is a trusted path for inputs on desktop computers. A secure attestation sequence and dedicated interface element (a trusted mobile device) are used for local attestation. For data protection, trusted hardware and the Flicker [70] project are used to process keyboard inputs securely within a trusted execution environment. Bumpy has the concept of *post-processors*, which are computations that can be run on inputs. In Bumpy, these are arbitrary machine instructions. They play a role similar to the restricted queries that the Hidden Buffer Service provides. Bumpy provides remote servers with an attestation that the inputs it is sending passed through its system and a list of the post-processors that ran on them. The remote server then makes trust decisions, based on this list of post-processors, that are similar to the trust decisions servers can make using the query log maintained by the Hidden Buffer Service.

Another project that can provide attested login is VeriUI. VeriUI uses ARM TrustZone to bring up a secure and trusted operating system where users enter credentials and specify to which URL those credentials should be sent. When this trusted operating system transmits the credentials, it includes an attestation, signed with an attestation identity key, that they were handled within the VeriUI system. This is similar to the attestation that Bumpy provides and that the Attested HTTP Service does in my design. The first case study, from Section 5.1, replicates parts of the functionality of VeriUI's attested login application.

GuarDroid is another Android project with similarities to Secure Input Overlays. GuarDroid's use of personalization and shadowing to protect password inputs was previously discussed. GuarDroid took a different approach to what they actually give the untrusted application and how they send the sensitive inputs in web requests. They give the application an encrypted blob that shadows a password. This precludes any kind of validation and limits possible feedback to the user. An advantage of this is that it is backwards compatible. A place where their goal of backwards compatibility weakens their design is how they support applications sending web requests that use the unencrypted versions of passwords. They run a web proxy that captures all HTTP traffic. It searches for and replaces instances of their encrypted blobs with the corresponding plaintext password. This adds significant overhead. It also reduces security—they perform a man-in-the-middle attack on SSL certificates in order to be able to examine the contents of HTTPS connections.

8 Future Work

This section discusses integrating a local attestation solution and other possible security extensions for Secure Input Overlays.

8.1 Integration With Local Attestation

As mentioned, this design focuses on protecting inputs from applications, and does that, but only once they have been securely entered. It ignores local attestation. Without local attestation, users cannot be sure that they are typing into a Secure Input Overlay. As discussed in Section 3, there have been many approaches for providing local attestation on mobile devices. Here, I suggest how Secure Input Overlays might be integrated with some of those solutions to provide an end-to-end trusted path.

8.1.1 Secure Attestation Sequence

A secure attestation sequence could be used for local attestation. One design that would be effective would be something like SpoofKiller [20], where users become trained to always click the power button before entering a password. The parallel for this project would be that if a user is typing into a Secure Input Overlay and presses a hardware secure attestation key, they could become convinced they are interacting with the trusted base as long as the screen does not go blank.

8.1.2 Dedicated Interface Element

A dedicated interface element could also provide local attestation. A straightforward way to do this would be to have a visible, external indicator. One option would be an external LED in the shape of a lock, similar to [23] and [24]. When keyboard focus is in a Secure Input Overlay, then the lock would glow. Otherwise, it would be off. Clearly, this would require substantial engineering changes. It also would rely on the user remembering to check the state of the indicator before entering sensitive inputs.

8.2 Security Extensions

Other future work is extending the security benefits of Secure Input Overlays by adding features and extending the design to new use cases. There are additional security features that would be useful. One would be, as in GuarDroid and VeriUI, to allow the user to confirm to what URL their inputs are being exfiltrated. This is a disruptive process that changes the application's experience. It might be worth the tradeoff in some cases. Another security feature would be to remedy the fact that remote attestation does not provide anonymity. A protocol such as direct anonymous attestation [71] could be used as an alternative attestation protocol that preserves anonymity. Although that might reduce usefulness of the attestation as a security and fraud detection signal.

There are also other use cases for the overall design. Secure Input Overlays deal specifically with user input, but there are other secrets that an application might want to just perform computation on before sending somewhere else, perhaps to check if something is in an address book, or that a GPS location is within a particular region. For this use case, the user would potentially have to pre-approve all the allowable computations, which could be integrated into an extended permission model.

Attested login on its own could be useful for embedded devices which need to access services on behalf of a user. For example, some public printers can access Google Drive and Dropbox [72]. These printers need to collect a user's credentials in order to access their files. Attested login could be used to convince a remote authentication provider that it is an authentic and trusted printer that is submitting those credentials.

9 Conclusion

This paper presented Secure Input Overlays, a new method for protecting sensitive inputs from untrusted applications. The strong isolation they provide still allows for rich client-side validation. A remote service provider receives the sensitive inputs along with an attested log of what computations the untrusted application ran on them. It can use this log to make trust decisions about the inputs. The design, implementation, and evaluation of this system showed that this approach is feasible, provides strong security properties, and could realistically be integrated into applications.

The problem solved was protecting data, which is one of two parts to establishing a trusted path for sensitive inputs on mobile devices. A local attestation solution is needed to complete the path. There is a large and growing body of research into both orthogonal legs of a trusted path. Secure Input Overlays, with their strong isolation, flexible validation, and remote attestation, are a novel and effective point in the design space.

10 References

[1] Facebook API Documentation, "Facebook login overview." https://developers.facebook.com/docs/facebook-login/overview/v2.3. Version 2.3.

[2] Google Developers, "Google identity platform." https://developers.google.com/identity/.

[3] Twitter Developer Documentation, "Sign in with twitter." https://dev.twitter.com/web/sign-in.

[4] T. Tong and D. Evans, "Guardroid: A trusted path for password entry," *Proceedings of Mobile Security Technologies (MoST)*, 2013.

[5] D. Hardt, “The OAuth 2.0 authorization framework.” IETF RFC 6749, October 2012. http://www.ietf.org/rfc/rfc6749.txt.

[6] J. Ruderman, “Same-origin policy.” https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Mozilla Developer Netork.

[7] D. Liu and L. P. Cox, “VeriUI: attested login for mobile devices,” in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, p. 7, ACM, 2014.

[8] M. Shehab and F. Mohsen, “Securing oauth implementations in smart phones,” in *Proceedings of the 4th ACM conference on Data and application security and privacy*, pp. 167–170, ACM, 2014.

[9] Lyft. https://www.lyft.com/.

[10] Braintree, a division of PayPal, Inc., “Braintree documentation.” https://developers.braintreepayments.com/ios+ruby/.

[11] Stripe, “Getting started.” https://stripe.com/docs.

[12] PCI Security Standards Council, “Pci ssc data security standards overview.” https://www.pcisecuritystandards.org/security_standards/.

[13] R. Rijswijk-Deij and E. Poll, “Using trusted execution environments in two-factor authentication: comparing approaches,” 2013.

[14] APWG, “Global phishing survey: Trends and domain name use in 1h2014,” tech. rep., September 2014.

[15] A. P. Felt and D. Wagner, “Phishing on mobile devices,” in *W2SP*, 2011.

[16] C. Marforio, R. J. Masti, C. Soriente, K. Kostiainen, and S. Capkun, “Personalized security indicators to detect application phishing attacks in mobile platforms,” *arXiv preprint arXiv:1502.06824*, 2015.

[17] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, “What the app is that? deception and countermeasures in the android user interface,”

[18] J. H. Saltzer, “Protection and the control of information sharing in multics,” *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[19] J. M. McCune, A. Perrig, and M. K. Reiter, “Safe passage for passwords and other sensitive data.,” in *NDSS*, 2009.

[20] M. Jakobsson and H. Siadati, “Spoofkiller: You can teach people how to pay, but not how to pay attention,” in *Socio-Technical Aspects in Security and Trust (STAST), 2012 Workshop on*, pp. 3–10, IEEE, 2012.

[21] 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*, pp. 291–304, ACM, 2013.

[22] Z. Chen and M. Cafarella, “TIVOs: Trusted Visual I/O Paths for Android,” Tech. Rep. CSE-TR-586-14, University of Michigan, May 2014.

[23] F. Dahan and B. Cornillault, “Secure mode indicator for smart phone or pda,” July 2 2013. US Patent 8,479,022.

[24] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, “Building trusted path on untrusted device drivers for mobile devices,” in *Proceedings of 5th Asia-Pacific Workshop on Systems*, p. 8, ACM, 2014.

[25] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[26] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati, “Spandex: Secure password tracking for android,” in *Proceedings of the USENIX Security Symposium (Sec)*, 2014.

[27] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, “Pios: Detecting privacy leaks in ios applications.,” in *NDSS*, 2011.

[28] A. Felt and D. Evans, “Privacy protection for social networking apis,” *2008 Web 2.0 Security and Privacy (W2SP’08)*, 2008.

[29] Windows Support, “Enable or disable secure logon (ctrl+alt+delete).” http://windows.microsoft.com/en-us/windows/enable-disable-ctrl-alt-delete-logon.

[30] Andrew Morton, “Linux 2.4.2 secure attention key (sak) handling.” https://www.kernel.org/doc/Documentation/SAK.txt.

[31] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, “Stronger password authentication using browser extensions.,” in *Usenix security*, pp. 17–32, Baltimore, MD, USA, 2005.

[32] Wikipedia Editors, “SiteKey.” http://en.wikipedia.org/wiki/SiteKey. Wikipedia.

[33] J. Lee, L. Bauer, and M. Mazurek, “Studying the effectiveness of security images in internet banking,”

[34] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, “The emperor’s new security indicators,” in *Security and Privacy,*

2007. SP'07. IEEE Symposium on, pp. 51–65, IEEE, 2007.

[35] J. Rutkowska and R. Wojtczuk, "Qubes os architecture," tech. rep., Invisible Things Lab, 2010.

[36] Mozilla Support, "How do i tell if my connection to a website is secure?." https://support.mozilla.org/en-US/kb/how-do-i-tell-if-my-connection-is-secure.

[37] Chrome Help, "Check a website's connection." https://support.google.com/chrome/answer/95617.

[38] 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*, pp. 601–610, ACM, 2006.

[39] D. Clarke, B. Gassend, T. Kotwal, M. Burnside, M. Van Dijk, S. Devadas, and R. Rivest, "The untrusted computer problem and camera-based authentication," in *Pervasive Computing*, pp. 114–124, Springer, 2002.

[40] R. Toegl, "Tagging the turtle: local attestation for kiosk computing," in *Advances in Information Security and Assurance*, pp. 60–69, Springer, 2009.

[41] ARM, "TrustZone." http://www.arm.com/products/processors/technologies/trustzone/index.php.

[42] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 263–278, USENIX Association, 2006.

[43] Android Open Source Project, "Android Open Source Project." https://source.android.com/.

[44] Android Open Source Project, "Android interfaces." https://source.android.com/devices/index.html.

[45] Android Open Source Project, "System and kernel security." https://source.android.com/devices/tech/security/overview/kernel-security.html.

[46] Android Open Source Project, "Application security." https://source.android.com/devices/tech/security/overview/app-security.html.

[47] A. Gargenta, "Deep dive into android ipc/binder framework." Android Builders Summit, 2013. http://events.linuxfoundation.org/images/stories/slides/abs2013_gargentas.pdf.

[48] A. D. Reference, "Binder." http://developer.android.com/reference/android/os/Binder.html.

[49] Trusted Computing Group, "Tpm main specification." http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[50] Trusted Computing Group, "Trusted platform module – faqs." http://www.trustedcomputinggroup.org/developers/trusted_platform_module/faq.

[51] The Chromium Project, "Tpm usage." http://www.chromium.org/developers/design-documents/tpm-usage.

[52] G. Coker *et al.*, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, 2011.

[53] A. Segall, "Using the tpm: Machine authentication and attestation." http://opensecuritytraining.info/IntroToTrustedComputing_files/Day2-1-auth-and-att.pdf. Slide 45.

[54] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX Association, 1993.

[55] J. Corbet, "BPF: the universal in-kernel virtual machine." https://lwn.net/Articles/599755/.

[56] J. Corbet, "Extending extended bpf." https://lwn.net/Articles/603983/.

[57] J. Schulist, D. Borkman, and A. Starovoitov, "Linux socket filtering aka berkeley packet filter (BPF)." https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation/networking/filter.txt?id=refs/tags/v3.19.2. Linux Source Code Documentation.

[58] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond.," in *USENIX Security*, pp. 97–112, 2013.

[59] L. Masinter, "An extension to HTTP: Digest access authentication." IETF RFC 2388, August 1998. http://www.ietf.org/rfc/rfc2388.txt.

[60] J. Goyvaerts, "How to find or validate an email address." http://www.regular-expressions.info/email.html.

[61] Wikipedia Editors, "Luhn algorithm." http://en.wikipedia.org/wiki/Luhn_algorithm. Wikipedia.

[62] J. R. Dabrowski and E. V. Munson, "Is 100 milliseconds too fast?," in *CHI'01 Extended Abstracts on Human Factors in Computing Systems*, pp. 317–318, ACM, 2001.

[63] B. Parno, "Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers," *Dissertations*, 2010. Paper 28.

[64] A. Starovoitov, "BPF backend." http://reviews.llvm.org/rL227008. LLVM Commit rL227008.

[65] Square, "OkHttp." http://square.github.io/okhttp/.

[66] A. D. Documentation, "Transmitting network data using volley." https://developer.android.com/training/volley/

index.html.

[67] Cyanogenmod. http://www.cyanogenmod.org/.

[68] A. Yu, D. Feng, and R. Liu, "Tbdrm: A tpm-based secure drm architecture," in *Computational Science and Engineering, 2009. CSE'09. International Conference on*, vol. 2, pp. 671–677, IEEE, 2009.

[69] R. Sailer *et al.*, "Attestation-based policy enforcement for remote access," in *Proceedings of the 11th ACM conference on Computer and communications security*, ACM, 2004.

[70] J. McCune *et al.*, "Flicker: An execution infrastructure for tcb minimization," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4, 2008.

[71] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 132–145, ACM, 2004.

[72] FedEx, "Convenient cloud printing at fedex office." http://www.fedex.com/us/office/instore-cloud-printing-2.html.