TruZ-View: Developing TrustZone User Interface for Mobile OS Using Delegation Integration Model

Kailiang Ying, Priyank Thavai, and Wenliang Du
Syracuse University, Syracuse, New York, USA
{kying,pthavai,wedu}@syr.edu

ABSTRACT

When OS and hypervisor are compromised, mobile devices currently provide a hardware protected mode called Trusted Execution Environment (TEE) to guarantee the confidentiality and integrity of the User Interface (UI). The present TEE UI solutions adopt a *self-contained design model*, which provides a fully functional UI stack in the TEE, but they fail to manage one critical design principle of TEE: a small Trusted Computing Base (TCB), which should be more easily verified in comparison to a rich OS. The TCB size of the self-contained model is large as a result of the size of an individual UI stack. To reduce the TCB size of the TEE UI solution, we proposed a novel TEE UI design model called *delegation model*. To be specific, our design reuses the majority of the rich OS UI stack. Unlike the existing UI solutions protecting 3-dimensional UI processing in the TEE, our design protects the UI solely as a 2-dimensional surface and thus reduces the TCB size. Our system, called *TruZ-View*, allows application developers to use the rich OS UI development environment to develop TEE UI with consistent UI looks across the TEE and the rich OS. We successfully implemented our design on HiKey board. Moreover, we developed several TEE UI use cases to protect the confidentiality and integrity of UI. We performed a thorough security analysis to prove the security of the delegation UI model. Our real-world application evaluation shows that developers can leverage our TEE UI with few changes to the existing app's UI logic.

CCS CONCEPTS

• **Security and privacy** → **Mobile platform security**;

KEYWORDS

TrustZone, Android, UI safety

ACM Reference Format:
Kailiang Ying, Priyank Thavai, and Wenliang Du. 2019. TruZ-View: Developing TrustZone User Interface for Mobile OS Using Delegation Integration Model. In *Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19), March 25–27, 2019, Richardson, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3292006.3300035

1 INTRODUCTION

Nowadays, users perform various essential activities, including banking, shopping, and financing, through the smartphone User Interface (UI). Because of the heavy reliance on this single interface, the security of the mobile software stack controlling UI has become increasingly critical. Unfortunately, CVE results of mobile OSes are not positive [6]. Take Android OS as an example – the number of vulnerabilities has skyrocketed over the last eight years. The mobile OSes cannot prevent untrusted code embedded in applications from running, which then leads to a broad attack surface. Untrusted code can exploit many vulnerabilities and can eventually manage to compromise the OS. Once the mobile OS is compromised, the last UI defense will be gone, and UI will then be controlled by malware. What's worse, malware can spoof actions on behalf of users without their consent.

Trusted Execution Environment (TEE), a technology that can secure the UI when the OS is compromised, has been developed to address this prominent security issue. As the most commonly deployed TEE on mobile devices, ARM TrustZone protects the UI in an isolated environment, *secure world*, inaccessible by the compromised mobile OS and normal apps, while in the *normal world*, untrusted normal apps run freely. We consider that there are mainly two directions to design TrustZone UI, as shown in Figure 1. The current design direction of the TrustZone UI solutions [4, 20, 30] is established to support a fully functional UI stack in the secure world mainly because these UI solutions support the *Trusted Application* (TA) to build isolated UI in the secure world. Such a UI design direction requires an isolated code stack (i.e., UI stack and TA) in the secure world and we call it the *self-contained UI model*, as shown on the left side of the Figure 1. Research works [3, 18, 28, 32] built various UI protection mechanisms (i.e., TAs) on top of the current TrustZone UI solutions [4, 30]. Although the self-contained UI model does provide UI security measures, we consider that the current TrustZone UI model fails to manage one critical design principle of TEE: a small Trusted Computing Base (TCB), which should be more easily verified in comparison to a rich OS. The TCB size of the self-contained model is large because it requires an individually functional UI stack in the secure world.

We intend to reduce the TCB size of the TrustZone UI. Our primary observation is that the functionalities of the secure-world UI stack can be further divided into UI development, UI services, and UI interaction, as shown in Figure 2. The main objective of the TrustZone UI solution is to secure the UI interaction, which involves displaying sensitive data on the screen and taking sensitive input from users. In order to protect the UI interaction, the self-contained UI model decides to also include both the UI development and the UI services in the secure world. When developing UI, developers construct the UI layer by layer. For example, developers can first

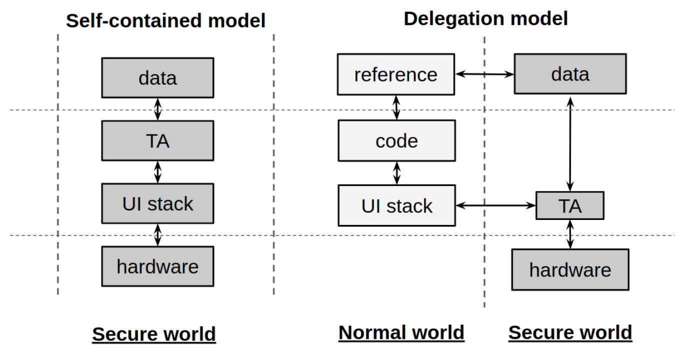


Figure 1: TrustZone UI Design Models

define the background layout (layer 1) and then put buttons (layer 2) on top of the layout (layer 1). The main tasks of the UI services are propagating touch events and compositing the screen content based on the multilayered UI structure defined by developers. Because of the complexity of this multilayered UI processing, the TCB size of the UI development and UI services is large. However, when users interact with the secure-world UI, they solely interact with a 2-dimensional surface on the screen. On a 2-dimensional surface, the protection of displaying the sensitive data is equivalent to the protection of a region on the image and the protection of the users' sensitive input is equivalent to the protection of users' touch coordinates on the screen. We can significantly reduce the TCB size if the secure world solely protects the UI as a 2-dimensional surface and leaves the 3-dimensional UI processing in the rich OS.

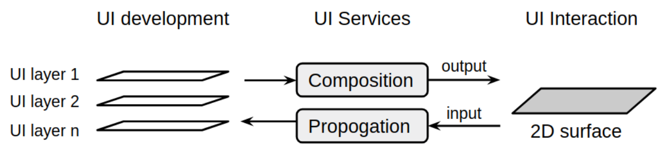


Figure 2: TrustZone UI Stack

Based on our observation, we propose the *delegation UI model* for TrustZone, as shown on the right side of the Figure 1. The main idea of our model is to delegate the UI development and the UI services to the rich OS. The normal world conducts the 3-dimensional UI processing, which does not contain any sensitive data. The output of the normal-world UI stack becomes an image on the screen. To display the sensitive data stored in TrustZone, the secure world takes a screenshot and overlaps the protected data on top of the screenshot before displaying them to users. To protect the users' sensitive input, our design keeps users' touch coordinates on the screenshot in the secure world. The protected data will never be leaked to the normal world. With this new design model, our approach significantly reduces the size of the secure-world UI stack from 3-dimensional UI processing to 2-dimensional UI processing.

This paper makes the following contributions: (1) provides the first study to propose a novel TrustZone UI design model called *delegation model* and to systematically study the properties and design principles of this new model; (2) the implementation of the TrustZone UI solution, called *TruZ-View*, applying the delegation model to protect UI interaction when the mobile OS is compromised; (3) the performance of a thorough security analysis to prove the security of TruZ-View; (4) the evaluation of our system by using real-world applications.

2 PROBLEM

In this section, we discuss our threat model, research problems, and design challenges when applying the delegation UI model. We systematically compare the design trade-off between the self-contained model and the delegation model.

2.1 Threat Model

The user of the device is trusted. The normal world filled with apps and Android OS is untrusted because it may attempt to take screenshots that contain users' confidential information or to keylog users' confidential input (e.g., password and credit card number), and may spoof an unauthorized action on users' behalf without their confirmation. The secure world that includes the Trusted Applications (TA) and TEE OS is trusted and preserves users' confidentiality and integrity when the normal world is compromised. We assume that the server remains trusted after it is authorized by the user. The authorized server aims to protect the users' confidential data and verify the integrity of the users' request.

2.2 Problem Statement

The research problem of protecting UI can be further broken down into protecting UI display and protecting UI input. In this paper, we apply the delegation UI model and mainly answer (1) how to securely display the protected data, which is downloaded from the server; and (2) how to securely take users' sensitive input in TrustZone.

2.3 UI Design Models Design Trade-off

To better understand characteristics of different design models, we systematically compare the design trade-off between the self-contained UI model and the delegation UI model in three aspects, namely security, system design, and application impact. The comparison result is summarized in Table 1. Relying on the comparison result, we further derive design principles when we apply the delegation UI model.

Security. The TCB size and the isolation boundary are two essential attributes to measure the security of TrustZone solution. The self-contained model requires a separate UI stack to support TAs to write the UI logic, which requires a large TCB in the secure world. By contrast, the delegation model pushes the majority of the UI stack in the normal world and requires a small TCB in the secure world. As a trade-off for the TCB, the self-contained model could maintain a clean isolation boundary between two worlds because the software stacks are completely isolated in two worlds, while the isolation boundary for the delegation model is not clear yet. Therefore, the UI design that applies the delegation model has to answer this challenging question.

System design. The code reusability and the modularity are two critical measurements for the TEE system design. First of all, the code reusability implies that the software stack developed for

Table 1: UI Design Models Comparison Summary

Model	Security		System Design		Application Impact		
	TCB	Isolation	Reusability	Modularity	Transparency	Consistency	Rich Usability
Self-contained	large	clean	low	clean	low	low	high
Delegation	small	unclear	high	unclear	high	high	none

one world can be reused for another world. The software developed based on the self-contained model always suffers a low code reusability because different OSes have different interfaces and development standards. In contrast to the self-contained model, the delegation model reuses the majority of the normal-world software stack including OS level and application level. Therefore, the delegation model can obtain a high code reusability. Secondly, modularity suggests that the system update of one world should not affect the other world. The self-contained model can update systems independently because software stacks of both worlds are entirely separated. As a trade-off for the code reusability, the delegation model cannot merely modularize two worlds into two modules. The UI design that applies the delegation model has to provide an insight to preserve the modularity between two worlds.

Application impact. We use transparency, consistency, and rich usability to assess the TrustZone design impact on applications. Firstly, transparency is how many efforts application developers need to make to integrate the TrustZone UI solution with their applications. The self-contained model requires application developers to work with vendors to develop UI in the secure world, and such an approach requires much effort of application developers to leverage the TrustZone UI. Moreover, putting application-specific logic in the TEE is not secure because it is the reason that broadens the normal-world attack surface. The delegation model has high transparency to applications because our model reuses the normal-world application UI logic to develop the secure-world UI and does not put any application-specific logic in the TEE. Secondly, the consistency means whether users have the same UI experience across worlds. Users usually endure an inconsistent user experience when interacting with the self-contained UI model because the screen images are produced by separate UI stacks. The delegation model can provide consistent UI experiences across worlds because a single UI stack produces screen data for both worlds. Thirdly, the rich usability refers to whether the secure-world UI supports the rich UI functionalities such as animation and UI extensible services such as autocomplete and spell checker. As a trade-off for the TCB, we decide not to support any rich UI functionality like animation inside the secure world while the self-contained model could support rich UI depending on how large its TCB is.

Delegation UI model design principles. To reach a conclusion on the comparison, we summarize design principles of the delegation UI model. When we apply the model to design the TrustZone UI, these principles will help us preserve the aforementioned properties of the delegation model: (1) maintain a small TCB in the secure world and reuse the rich OS for non-sensitive operation as much as possible; (2) find a clean cut for the isolation boundary; (3) require a minimum effort from the TrustZone integration for applications and the system update; (4) maintain a consistent UI experience across the worlds.

2.4 UI Design Challenges

In this section, we discuss UI design challenges by applying the delegation model. The secure world loses many capabilities during this TCB reduction process because our design model pushes the majority of the UI stack out of the secure world. However, some of the lost capabilities are important for the secure world to protect UI and become challenges for our UI design.

First of all, the secure world loses the capability to develop UI independently. We delegate UI development to the untrusted normal world. The data processed by the normal world is considered as untrusted. How can the secure world leverage the **untrusted** normal-world UI stack to develop UI for the secure world?

Secondly, the secure world loses the UI layer information because the delegation model solely protects the 2-dimensional image in the secure world while all the UI layer information is processed in the normal world. However, some of the UI layer information is important to define how to protect the UI display and the UI input. We need to find way to recover the **lost UI layer information** in the secure world.

3 IDEA

In this section, we discuss our key ideas to leverage the untrusted UI stack and to recover the lost UI layer information.

3.1 Splitting the UI Rendering Process

Our main idea of producing the secure-world UI is to split the normal-world UI rendering process at the last step. We categorize all existing TrustZone UI solutions as splitting the UI rendering process at different layers. The Figure 3 is an abstraction diagram of splitting design options. The first option [4, 20, 30] is to split the UI rendering at the application layer and to put all three layers into the secure world. The second option [32] is to split the UI rendering process at the framework level. Such a splitting option leaves the application logic in the normal world and puts the remaining two layers in the secure world. The challenge of the 2nd option is to maintain the binding between the application code in the normal world and the UI in the secure world. Although these two existing splitting options provide UI security measures, such splitting approaches require a separate UI stack to support the secure-world UI.

We suggest splitting the UI rendering process at the last step. The normal world produces the screen data without having the TrustZone-protected data. Our design takes the normal-world screenshot as the secure-world UI. The secure world protects the sensitive data display by overlapping the protected data on top of the screenshot and protects the users' input by keeping the touch coordinates

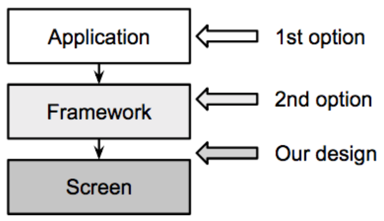


Figure 3: UI Rendering Split Options

inside the secure world. Our splitting option allows us to protect the UI interaction on a 2-dimensional surface.

3.2 Why securing a 2D UI is sufficient?

Here we explain why securing the display of sensitive data on a 2-dimensional surface is sufficient. First of all, users see the UI as a 2-dimensional image. Second, the sensitive data displayed on the UI is also stored in a 2-dimensional format in the secure world. We can directly perform image operation on the 2-dimensional surface to overlap the sensitive data on top of the image, thus protecting a 2-dimensional surface is sufficient to protect the sensitive data display.

Next, we describe why securing the users' sensitive input on a 2-dimensional surface is adequate. First of all, the initial form of the touch event is touch coordinates, which are a pair of float numbers. Second, all security-related inputs (i.e., confidential input, integrity-preserved input) are consumed at the UI's top layer, which users can see. There is no need to propagate such touch events to lower UI layers. For instance, when users type a password (i.e., confidential input) through a keyboard, they intend to click on the keyboard layer that they can see, not on the underlying invisible layers. The touch event is always consumed in an area of the 2-dimensional surface where users can see. Thus protecting the UI's top visible layer, a 2-dimensional surface that is enough to protect the UI input.

In this section, we convey our high-level isolation boundary. We further performed a detailed security analysis in Section 6 to prove the security of the 2-dimensional UI protection.

3.3 Recover UI Layer Information

The UI layer information is missing in the 2-dimensional surface because our design decides to split the UI rendering at the last step. However, some of the layer information is important to preserve the consistent UI experience across worlds. For example, the secure world needs to know how to protect the users' input (e.g., keyboard layout) and know where to display the protected data in the screenshot.

Our main idea of the recovery of the UI layer information is to let the `Android view system` send views' coordinates to TrustZone, as shown in Figure 4. We observe that the UI layer information is initially defined by the UI view that is the basic building block to construct the UI in the normal world. Our design allows developers to label certain views as the TrustZone-protected when they develop the UI. We follow the same UI development workflow and create a TrustZone tag in the view system. Our modified view system sends coordinates of TrustZone-enabled views to the secure world. Based on these coordinates, our design can recover the UI

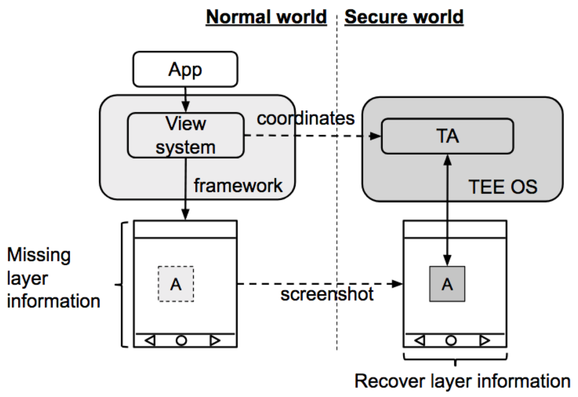


Figure 4: Recover UI Layer Information

layer information on a 2-dimensional surface. We have conducted a thorough security analysis in Section 6 to prove that the normal world cannot misuse the wrong view coordinates.

Our design provides easy-to-use TrustZone UI building blocks for developers. Developers can simply add these UI building blocks into an existing application's UI. We further categorize these UI building blocks into (1) confidential display, (2) confidential input, (3) integrity-preserved interaction. In Section 4, we will describe our UI building blocks design in detail.

4 UI DESIGN

In this section, we discuss our UI building blocks based on three categories: (1) confidential display, (2) confidential input, (3) integrity-preserved UI interaction. We refer to such views as *truz-views*.

4.1 Confidential Display

Application developers can protect users' confidential information displayed on the screen by adding a truz-view.

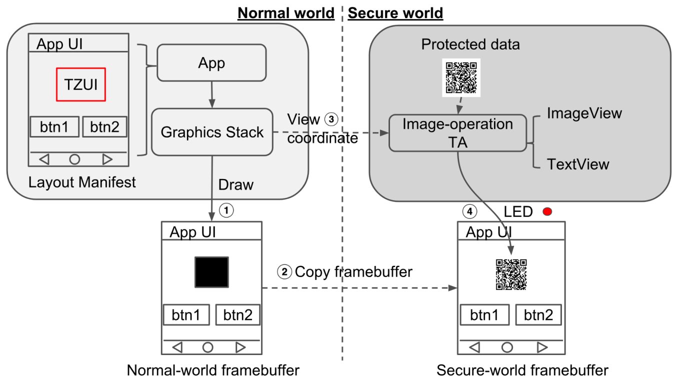


Figure 5: Confidential Display Design

Our confidential display design enables the normal-world graphics stack to recognize the truz-view and to work with TrustZone to display the protected data stored in the secure world. The normal-world graphics stack composites the app's UI into an image but leaves the area of truz-view blank, as shown in Figure 5 ❶.

Users first see the normal-world UI, as shown in Figure 6 WeChat example. To view the TEE-protected data, users click on the truz-view's area. Our design handles this particular click and takes a screenshot of the current UI. The screenshot is then sent to the secure world, as shown in Figure 5 ❷. Our design further transfers the coordinate and size of truz-view to the secure world, as shown in Figure 5 ❸, to guarantee that the protected data is filled in the secure region of the app's UI. The secure world then obtains the exclusive control of the screen hardware and fills in the protected data on the screenshot and displays the complete UI to the user, as shown in Figure 5 ❹. Throughout this process, users see the familiar app's UI with the protected data filled in. We use WeChat barcode payment UI to illustrate our UI in the secure world (Figure 6 WeChat secure world). Users scan the barcode in the secure world and can click the back button to switch the control of the screen to the normal world. When in the normal world, the protected data disappears and the truz-view's area is shown blank on the screen (Figure 6 WeChat normal world). In Section 6.1, we explain why the users' confidential information will not be leaked during the confidential display process.

We totally developed two types of confidential display UI building blocks, namely ImageView and TextView. Developers can simply embed the confidential display UI into the app using the code in Listing 1. They only require the secure world to perform image operation to render the protected data. The confidential information can be downloaded from servers to the secure world. Our design returns a reference to the protected data to the normal world. Developers then use the reference to display the TEE-protected data in the secure world. We will discuss our implementation for data downloading and reference management in Section 5.

Listing 1: Application change for confidential display

```
<View android:tzSecure="true"
android:src="reference to protected data" />
```

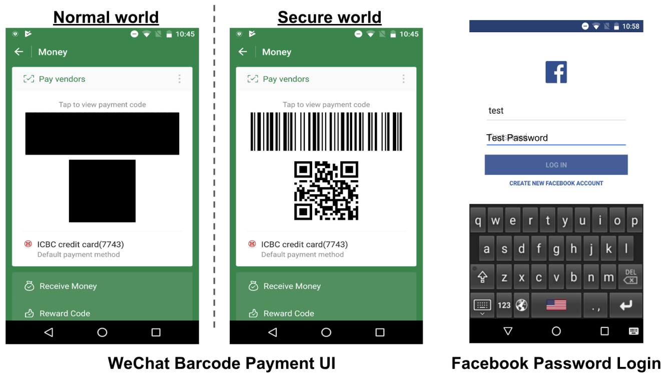


Figure 6: Confidential Display and Input Use Cases

4.2 Confidential Input

Application developers can protect users' confidential input by embedding a truz-view in the app's UI. Apps commonly accept users' confidential input, such as password and account number, through a keyboard, or let users sign on the smartphone to approve a purchase request. Our design mainly focuses on two common confidential input UI: typing password through a keyboard and signing through a signature pad. We use the keyboard to explain our main idea and the concept of securing users' signature is similar.

The keyboard is a standalone application. Android allows apps to use either a system keyboard or a customized keyboard to process users' input. Our design enables both types to protect users' keystroke inside the secure world. To give feedback on users' inputs, the app, which uses the TrustZone-enabled keyboard, needs to display secure-world keystrokes on a text field (i.e., EditText). Application developers can then embed a TrustZone-enabled EditText to be bound with the secure-world keyboard.

The normal world composites the app's UI including the TrustZone-enabled keyboard and EditText into an image. When the normal world displays the UI, our design takes a screenshot and then sends the image to the secure world. Then the secure world takes over the control of the screen and lights on a LED to notify users that they can securely enter their confidential input on the screen.

Users see the same app's UI in the secure world, as shown in Figure 6 Facebook example. To allow the secure world to handle users' click on the keyboard, our design sends the coordinates of truz-views and the layout of the KeyboardView to the secure world. Note that Android requires each Keyboard app to have an XML layout file and our design reuses this file to gain the layout of the keyboard in the secure world. When the user types in the keyboard, the secure world knows how to fill in the corresponding character on the EditText. Once users finish typing, they can click the back button to switch the control of screen to the normal world. Our design stores the user's input in the secure world and returns a reference to the input to the normal world.

The developers can simply embed the confidential input UI into the app using the code in Listing 2. We assume that the input is processed only by the server, not by the client. We will explain our implementation of secret transmission in Section 5. In Section 6.2, we explain why the users' confidential information will not be leaked during the confidential input process.

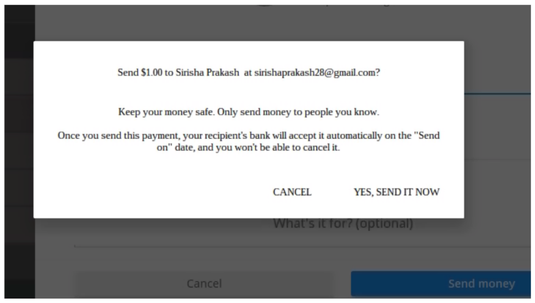
Listing 2: Application change for confidential input

```
<View android:tzSecure="true" />
```

4.3 Integrity-preserved UI Interaction

App developers desire to obtain the user's consent without it being modified by the compromised OS. Our integrity-preserved UI design focuses on a number of UIs such as Dialog, Confirmation Activity, PIN Pad, Pattern Locker, and Password Locker. The common characteristic of these UIs is that they all require a group of view elements to display the confirmation message to the user and to request the users' agreement to move forward. We use Dialog to illustrate our main idea, and the way to secure other integrity-preserved UIs is similar.

Take Chase monetary transaction Dialog as an example – developers want to allow users to confirm the transaction information displayed in the secure world, as shown in Figure 7. To display the confirmation message, app developers can embed a TrustZone-enabled TextBox. To obtain the user's decision, app developers can embed two TrustZone-enabled buttons, one for a positive decision


Send $1.00 to Sirisha Prakash at sirishaprakash28@gmail.com?

Keep your money safe. Only send money to people you know.

Once you send this payment, your recipient's bank will accept it automatically on the "Send on" date, and you won't be able to cancel it.

Figure 7: Chase Payment Confirmation Dialog

and the other for a negative decision. Our design recognizes these specialized views and asks the normal-world graphics stack to composite the app's UI including the truz-view's region, as shown in Figure 8 ❶. When the normal world displays the UI, our design takes a screenshot and sends it to the secure world. To fill in the integrity-preserved content on the right region of the app's UI, our design transfers the views' coordinates, the confirmation message, and properties of buttons (i.e., positive or negative) to the secure world, as shown in Figure 8 ❷. The secure world first erases the normal-world text on the TrustZone-enabled views and then fills in the text on these views, as shown in Figure 8 ❸. Two options guarantee the integrity of the text: (1) the TrustZone provides the text content and (2) the server attests the text. For example, the secure world fills in 'Yes' on the positive button and 'Cancel' on the negative button and the confirmation message on the TextView region. Our design prevents the normal world from fooling users. Preserving the integrity of the normal-world UI is nontrivial and we have performed a thorough security analysis in Section 6.3.

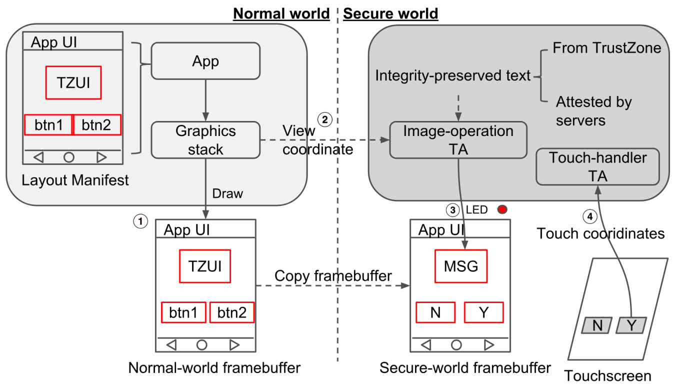


Figure 8: Integrity-preserved UI Interaction Design

Users see the familiar app's UI with the integrity-preserved data filled in. When the user confirms the message, our design generates an attestation of the message inside TrustZone, as shown in Figure 8 ❹. We will explain how the TrustZone and the server exchange the attestation key in Section 5. Once the user clicks on the button, our design injects the user's click along with the attestation back to the normal world. The normal-world OS continues to propagate the touch event to the corresponding Dialog button.

Developers can simply embed the integrity-preserved UI into the app using the code in Listing 3. In Section 6.2, we explain why the users' confidential information will not be leaked in our injecting click design decision.

Listing 3: Application change for integrity-preserved UI

```
<View android:tzSecure="true"
android:tzProperty="Positive" />
```

4.4 Hardware Design

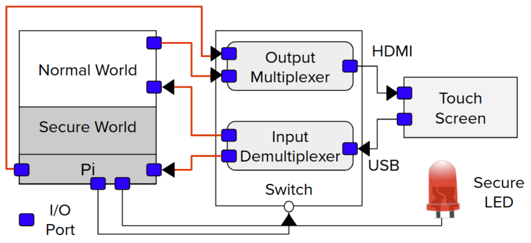


Figure 9: Hardware Design

Our hardware platform runs Android 9.0 in the normal world and OPTEE [22] in the secure world. We built our prototype using the HiKey board as our base platform [1]. We used a TFT LCD panel as the screen. The screen uses the HDMI interface for the display and the USB interface for controlling the touch.

As shown in Figure 9, to allow the secure world to drive the screen, we introduce an additional board (i.e., Raspberry Pi) controlled by the secure world. The secure world communicates with the Raspberry Pi through the serial communication (i.e., UART) and the secure world has the exclusive control of the UART. We achieve the screen isolation at the circuit level. We connect the screen I/O to the multiplexer/demultiplexer. The multiplexer takes the HDMI signal from both worlds and outputs the signal from one of the worlds to the screen. The demultiplexer takes the touch input from the screen and gives it to one of the worlds. The secure world controls the switch of the multiplexer/demultiplexer. Each world has separate I/O ports that are connected to multiplexer/demultiplexer. Users are informed that the screen is controlled by the secure world when an LED is turned on. We configure the TrustZone Protection Controller (TZPC) to allow the secure world to have exclusive control of the switch, LED indicator, and UART. Our hardware implementation is shown in Figure 10.

The main reason to use a separate Raspberry Pi is that the OPTEE does not have a series of device drivers for display and input. Developing such drivers requires accessing the design reference documents of the hardware platform (e.g., HiKey). However, the silicon vendors commonly conceal their implementations and NDA-protected documents, making it a great challenge for third-party developers to write drivers. We also consider that writing drivers is unnecessary for our project as our design mainly targets those vendors who have driver stacks for their own hardware. If the vendors adopt our design, they can eliminate the Raspberry Pi in our design by directly deploying the driver stacks in the secure world. Our design only uses Raspberry Pi as a bridge between the secure-world OS and the screen hardware. We do not use any other Linux OS functionalities in Raspberry Pi. We consider that our hardware design provides a reference for the research communities that desire to conduct a hardware-related research but are discouraged by device-driver accessibility in the secure world.

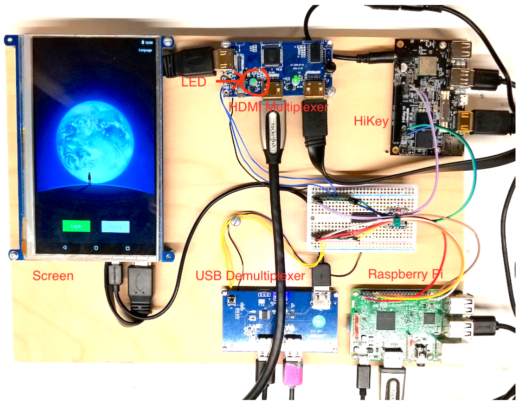


Figure 10: Hardware Implementation

5 DATA PROTECTION AND MANAGEMENT

In this section, we discuss how we protect the data during online data transmission and during the offline data usage. During the online data transmission, our system allows developers to use standard network protocols (e.g., HTTP, SSL) to transfer the confidential data between the server and TrustZone. During the offline data usage, we allow normal-world applications to display the data stored in TrustZone without it leaked into the normal world.

High-level solution. The high-level idea of data protection is shown in Figure 11. Having strong incentives to protect users' sensitive data stored in the cloud, the server establishes a TLS encrypted channel with TrustZone and sends the protected data through HTTPS. The normal world cannot eavesdrop on the network traffic without obtaining the TLS encryption key protected by TrustZone. The secure world decrypts the HTTPS packet and protects the confidential data. Our solution returns a reference to the protected data to the normal world and keeps the data in the secure world. This design ensures that the normal world cannot read and modify the data stored in TrustZone.

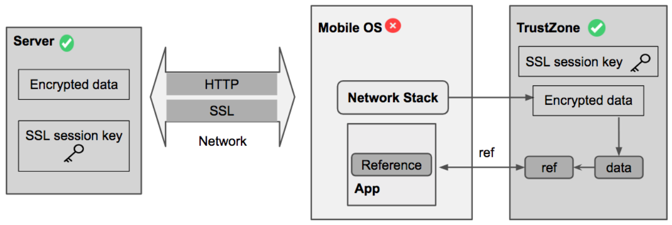


Figure 11: Data Protection High-level Solution

Split SSL. The pioneering researcher has invented a solution called *Split SSL* [32] to protect the online data transmission by using TrustZone. We build on top of the Split SSL to transfer confidential data between the server and TrustZone. The Split SSL secures the server-TEE communication through the TLS protocol and conducts all crypto operations (e.g., encryption, PKI) inside TrustZone. One advantage is that the Split SSL is entirely transparent to the server so that the server can use standard network protocols for data transmission. The other advantage of the solution is that it allows the TrustZone-protected data and the normal-world data to be mixed in a single HTTPS packet. Their solution is to use an HTTP header to create a boundary between two types of data. They have conducted a thorough security analysis to prove that no information was leaked during the Split SSL connection. The Split SSL allows TrustZone and the server to establish a mutual trust during the login process and exchange the attestation key with TrustZone through the Split SSL. By checking the attestation of the HTTP requests after the login, the server can test whether the request comes from TrustZone.

Engineering Challenges. The Split SSL [32] mainly focuses on uploading a secret to the server. To apply the same solution to the download of the protected data from the server, we need to overcome two additional engineering challenges. Firstly, the HTTP response may be fragmented into multiple TLS records and TEE does not know what to return without having the complete data. The fragmentation is caused due to the limitation of the TLS record length. The protected data will be sealed into multiple TLS records if the data size exceeds the TLS record limitation. Secondly, the secure-world reference may break the normal-world application logic because the logic written by developers should be operated on the actual data, not on the reference to the protected data. The delegation model requires developers to spend the minimum effort integrating TrustZone protection. Thus our design needs to find a way to let the application logic operate on our reference without breaking it.

HTTPS Packet Fragment. The Figure 12 is the overview of our TrustZone-protected HTTPS download. Our system can handle the HTTPS packet fragmentation and return a partial reference to the normal world. One single HTTP packet that is put into the TLS layer could be potentially sealed into multiple TLS records because the TLS record has a maximum of 16KB length, as shown in Figure 12 ❷. Once the secure world knows the size of protected data from the HTTP header, and if the size exceeds the TLS record size limit, the secure world will start to track the offset of currently received data. The protected data will be firstly saved in chunks with the length of the TLS record. Our solution returns a partial reference for each TLS record. The partial reference is combined with the reference to the protected data and the sequence order of the protected data. We embed the reference in a shadow copy of the TLS record, which has the same length as the TLS record, and return the shadow copy to the normal world, as shown in Figure 12 ❹.

Our solution manages all partial references and their corresponding data in a reference management table. Table 2 is a simplified example of our reference management table. We bind the protected data with the server name. If the data is fragmented, we have metadata to describe the sequence and the position of the fragmented data. Once the secure world gets all pieces, our solution concatenates all parts into one buffer and saves it in the secure world.

Reference Design. Applications typically store the data in two places: memory and file. When applications save the data in the memory, our design embeds the reference in a shadow copy that has

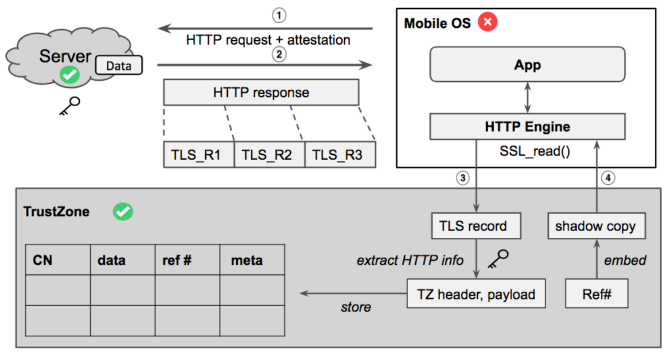


Figure 12: TrustZone-enabled HTTPS Downloading

Table 2: Reference Management Table

CN	data	ref	metadata
a.com	ptr1	12345678(1)	Seq=1,offset=0:16KB
a.com	ptr2	12345678(2)	Seq=2,offset=16:32KB
a.com	ptr3	12345678(3)	Seq=3,offset=32:42KB

the same length as the protected data and returns the shadow copy to the normal world. The shadow copy also preserves the file header in the shadow copy to ensure that the returned reference does not break the normal-world application logic. When applications save the data in the file, the data will be saved in the secure-world file system. The application can ask the secure world to save the content and get a secure-world file path back. Application developers can display the confidential data using both types of reference.

6 SECURITY ANALYSIS

In this section, we present the security analysis of our TrustZone UI design. Our design can guarantee the confidentiality of UI display and UI input, and the integrity of UI interaction when OS and hypervisor are compromised. We also analyze the security of the secure downloading feature that we built on top of the Split SSL [32]. Our analysis assumes that the TrustZone hardware platform is trusted and the secure boot process has initialized the integrity-verified secure-world OS. Hardware attacks, side-channel attacks, shoulder surfing, and DOS attacks are considered out of scope.

6.1 Confidential Display Security Analysis

The objectives of adversary include stealing the TEE-protected data stored in the secure world, accessing the data loaded in the secure-world memory and that displayed on the screen, inferring the data based on the normal-world view information, and accessing the secrets without authorization of the real user.

The normal world cannot steal the protected data stored in the TEE because the protected data is stored in the TEE trusted storage, which is a standard TEE storage solution. When the TA inserts the protected data into the secure-world framebuffer, the TA loads the data into a piece of secure-world memory. Because of the TEE memory isolation, the normal world cannot access the secrets in the secure-world framebuffer.

The normal world cannot access the protected data displayed on the screen. As mentioned in the hardware setup in Section 4.4, the secure world has an exclusive control over the secure-world I/O ports. The normal world thus cannot access the displayed content when the secure world controls the display. We clean up the screen cache before the CPU switches to the normal world to prevent the normal world from reading the cache residue.

The normal world cannot infer the confidential data based on the view information because the content of the view is protected in the secure world. The untrusted normal world can only DOS the confidential display with wrong view coordinates or the wrong framebuffer content.

The unauthorized user who obtains the smartphone cannot see the secrets protected by the secure-world Pattern Locker. For example, some secrets, such as ID card, are visible to users only after the authorization. Our design will not unlock the Pattern Locker if users do not know the pattern and will prevent them from continuing to brute force patterns after 10 failed attempts.

6.2 Confidential Input Security Analysis

The objectives of the adversary include stealing users' inputs, tricking users to type the secret in the normal world, and inferring users' clicks from the secure-world return values.

The normal world cannot get the user's input from the touchscreen. As discussed in Section 6.1, the normal world cannot access the touchscreen when the secure world controls the screen hardware. Therefore, the normal world cannot get the user's touch coordinates generated by the touchscreen.

The normal world cannot fool users into typing secrets. Although the normal world can fake the same UI look and trick the users to type secrets, we use an LED light to indicate to users whether they are typing in the secure world. The secure world obtains the exclusive control over this LED, so the normal world cannot control the indicator. The existing work [32] has done the survey to study whether users can correctly recognize the LED as the world indicator. The study result concludes that users are capable of differentiating worlds based on the LED light.

The normal world cannot infer the touch coordinates from the TA return values because both confidential-display and confidential-input TAs only return the back button event to the normal world. In the case of the Confirmation Activity, only the confirmation button event is returned to the normal world. The normal world cannot use the confirmation button to construct a confidential keyboard and let the secure world return the user's confidential input because only the secure world can render the texts (e.g., Yes, Cancel) for confirmation buttons. The normal world cannot fool users if it cannot render contents on buttons.

The untrusted normal world can only DOS the confidential input with wrong view coordinates or the wrong framebuffer content.

6.3 Integrity-preserved UI Security Analysis

The objectives of the adversary include fooling users into confirming the wrong action, and forging the attestation.

We prevent the normal world from drawing the untrusted confirmation content on the UI. Because the normal world has the full control of the UI drawing, the normal world can draw the UI in a way that makes the user's intended message visible to users and makes the actual attested message inconspicuous to users. To prevent such an attack, the secure world blurs all non-sensitive regions that are not labelled as TrustZone-enabled in the screenshot. Furthermore, before rendering the integrity-preserved text on the truz-view, the TA cleans the secure view region to prevent the normal world from displaying untrusted contents in the secure view region. The TA fills in the text that is either from TrustZone or is verified by the server.

The normal world cannot fool users into making a wrong decision by providing the wrong view coordinates. The normal world can send the wrong coordinates of truz-views to the secure world. For example, the normal world can swap coordinates of positive and negative buttons and send them to the secure world. However, the secure world renders texts including confirmation message and button text. For example, the TA ensures that the negative button only renders the negative text (e.g., Cancel) on the button.

We provide users with solutions to check the identity of the TEE-protected data before the secure world displays the data. The normal world can fool the secure world into displaying the wrong data because the normal world controls the reference to the data. For example, the attacker can ask the bank server to load the attacker's receiving payment barcode into the secure world. When the user receives money by providing the barcode, the normal world can provide the secure world with the reference to the attacker's receiving payment barcode. In that case, users have to check the identity of the data before the secure world shows the data. We allow the authorized server to provide metadata along with the protected data when the secure world downloads data from the server. If the data needs additional metadata to describe the identity of the data (e.g., barcode), the server can use our Pattern Locker to display the barcode's metadata in the secure world before it shows the barcode. The TA allows users to confirm the metadata (e.g., account name) in the secure-world LockPatternView before displaying the content. In our design, we assume that the metadata from the authorized server is trusted and the user can verify the integrity of the data using our TrustZone-enabled LockPatternView.

The normal world cannot forge the secure-world attestation. The server and TrustZone exchange the attestation key through the Split SSL. The normal world cannot forge the attestation without the key. Furthermore, we append a nonce when computing the attestation to avoid replayability.

6.4 Data Download Security Analysis

Attackers' goal includes stealing the protected data and loading malicious data into the TEE.

To prevent the server from leaking the confidential data, when developers send the download request to the server, our solution inserts an attestation that the normal world cannot forge in the HTTP header field. The server always verifies the attestation before sending the protected data.

To avoid attackers from arbitrarily downloading data into the TEE, the secure world has a whitelist that stores servers that users sign in through the secure-world UI. The secure world refuses to save data that comes from an untrusted domain.

The integrity of the HTTPS response cannot be changed by the normal world because the HTTPS packet is first decrypted inside the secure world.

During the data transmission, the untrusted normal world cannot eavesdrop on the encrypted channel because we conduct the SSL key exchange between the server and TrustZone. Without knowing the SSL encryption key, the normal world cannot decrypt the HTTPS response.

After the protected data is stored in the secure world, our design prevents the data from being uploaded to an untrusted domain. Our design binds each piece of data with a whitelist of trusted domains, which can be set by the user or the trusted server.

7 EVALUATION

In Section 4, we have demonstrated applications of our design (i.e., ImageView, KeyboardView, etc.). In this section, we further evaluate TruZ-View from three aspects: TCB reduction, ease of adoption, and performance. For TCB reduction, we compare our TCB size with a standard UI stack and network stack. We quantified the effort that developers take to adapt our solution using the real-world apps and measured the performance overhead that we introduced to both the normal world and secure world. We conducted the evaluation on HiKey 620 board [1], which runs both Android 9.0 in the normal world and OPTEE OS [22] in the secure world.

7.1 TCB Reduction

Our solution eliminates large swaths of code, compared with the existing TrustZone UI works [4, 20, 30], which installed an individual UI framework in the TEE. The comparison result is summarized in the table 3. Take a standard UI stack as an example - Xlib [10] is used for the UI display and the UI input. Xlib has more than 100k LOC. The widget toolkit called tk [8] that runs on top of Xlib contains half a million LOC.

We installed a modified font rendering library [7] that contains 1453 LOC in the secure world. Our TA code is only approximately 2000 LOC. Notably, we require screen drivers in the secure world for display and input. All existing works [4, 20, 30] also installed drivers in the TEE to keep a robust TCB. Our solution only requires a SSL crypto library in the secure world for secure server communication.

Table 3: TCB Reduction

Component	Existing TCB size	Our TCB size
UI stack	100k LOC	1453 LOC
UI widget	500k LOC	2000 LOC

7.2 Ease of Adoption

Methodology. We evaluated the ease of adoption of our design by measuring how much effort developers need to take to add TrustZone support to protect apps' existing UI and to conduct a system update. We quantify the effort as Line of code (LOC) added to use our solution, time spent on making the change, and success rate. We conducted the application evaluation using both open- and closed-source apps. We downloaded the code of open-source apps from F-Droid [2] and the closed-source apps from Google Play. We manually identify the UI risks (e.g., screenshot attack, keylog attack, etc) from the collected apps. Then we applied truz-views to eliminate the UI risks. For open-source apps, we count the LOC changes and modification time in order to integrate our solution into apps. For closed-source apps, we count the success rate of our integration. We initially implemented our solution on Android 7.0 and recorded the time to incorporated our solution on Android 9.0.

We need to solve one engineering challenge in order to evaluate our system on closed-source apps. Our design is mainly designed for open-source applications where we can label the view as TrustZone-enabled. To overcome the limitation that we cannot label views as TrustZone-enabled for closed-source apps, we customized the view system just for the purpose of evaluation. Our main idea is that each view object has a unique resource ID during runtime. We provide a configure file for Android view system. We can label the closed-source app's view by adding the resource ID of view into the configuration file. Our evaluation methodology on closed-source apps does not need to repackage the binary files and thus avoids all types of app crash caused by repackaging.

For use cases, we let apps display various types of sensitive data in the secure world by using our ImageView and TextView. Apps can further use our LockPatternView to display the data identity or authenticate users. We allow apps to secure the users' input in the secure world using our KeyboardView and SignatureView, and to confirm an important action inside the secure world using our Confirmation Dialog and Confirmation Activity.

Result. We totally modified 14 open-source apps, and results are shown in Table 4. It takes fewer than 5 minutes to modify all apps. Most apps take 2 LOC. One of the 2 LOC labels the view as TrustZone-enabled and the other LOC provides the reference to the TEE-protected data. The Confirmation Activity depends on how many views are needed to display in the secure world. The dialog has a fixed pattern, which is two buttons (i.e., positive and negative button) and a TextView for the confirmation message. Both Dialog and Confirmation Activity need 1 LOC to extract the attestation result from the `MotionEvent`.

Table 4: Evaluation Results for Open-Source Apps

App name	Test Case	LOC	Time (min)
Bitcoin wallet	ImageView	2	2
Bitcoin wallet	LockPatternView	2	2
Loyalty card	ImageView	2	3
Loyalty card	LockPatternView	2	3
andOPT	TextView	2	2
NoteCrypt	TextView	2	2
Signal	KeyboardView	1	1
Telegram	KeyboardView	1	1
Android-Signaturepad	SignatureView	2	2
Signatureview	SignatureView	2	2
UPM	Dialog	2	3
NoteCrypt	Dialog	2	3
Peanut Encryption	Confirmation	6	5
Note Buddy	Confirmation	6	5
Keypass DX	LockPatternView	2	2
Sealnote	LockPatternView	2	2

We collected 42 apps, including Chase, WeChat, Facebook, Linkedin, Instagram, Twitter, Alipay, etc. We used 10 apps for each use cases. Our result is shown in Table 5. All experiments were successful.

We migrated our design from Android 7.0 to 9.0. It took 6 hours to migrate our design to the new view system. Our system update is quick because our design of the view system follows its existing workflow.

7.3 Performance

In this section, we present results of performance evaluation for each major component in our system.

View system overhead. Our design modifies the view system in the Android graphics stack. We measure the performance overhead introduced to the Android graphics stack by running the benchmark tool called `Basemark OS`, which runs a series of test cases and provides a score report. We conducted the benchmark ten times, with a reboot to remove the impact caused by other factors, and then calculated the average score. As shown in Table 6, the major impact is caused by our modification in Android view system and by the memory access of framebuffer.

Input event latency. We calculated the touch input overhead by measuring the additional logic added to the `onTouchEvent()` in the view system. Because the logic added to each view is almost the same, we used the ImageView to measure the touch input overhead. The touch response overhead is less than 1 ms.

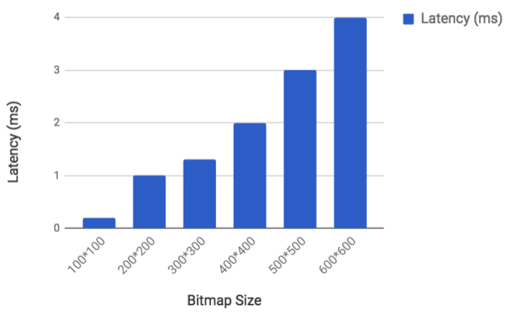


Figure 13: Image Rendering Overhead

Secure-world rendering overhead. Our image-operation TA renders content on the secure-world framebuffer. We measured the performance of the bitmap operations for both image and text in OPTEE OS [22]. We measured the time needed to construct the final framebuffer for various image sizes, word lengths, and font sizes. Figure 13 shows the image rendering overhead with common image sizes from 100*100 to 600*600. The overhead is less than 4 ms. Figure 14 shows the text rendering overhead with different lengths from 100 to 300 and different font sizes from 10 to 30. The overhead is less than 5 ms for the largest combination. Our rendering performance evaluation shows that our bitmap rendering approach is feasible to the real TrustZone platform with a low-performance cost.

Framebuffer sharing overhead. We measured the overhead of copying the normal-world framebuffer to the secure world. Because we introduce a separate Raspberry Pi to control screen, the actual

Table 5: Evaluation Result for Closed-Source Apps

Test Case	EditText	ImageView	TextView	SignaturePadView	Dialog/Confirmation	LockPatternView
Success/Total	10/10	10/10	10/10	10/10	10/10	10/10

Table 6: Framebuffer Transfer Performance Overhead

Benchmark	Origin	Modified	Overhead
System	1506	1499	0.4%
Graphics	306	302	1.4%

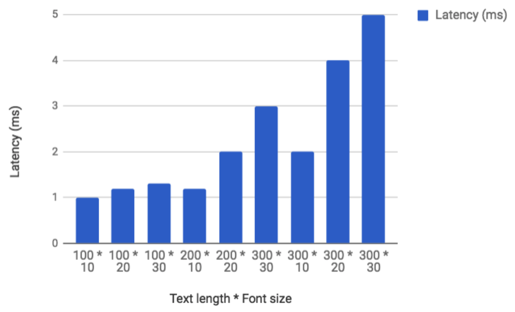


Figure 14: Text Rendering Overhead

Table 7: Framebuffer Transfer Performance Overhead

Data Size	TCP (second) for Emulation	SHM (second) for Real System
2MB	2.28	0.013
4MB	4.45	0.025
8MB	8.76	0.046

framebuffer memory sharing is over TCP (from the normal world to the Raspberry Pi). We also measured the OPTEE shared memory to transfer various sizes of framebuffer (2 MB - 8 MB). Table 7 shows both our framebuffer transferring overhead and the projected overhead. We argue that when vendors adopt our solution, the extra cost introduced by the Raspberry Pi can be reduced easily and replaced with the projected overhead because all vendors have drivers to their screen hardware, but rare research groups can obtain the access to drivers to a particular model of SOC.

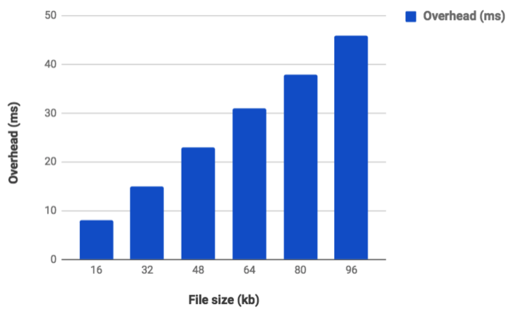


Figure 15: Data Download Performance Overhead

Download data overhead. We measured the overhead of our secure downloading feature. The overhead (average of 20 trails) is calculated based on the downloading time of various file size (16KB - 96KB). To eliminate the overhead caused by the Split SSL solution, we only calculated the overhead introduced by the `SSL_read()` and excluded the time for the TLS handshake. To eliminate the fact of the network bandwidth, we used the following way to calculate the overhead:

overhead = TZ download time - normal download time

Figure 15 summarizes our performance result. The main overhead is caused by the world switch of each TLS record decryption.

8 DISCUSSION

Limitation. Our current approach cannot support dynamic UI features, such as animation, scroll up/down, a timer, and touch event propagation, in the secure world. Our design is mainly used for static UI, which requires the TrustZone assistance. However, most of the security-related tasks do not involve these dynamic UI features (e.g., animation, timer, etc). If users need to use the dynamic UI features, users can interact with the same app's UI (without the protected data on the UI) in the normal world. We consider that the security benefits of our design are worth the cost of these UI dynamic features in the secure world.

9 RELATED WORK

In this section, we compare our work with other TrustZone UI solutions. Our UI solution can benefit all research works that build on top the TrustZone UI solution. We also list present Android UI security-related research.

TrustZone UI solutions. TrustZone UI solutions provide applications a development platform to build secure-world UI. Our work fits into this category. Several TrustZone UI solutions [4, 20, 30] follow a self-contained model. These solutions require a large TCB in the secure world. Another category of the solution [11] leverages the normal-world hypervisor protection. Our work is different from ShrodinText [11] in two aspects: (1) our threat model is stronger. ShrodinText requires a trusted hypervisor in the normal world to secure screen. We do not trust any normal-world components including hypervisor since hypervisor can be compromised by the untrusted OS [9, 25]; (2) our solution has a comprehensive UI coverage. Our design offers a generic UI solution to protect both UI input and UI display while ShrodinText mainly focuses on a single use case of UI display (i.e., display text). It is difficult to generalize ShrodinText design to protect both UI input and output like what we do.

TrustZone UI applications. Several research works identified the security UI risks and develop various solutions on top of the TrustZone UI. Li et al. [17, 18] built on top of T6 [4] and improve the integrity of mobile UI. Samsung KNOX [27] built on top of Trustonic [30] and protected the confidentiality and integrity of the UI interaction. TruZ-UI [32] provided generic secure-world UI and bound with the normal-world application code. TrustOTP [28] leveraged TrustZone UI to protect the one-time password display.

TrustPay [35] proposed a mobile payment framework on the TrustZone platform to protect the display of the user's payment information. IM-Visor [29] and Li et al. [19] protected the users' inputs by capturing the users' sensitive keystrokes inside the secure world. Dmitrienko et al. [14] proposed a security architecture for the protection of electronic health records and authentication credentials used to access e-health services. AEP-M [31] adapted TrustZone to protect users' money and critical data during the e-payment process. Our research benefits all these works that built on top of the TrustZone UI solution and provides an easy-to-use UI solution for them.

Android UI Security. Researchers have discovered various vulnerabilities [12, 13, 15, 16, 24, 34] in Android UI system and proposed several solutions [5, 12, 23, 26, 33] to mitigate the problems. Different from TEE researches, Android UI researches assume that the Android OS is robust and the malware wants to gain unauthorized access to the UI. Researchers discovered several UI task hijacking techniques [13, 15, 24, 34] to render phishing UI. Antonio et al. [12] and WindowGuard [23] proposed solutions to mitigate the UI task hijacking. Yanick et al. [15] and Luo et al. [21] discovered touch jacking in the Android UI. Android has been aware of the screenshot attack and has allowed developers to set a secure window to prevent the app's UI from being taken screenshots [5]. However, the Android secure window cannot protect the UI in the event of a compromised OS.

10 SUMMARY

In this paper, we proposed a TrustZone UI design model called *delegation model* and systematically studied the design properties of the delegation model. Based on our new model, we developed truz-views that includes confidential display, confidential input, and integrity-preserved interaction. We implemented our design on the HiKey board and evaluated our system using real-world apps. The evaluation results show that our solutions can be adopted easily by existing apps with a low-performance overhead.

11 ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their insightful comments. This project was supported in part by the NSF grant 1718086. We also thank Xiaoyun Shen-Krizic for her detailed editing suggestion.

REFERENCES

[1] 2017. Android Developers: Selecting Devices. https://source.android.com/source/devices.html. (2017).
[2] 2017. F-Droid repository. https://f-droid.org/en/packages/. (2017).
[3] 2017. Samsung Pay. http://www.samsung.com/us/samsung-pay/. (2017).
[4] 2017. TrustKernel T6 Secure OS. https://www.trustkernel.com/en/products/tee/t6.html. (2017).
[5] 2018. Android WindowManager FLAG SECURE. https://developer.android.com/reference/android/view/WindowManager.LayoutParams. (2018).
[6] 2018. Google Android: Vulnerability Statistics. http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224. (2018).
[7] 2018. MCUFont. https://github.com/mcufont/mcufont. (2018).
[8] 2018. Tk graphical user interface toolkit. https://www.tcl.tk/. (2018).
[9] 2018. XEN security vulnerability. https://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html. (2018).
[10] 2018. Xlib X Language X Interface. https://www.x.org/archive/X11R7.5/doc/libX11/libX11.html. (2018).
[11] Ardalan Amiri Sani. 2017. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*. ACM, New York, NY, USA, 197–210. https://doi.org/10.1145/3081333.3081346
[12] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. (2015).
[13] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*.
[14] Alexandra Dmitrienko, Zecir Hadzic, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. 2013. Securing the Access to Electronic Health Records on Mobile Phones. In *Biomedical Engineering Systems and Technologies*.
[15] Yanick Fratantonio, Chenxiong Qian, Simon Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.
[16] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. 2017. Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.
[17] W. Li, H. Li, H. Chen, and Y. Xia. 2015. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. NY, USA.
[18] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. [n. d.]. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '18)*.
[19] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C. Chu, and T. Li. 2014. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*. Beijing, China.
[20] D. Liu and L. Cox. 2014. VeriUI: Attested Login for Mobile Devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. CA, USA.
[21] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. 2012. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Proceedings of the 5th International Symposium on Foundations & Practice of Security*.
[22] OP-TEE. 2015. OPTEE OS. (2015). https://github.com/OP-TEE/optee_os
[23] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *NDSS*.
[24] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *24th USENIX Security Symposium (USENIX Security 15)*.
[25] B. Robert, V. Julian, and N. Jan. 2016. The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture. In *Information and Communications Security*.
[26] Franziska Roesner and Tadayoshi Kohno. 2013. Securing Embedded User Interfaces: Android and Beyond. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX.
[27] Samsung. 2013. KNOX White Paper. (2013).
[28] H. Sun, K. Sun, Y. Wang, and J. Jing. 2015. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*. Denver, Colorado, USA.
[29] Chen Tian, Yazhe Wang, Peng Liu, Qihui Zhou, Chengyi Zhang, and Zhen Xu. 2017. IM-Visor: A Pre-IME Guard to Prevent IME Apps from Stealing Sensitive Keystrokes Using TrustZone. *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2017).
[30] Trustonic. 2012. Trustonic TEE Trusted User Interface. (2012).
[31] Bo Yang, Kang Yang, Zhenfeng Zhang, Yu Qin, and Dengguo Feng. 2016. AEP-M: Practical Anonymous E-Payment for Mobile Devices Using ARM TrustZone and Divisible E-Cash. In *Information Security*.
[32] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. 2018. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services* (June 10-15) *(MobiSys '18)*. Munich, Germany.
[33] Xiao Zhang, Yousra Aafer, Kailiang Ying, and Wenliang Du. 2016. Hey, You, Get Off of My Image: Detecting Data Residue in Android Images. In *Computer Security – ESORICS 2016*.
[34] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. 2016. Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android. In *NDSS*.
[35] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. 2016. TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms.. In *ISCC*.