May 2019

Integrating TrustZone Protection with Communication Paths for Mobile Operating System

Kailiang Ying
Syracuse University

ABSTRACT

Nowadays, users perform various essential activities through their smartphones, including mobile payment and financial transaction. Therefore, users' sensitive data processed by smartphones will be at risk if underlying mobile OSes are compromised. A technology called `Trusted Execution Environment` (TEE) has been introduced to protect sensitive data in the event of compromised OS and hypervisor.

This dissertation points out the limitations of the current design model of mobile TEE, which has a low adoption rate among application developers and has a large size of Trusted Computing Base (TCB). It proposes a new design model for mobile TEE to increase the TEE adoption rate and to decrease the size of TCB. This dissertation applies a new model to protect mobile communication paths in the Android platform. Evaluations are performed to demonstrate the effectiveness of the proposed design model.

INTEGRATING TRUSTZONE PROTECTION WITH COMMUNICATION PATHS FOR MOBILE OPERATING SYSTEM

by

Kailiang Ying

M.S., Syracuse University, December 2013

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer and Information Science and Engineering.

Syracuse University

May 2019

ACKNOWLEDGMENTS

The work presented in this thesis could not have been created without the encouragement and guidance provided by many others. I would like to acknowledge those who have helped me throughout this effort.

First, I would like to express my sincere gratitude to my advisor, Prof. Wenliang Du for all guidance and support during my PhD study. His expertise has taught me the art of finding "good" research problems to think about. He has also taught me more than I could ever imagine about the fundamentals of research and technical presentation. All of these aspects will serve me very well in the years to come, and I am very grateful that he decided to take a chance on working with the random graduate student that was taking his class and asked to learn more about his research work.

While my advisor has been my primary source of guidance during my time at Syracuse University, he is one of many faculty members that have given me an opportunity to gain both breadth and depth of knowledge during my doctoral studies. I appreciate Prof. Yuzhe Tang, Prof. C.Y. Roger Chen, Prof. Jae C. Oh, Prof. Vir V. Phoha, and Prof. Young B. Moon for agreeing to be on my thesis committee. I was also fortunate to have an opportunity to collaborate with Prof. Heng Yin. I am grateful for their time and efforts to help me succeed in my PhD dissertation.

The help and friendship of my colleagues, has made the long doctoral process much more enjoyable. I am very fortunate to cooperate with several colleagues, including Xing

Jin, Dr. Xiao Zhang, Dr. Xunchao Hu, Dr. Yousra Aafer, Amit Ahlawat, Priyank Thavai, Bilal Alsharifi, Yuexin Jiang, Zhenshen Qiu, and Gautam Nagesh Peri. Special thanks to the guidance of Dr. Tongbo Luo, Yifei Wang, Dr. Paul Ratazzi, and Dr. Andrew Henderson and the company of Haichao Zhang, Hanyi Li, Francis Akowuah, Hao Hao, Zhenyu Wang, Amey Ashok Patil, Curtis Robinson, and others.

None of this would have been possible without love and support from my parents. They are always encouraging me to pursue my own dream, which has led to this great journey of my PhD study.

Finally and most importantly, I am proud to acknowledge my wife Minqi Cai for all support and encouragement that she has given me during my doctoral studies. Without her help to make the doctoral process more tenable, I doubt I would have made it through. For this, I will forever be thankful. I look forward to all long walks and trips to the park that we will have in our future.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Nowadays, users perform various essential activities through their smartphones, including mobile payment and financial transaction. A Pew Research Center's survey in 2018 showed that 77% of adults in the United State have smartphones, particularly among the 18-29 year-old and the 30-49 year-old. Nearly 94% of the former have smartphones and 89% of the latter do [28]. Smartphone thus has become an essential medium for users to communicate with the public. The Pew Research Center also points out that 57% of adults in the United States regularly conduct financial transaction using their smartphones [27]. In China, 47% of online transactions were completed on smartphones in 2018 [30].

1.1 Mobile Communication Path and its Risks

As an important bridge between a user and the public, the mobile provides various communication paths for interacting with users as well as for helping users to communicate with the public. During these communications, users' sensitive data such as passwords and payment information are passed through these communication paths. This dissertation summarizes these communications as the *communication path model*, as illustrated in Figure 1.1.

There are two types of communication paths. The first type is between mobile devices and individual users, in which users send instructions to, and receive visual or audible

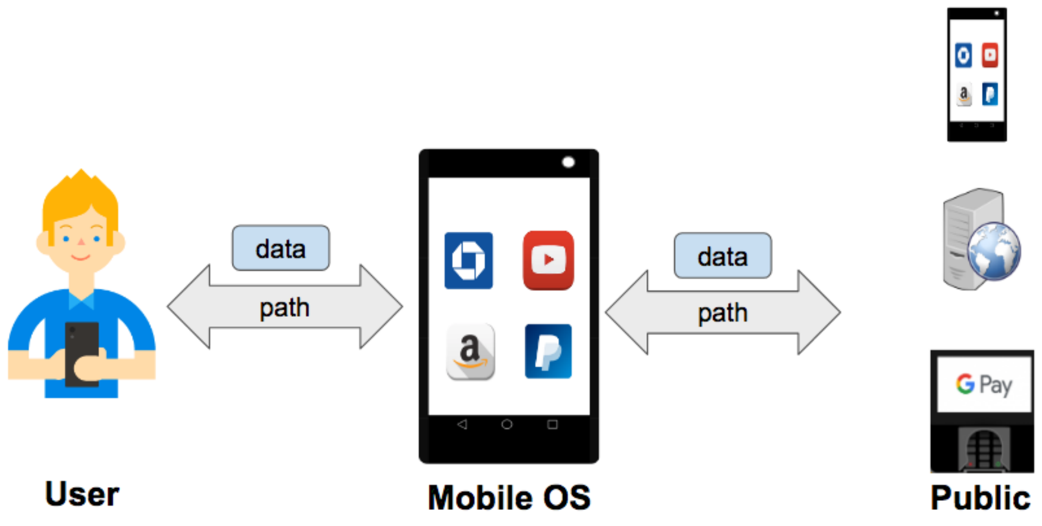


Fig. 1.1.: Mobile Communication Path Model

feedback from mobile system. One example is user's interaction with the User Interface (UI). Other instructions like voice or facial instructions are invented with the evolution of mobile technologies [23, 24]. The second type is the communication path between mobile devices and the public. The current technology provides various communication paths for the mobile to talk to the public. For instance, network can help mobile devices link a wide variety of internet services such as online banking, health services, payment transaction, etc. Users can also connect to other physical devices such as an additional smartphone or a vending machine through built-in Bluetooth, NFC on their smartphones.

Mobile technology safeguards both types of communication paths in the following ways. Mobile OS isolates all applications that communicate through the paths. No application can access the data stored by other applications. Mobile hardware further introduces a virtualization technique called hypervisor, which has a higher privilege than a mobile OS to secure these communication paths in case malware bypasses the OS-level access control.

Unfortunately, the CVE reports show that the number of vulnerabilities in mobile OSes and hypervisors has been increasing over the years [20, 33, 78]. Take the Android OS as an example – the number of vulnerabilities has skyrocketed over the last eight years. 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 OSes. Researchers also report several vulnerabilities inside hypervisors [33, 78]. Once mobile OSes and hypervisors are compromised, the last defense of mobile communication paths is gone, and the data passing through the communication paths will then be controlled by malware.

1.2 Trusted Execution Environment

Challenged by potential risks mentioned above, a technology called *Trusted Execution Environment* (TEE) has been introduced to help protect users' data on mobile communication paths in the event of compromised OS and hypervisor. The most commonly deployed TEE on mobile devices is *ARM TrustZone*. TEE is also supported by other hardware technologies including AMD Platform Security Processor, Apple Secure Enclave, and Intel Software Guard Extensions (SGX). TrustZone builds a TEE on the device (called the *secure world*), isolating itself from the *normal world* where normal apps are installed. The secure world runs a separate OS called *TEE OS* that can provide confidentiality and integrity needed to protect users' intended activities. TEE is now being utilized in many scenarios. For instance, Samsung pay's *Trusted Application* (TA) [16] built on top of the Trustonic OS [92] protects users' payment tokens in the secure world. Bitcoin Ledger [4] allows users to confirm a bitcoin transaction inside the secure world; otherwise, the transaction is not accepted by the server.

This dissertation categorizes the existing TrustZone related works into two groups: (1) TrustZone OS related work and (2) TrustZone integration related work (see Figure 1.2).

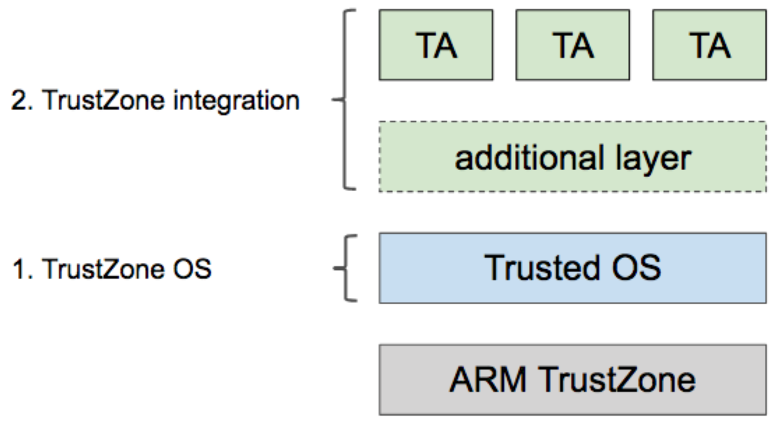


Fig. 1.2.: Existing works categorization

TrustZone OSes [17, 75, 92] provide a TEE as a companion to the rich OS running on the same ARM hardware. As shown in Figure 1.3, the main components of TrustZone OS include the followings: (1) a secure monitor that monitors the transition between the normal world and secure world; (2) a TEE core that executes all the privileged level ARM instructions; (3) a collection of TEE libraries designed to be used by Trusted Applications; and (4) a HAL layer to drive the hardware. Secure monitor and TEE core are two unique components for a TEE OS to utilize TrustZone hardware features. To support different logic in an isolated running environment for TAs, TEE OSes have to incorporate many libraries. Over the last decade, the size of the TEE has increased significantly due to the need of TrustZone OSes to support various functionalities for TAs. Therefore, TEE currently relies on complex and oversized Trusted Computing Base (TCB). However, such a design direction has violated a critical TEE principle: having a small TCB that can be more easily verified in comparison to a rich OS.

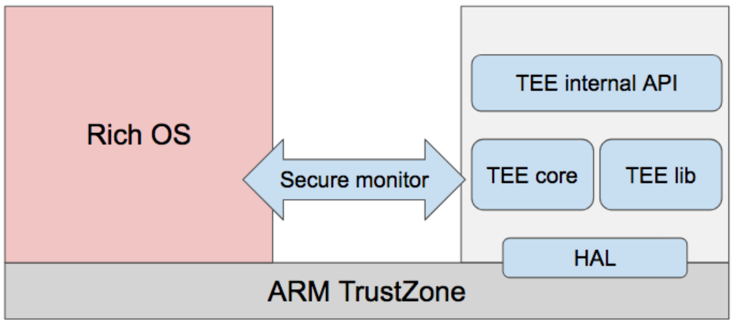


Fig. 1.3.: TEE OS Components

The second group of TrustZone related works focuses on TrustZone integration. Until now, TrustZone integration solutions [63, 65, 83, 88] have successfully built various protection mechanisms (i.e., TAs) into TrustZone OSes [17, 75, 92] to support secure payment, one-time password generation, confirmation attestation, etc. For instance, in its Trustonic [92], Samsung KNOX [83] built a payment TA to protect transactions and payment information in the secure world. TrustOTP [88] has leveraged TrustZone UI to protect the generation and the display of a one-time password. Li et al. [63, 65] has built attestation TAs on top of T6 [17] to ensure the integrity of mobile UI.

TrustZone integration, though can strengthen security of mobile applications, has only been adopted by developers who are either TEE vendors or partners. Unlike vendor-developed TAs that already reside in the secure world, third-party applications and data must be installed remotely because the devices are owned by individual customers.

The process to load data securely therefore poses unique requirements. An identity of a unique device must be embedded to enable remote attestation that the device is genuine before the installation of the application in the secure world. The application developers have to work closely with device vendors in order to integrate TrustZone. Such a complex integration process has restricted many third-party applications from getting the benefits of TrustZone.

There is a severe security issue associated with the current TEE integration model. It allows application-specific code to run in the secure world. Running application logic is exactly what has broadened the normal-world attack surface. This dissertation intends to enable all third-party applications to use TEE. If third-party application developers follow the existing TEE integration model, the number of TEE vulnerability will be significantly increased. An ideal TEE integration model would require no application logic in the secure world and would provide generic security mechanisms for mobile applications to enhance their security.

1.3 Thesis statement and contributions

The thesis statement of this dissertation is that, **design easy-to-use solutions with small TCB's size for mobile applications to use TrustZone without including their application-specific code in the TEE.** In support of this statement, this dissertation describes the following contributions:

1. **Delegation TEE Model**: To design easy-to-use and small TCB sized TrustZone TEE solutions, this dissertation proposes a novel design model of TrustZone TEE

called *Delegation TEE Model*. To be specific, the design integrates the TrustZone protection at the normal-world OS level and leaves the generic protection mechanism in the secure world. The delegation model conducts all non-sensitive operations in the normal world and allows application developers to use the normal-world OS interfaces to integrate with the TrustZone protection. This dissertation systematically compares the design trade-off between the delegation and traditional design model. Moreover, it derives the design principles and points out the potential challenges when the delegation TEE model is applied to design protection solutions for the mobile-user and mobile-server communication paths.

2. **Split SSL and TruZ-HTTP**: Users' sensitive data need to be transferred between the mobile and the server through the network communication path. Confidential information should not be revealed to the normal world, but they need to be sent to servers. Given that the normal-world OS is untrusted, this dissertation applies the delegation TEE model to help applications transfer the TrustZone-protected data between the authorized server and the mobile without the data disclosed to the normal world. This dissertation presents the design and implementation of *TruZ-HTTP* and *Split SSL*, an easy-to-use TrustZone-server secure communication mechanism with a slight increase in TCB's size. The main idea is to integrate the TrustZone protection with the standard HTTPS and to move only the protection mechanism, the SSL crypto, into the TrustZone. The design reuses the normal-world network stack for the communication with servers. The prototype is built using the TrustZone-enabled HiKey board to evaluate the design and various

new security features on existing applications to protect users' sensitive information. The real-world use case evaluation shows that application can leverage the new design with few changes to their existing applications. The usability study proves that users can interact with the system to protect their security sensitive activities and data in a correct way.

3. **TruZ-View**: The present TEE UI solutions provide a fully functional UI stack in the TEE, but they fail to manage one critical design principle of TEE: a small TCB should be more easily verified in comparison to a rich OS. The TCB's size of the current TEE UI model is large due to the size of an individual UI stack. To reduce the TCB's size, this dissertation builds a novel TEE UI solution called *TruZ-View* based on the delegation TEE model. To be specific, the design reuses the majority of the rich OS UI stack. Unlike the existing UI solutions protecting 3-dimensional UI processing in the TEE, the design protects the UI solely as a 2-dimensional surface and thus reduces the TCB's size. The new UI solution 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. The design is implemented on HiKey board and several TEE UI use cases are developed to protect the confidentiality and integrity of UI. A thorough security analysis is performed to prove the security of the delegation UI model. The evaluation of the real-world applications shows that developers can leverage the TEE UI with few changes to the existing app's UI logic.

1.4 Organization of Dissertation

Chapter 2 provides an overview of TEE technology, architecture of mobile-server communication path, and the Android User Interface. Chapter 3 reviews related literature on TEE solutions and Android security in general. Chapter 4 presents a new design model of TrustZone TEE called *delegation TEE model*. Chapter 5 proposes TEE solution *Split SSL* and *TruZ-HTTP* that can secure mobile-server communication path. Chapter 6 presents another TEE solution to secure the mobile-user communication path called *TruZ-View*. The last chapter summarizes this dissertation.

2. BACKGROUND

2.1 ARM TrustZone Tutorial

The TrustZone technology is a system-wide approach to security that allows the building of secure endpoints with a root of trust. TrustZone-based systems provide security by partitioning the System-on-Chip's (SoC) hardware and software resources such that they exist in one of two hardware-separated worlds, the *secure world* for a security subsystem and the *normal world* for everything else. Figure 2.1 shows the isolation between two worlds, with the normal world being Android. For a SoC using TrustZone, the boot sequence involves a chain of trust. The secure-world OS boots before the normal-world OS and the boot sequence includes cryptographic checks to each stage of the boot process of the secure world. The normal-world software is not allowed to access the secure-world resources. The normal- and secure-world concepts are applied to various parts of the SoC, including memory, software, bus transactions, interrupts, and peripherals. Two worlds are partitioned using the hardware logic implemented in the bus fabric, peripherals, and processors. Each physical processor core executes two virtual cores, one considered secure and the other considered non-secure. The two virtual processors execute in a time-sliced fashion and switch based on a control signal, NS bit. The secure-world implementation includes various software components: trusted boot, a secure-world switch monitor, a small trusted OS and trusted apps (or trustlets). There are several trusted OSes currently in development, including OPTEE [75], T6 [17], trustonic [92], etc. ARM provides a reference implementation of secure-world software known as ARM Trusted Firmware, which includes trusted boot and a secure runtime that takes care of the switching between the normal- and secure-worlds using Secure Monitor Code Calling Convention (SMCCC).

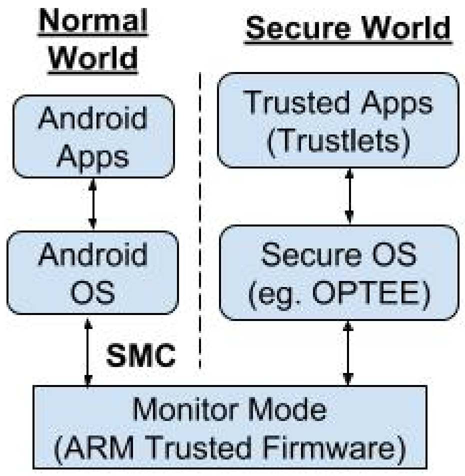


Fig. 2.1.: Normal and Secure Worlds

OPTEE is an open-source TEE OS maintained by Linaro, based on the GlobalPlatform TEE system architecture specification. It is designed to be compatible with any isolation technology suitable for TEEs, including TrustZone. In TrustZone, the OPTEE OS kernel allows `trusted applications` (`TAs`) to run in the user space. A TA provides a set of commands, each of which is a function that can be invoked by the normal world. The OPTEE kernel forwards the normal-world request to a TA and returns the result to the normal world.

Secure Boot As shown in Figure 2.2 [1], after the SoC is powered-on, a ROM-based bootloader is executed which initializes critical peripherals. It then invokes the device bootloader located in flash memory. The boot sequence then proceeds through the secure-world OS initialization stages. Once the process is completed, the control is passed to the normal-world bootloader. This starts the normal-world OS, at which point the system is considered running. The secure boot sequence includes cryptographic checks to each stage of the boot process of the secure world. It aims to assert the integrity of the secure-world software, preventing any unauthorized or maliciously modified software from running. It provides isolation from the normal-world OS (Linux/Android) and uses underlying hardware support to protect trusted apps. It is designed to have a small footprint, such that code and data required to provide isolation can reside in a small amount of on-chip memory. It is portable and can be used with different architectures and hardware, and it provides support for various setups such as multiple TEEs or multiple client OSes.

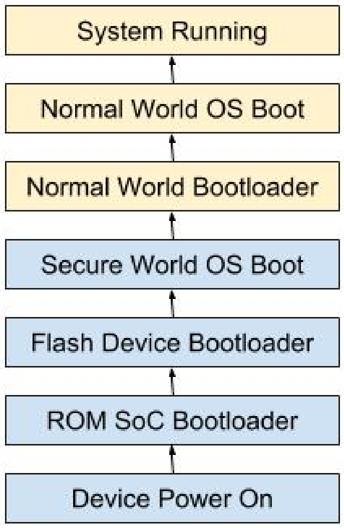


Fig. 2.2.: Secure Boot

2.1.1 ARM TrustZone vs Intel SGX

Given that in addition to ARM TrustZone, there exists another popular TEE platform called Intel Software Guard Extensions (SGX), this dissertation intends to highlight the differences between TrustZone and SGX to demonstrate why SGX is not a suitable platform to tackle the problem that we intend to solve. Like ARM TrustZone, Intel SGX is a trusted execution environment technology. It provides a set of new CPU instructions that allows applications to create secure regions (called `enclaves`) for code and data. Such regions are protected from untrusted host environments where the enclaves reside. The protection is achieved via memory encryption that can only be decrypted by the CPU, making the memory content non-interpretable by the untrusted OS. Common SGX applications include secure containers used for executing trusted client code in untrusted cloud environments [37, 39, 56, 86].

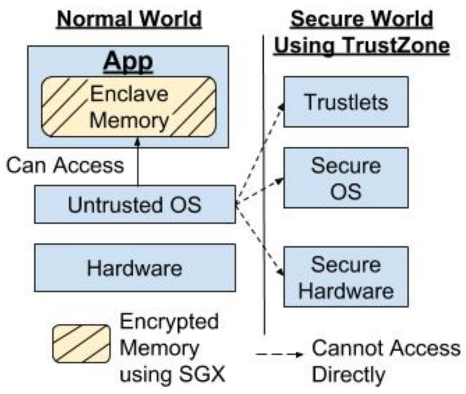


Fig. 2.3.: SGX versus TrustZone

This dissertation intends to allow apps to securely obtain inputs from users. To achieve this, the path from user to the I/O devices must be trusted. As illustrated in Figure 2.3, for Intel SGX, only the CPU is considered trusted, while the rest of the hardware, the host

OS, and the operator of the hardware are considered untrusted. Since I/O operations need to go through the untrusted OS, it is difficult, if possible, to secure users' inputs through these I/O channels. In contrast, for TrustZone, whose common deployment is on personal mobile devices, the operating system running inside the TEE and users are both considered trusted. Moreover, TrustZone can have exclusive control of I/O devices. Therefore, I/O channels inside TrustZone can be secured. With the assistance of TrustZone, users obtain the capability of protecting the confidentiality and integrity of their inputs.

2.2 Transport Layer Security (TLS)

Nowadays more and more users' data are transmitted through the internet because of the frequent communication between mobile applications and their backend servers. However, when data are transmitted over such an unprotected public network, they can be read or even modified by others. Mobile applications are wary of the security of mobile-server communication. The cryptographic solution can achieve the goal. TLS, Transport Layer Security, is a standard network protocol for applications to protect their data.

This section discusses how TLS works and particularly focuses on two most important aspects of TLS that are TLS handshake and TLS record. The TLS handshake is used to establish a secure channel and the TLS record is the transmit data inside the secure channel. The secure channel provided by TLS has the following three properties:

- *Confidentiality*: Nobody except two ends of the channel can see the data via the channel.
- *Integrity*: If data are tempered by others during the transmission, the receiver can detect them.
- *Authentication*: It can be assured that one end is communicating to the intended host. Without the proper authentication, the client may unknowingly establish a communication path with an attacker.

TLS sits between the Application Layer and the Transport Layer, as Figure 2.4 shows. Application layer protocols can send their unprotected data to the TLS layer. TLS handles the encryption, performs the decryption, and completes the integrity check. The cryptographically protected data is given to the underlying Transport layer for data transmission.

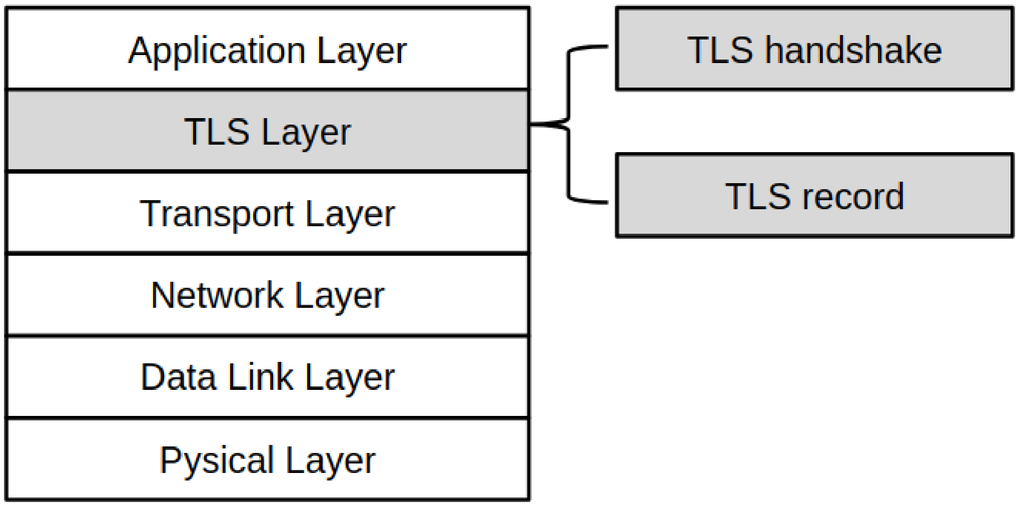


Fig. 2.4.: TCP/IP network stack with the TLS layer

2.2.1 TLS Handshake

To set up a secure channel between the client and the server, several things are needed to be agreed upon between two ends, including the key exchange algorithm, the encryption algorithm, the encryption keys, etc. These cryptographic parameters are needed to be set up during the TLS handshake. The discussion is based on TLS version 1.2 [32].

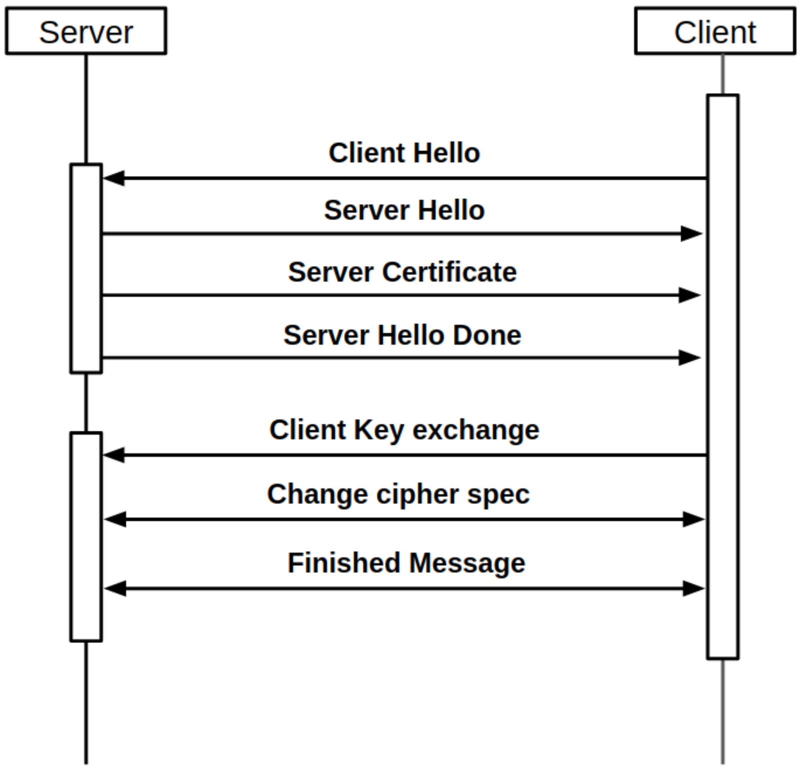


Fig. 2.5.: TLS handshake protocol

Figure 2.5 illustrates the steps of the TLS handshake protocol. Details are further explained in the following:

- Client: The client initiates the TLS connection by sending the *Client Hello* message. The message tells the server which cipher suites are supported by the client. The client also exchanges a client random number with the server for key generation.
- Server: The server replies with a *Server Hello* message. The server selects a cipher suite from the *Client Hello* message and replies to the client. The server also exchanges a server random number with the client for key generation.
- Server: The server sends its certificate, which includes a public key, a signature, and a server common name, to the client. The client verifies the certificate to check the authenticity of the server.
- Server: The server sends a *Server Hello Done* message to the client.
- Client: The client verifies the server certificate and generates the premaster secret, the master secret, the cipher key block, and session keys. The premaster secret is encrypted using the server public key and is sent with the *Client Key Exchange* message.
- Client and Server: Both ends send a *Change Cipher Spec* message to indicate that further communication will be authenticated and encrypted.
- Client and Server: The client sends the first encrypted message called *Finished* message to the server. This message indicates that the secure channel is established. This message hashes all previous handshake messages and the server needs to verify the MAC. The server does the same action by sending a Finished message to the client.

2.2.2 TLS Record

Once a client and a server finish TLS handshake protocol, they can start exchanging application data using the TLS Record protocol. Each record contains a header and a payload, as shown in Figure 2.6.

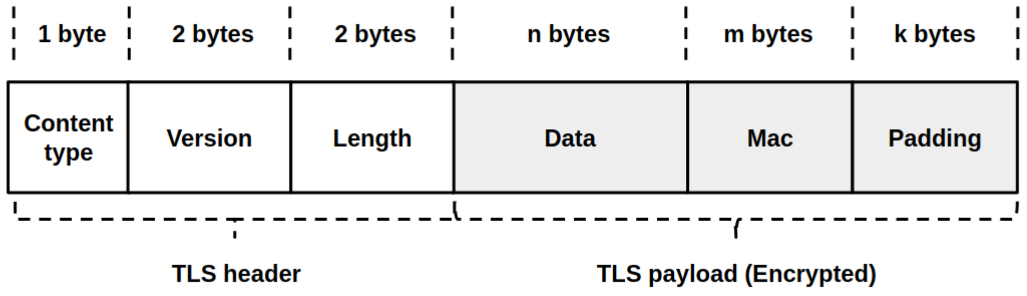


Fig. 2.6.: TLS Record

The record can also be used to send the other type of message like the handshake message. This section focuses on explaining the application data (the content type value is 0x17). The length of the payload cannot exceed 16KB due to the limit of the length field.

Sending Data with TLS Record Protocol After the secure channel is established, both ends can invoke the `SSL_write()` API to send data through the secure channel. The TLS record layer breaks the data into multiple blocks (TLS record size) and generates a MAC for each block. TLS then encrypts each block and creates TLS records. The record is given to the underlying transport layer for transmission.

Receiving Data with TLS Record Protocol When receiving the data from the TLS channel, the application called the `SSL_read()` API to read the decrypted data. The TLS layer reads one or multiple records from the TCP stream, decrypts them, verifies their MAC, and decompresses the data, before giving the data to the application.

2.3 Android UI

The most frequently used communication path between the user and the mobile is the User Interface (UI). Because of the heavy reliance on this single interface, users display and input many sensitive data through the UI. This section provides the background information about the Android UI development process and the architecture of UI stack.

2.3.1 Develop Android UI

The Android app's UI is defined by the layout. Developers use OS predefined view elements to build the UI layout. The view elements are usually called ”widgets”, such as ImageView, TextView, etc. Developers design their UI layout using Android's XML vocabulary and add view elements into XML to gradually build a view hierarchy that defines the App's UI. The Listing 2.1 is an example of how developers define a button in the application layout.

Listing 2.1: Android UI Layout Example

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/
                                            apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
   <Button android:id="@+id/button_id"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="click this" />
</LinearLayout>
```

To customize the widgets' behavior, developers initialize the java view object and override the callback events (e.g., `onTouchEvent`, `setOnClickListener`) of the view. These view elements are the building blocks to build the App's UI. The Listing 2.2 is an example of how developers register click listener for a button on the application layout.

Listing 2.2: Android Register Button Click Listener

```
<?xml version="1.0" encoding="utf-8"?>
    final Button button = findViewById(R.id.button_id);
    button.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            // listen logic
        }
    });
```

2.3.2 Android UI Stack

Android UI stack is composed of various system services and components that require a close collaboration with them. `Activity Manager Service` (AMS) and `Window Manager Service` (WMS) are the most important ones.

Activity Activity is a type of Android component that users see. It contains one or more windows to the user. Each activity must have a window instance, which contains the content that the application wants to display. The application developers draw the UI through the SDK provided by the Android.

Window Window is a visual area on the screen. It holds the hierarchy of the UI elements that are displayed on the screen. In WMS, each window instance is represented by a `WindowState` instance that contains the window size, location, etc. Commonly, the

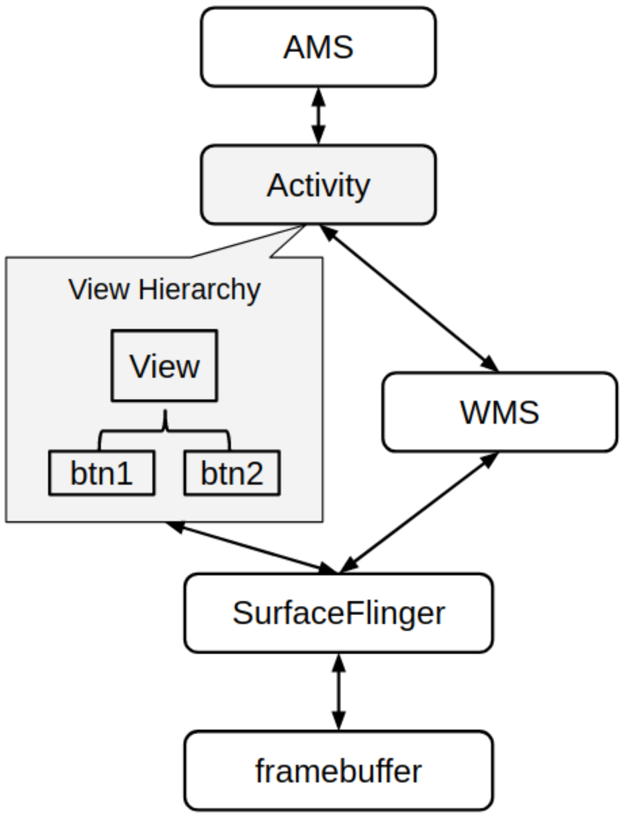


Fig. 2.7.: Android UI Stack Overview

system composes multiple window instances into one frame that is displayed on the screen.

Architecture overview Figure 2.7 is an overview of the Android UI stack. When an app comes to the foreground, the AMS launches a new activity of the app. The WMS then creates a window instance for the activity. On behalf of the app, WMS asks the `surfaceFlinger` to create a drawing surface for the window. The surface is a bitmap represented by RGBA (red, green, blue, alpha) values. The Android OS creates an IPC channel (i.e., binder) between an app and the `SurfaceFlinger`. The app can directly send app hierarchy of the UI elements to SurfaceFlinger. SurfaceFlinger accepts buffers of data from multiple resources (e.g., apps, system), composites them into a single frame called `framebuffer` and sends it to display. The framebuffer is a piece of memory that contains a bitmap that displays on the screen.

3. RELATED WORKS

This chapter discusses the existing works related to the areas of TEE research and Android privacy enhancement research. This dissertation proposes easy-to-use TEE solutions and has generic support for common application requirements. Moreover, it intends to maintain a small TCB for these TEE solutions. These features distinguish this dissertation from the most closely related works.

3.1 Trusted execution environment.

Rubinov et al. [80] automatically partitions the app sensitive logic into the TEE. DroidVault [66] establishes a secure channel for uploading and downloading sensitive data to/from the server by leveraging TrustZone. LightSPD [97] emulates a secure portable device in TrustZone to protect the users' privacy. TrustUI [64] enables secure user's interaction by leveraging TrustZone while maintaining a small TCB. TrustICE [89] and PrivateZone [59] load verified normal-world code into the TEE-protected memory and executed them in containers. Liu et al. [69] preserves the integrity of sensor readings by protecting peripherals using TrustZone. Keystore [3] and Fingerprint [2] are parts of Android's built-in TrustZone support but their threat model is different from the threat model in this dissertation. Keystore [3] only protects the keys in the secure world but the compromised normal world can still ask the secure world to decrypt the content using the key. Fingerprint [2] only protects the users' biometrics data but the compromised normal world can still spoof the fingerprint approval without users' consent. All these works [59, 64, 66, 69, 80, 89, 97] require application logic in the secure world. TrustOTP [88] integrates a hardware-based one-time password solution with TrustZone. AdAttester [65] uses TrustZone to provide attested click and display for android advertisements. SchrondinText [36] protects the text output of the applications while we focus on text input.

TEE researchers also cover a variety of research directions. TrustZone has an attack surface when using shared memory [71] during the communication between the two worlds, or via side channel like cache [54, 106]. TrustZone also suffers from physical memory forensics [45, 82]. CaSE [105] and CacheKit [104] enhance TrustZone's memory privacy against physical attack. One category of TrustZone research focuses on monitoring the integrity of normal-world memory [38, 58]. VTZ [55] virtualizes TrustZone in the VM.

SGX cannot protect users' input like TrustZone, because SGX does not have a separate OS to control the I/O peripherals (reason described in Section 2.1.1). SGX is mainly applied in cloud-based applications. SGX studies [37, 39, 84, 86] propose to protect the user's code and data in the enclave even when they are running in a hostile environment. OpenSGX [57] provides Intel SGX emulation platform to develop enclave programs in the emulator. SGX platform suffers from side-channel vulnerabilities [94, 96]. Various solutions [74, 85] have been proposed to solve the side-channel vulnerabilities in SGX.

3.2 TrustZone-protected UI

This section reviews existing TrustZone UI solutions and applications. The UI solution can benefit all applications built on top of the TrustZone UI solution.

TrustZone UI solutions. TrustZone UI solutions provide applications a development platform to build secure-world UI. TruZ-View [101] described in Chapter 6 fits into this category. Several TrustZone UI solutions [17, 67, 93] follow the rich OS design direction. These solutions require a large TCB in the secure world. Another category of the solution [36] leverages the normal-world hypervisor protection. SchrodinText [36] provides a small TCB solution for text rendering. The TCB of this dissertation is more robust than SchrodinText [36] because it leverages a trusted hypervisor in the normal world to secure the screen while the hypervisor can be compromised by the untrusted OS [33, 78]. This dissertation proposed TEE UI solution that maintains a small TCB in the secure world and does not trust any component in the normal world.

TrustZone UI applications. Several research works identified the UI security risks and developed various solutions on top of the TrustZone UI. Li et al. [63, 65] built on top of T6 [17] and improved the integrity of mobile UI. Samsung KNOX [83] has built on top of Trustonic [93] and protected the confidentiality and integrity of the UI interaction. TruZ-UI [100] provided generic secure-world UI and bound with the normal-world application code. TrustOTP [88] leveraged TrustZone UI to protect the one-time password display. TrustPay [109] proposed a mobile payment framework on the TrustZone platform to protect the display of users' payment information. IM-Visor [90] and Li et al. [64] protected users' inputs by capturing users' sensitive keystrokes inside the secure world.

LightSPD [97] emulated a reliable portable device in TrustZone to protect users' privacy. Dmitrienko et al. [48] proposed a security architecture for the protection of electronic health records and authentication credentials used to access e-health services. AEP-M [98] adapted TrustZone to protect users' money and critical data during the e-payment process. Marforio et al. [72] secured the binding between users and their TEE devices by providing enrollment protocols. This dissertation proposed a novel UI design that can benefit all these works that built on top of the TrustZone UI solution and can provide an easy-to-use UI solution for them.

3.3 Secure Network Communication

This section reviews related works to secure the mobile-server communication path.

The first type of work [67, 80] moves all layers of communication, including the TCP/IP stack, into the secure world. Although this approach creates a completely isolated communication channel for sending the TEE-protected data to servers, it significantly increases the TCB size. To lower the size, the second type of work [42, 66, 83] keeps the TCP/IP stack in the normal world but ensures that the payload is encrypted in the secure world, and the normal world cannot know the actual content.

TruWalletM [42] proposes to use two logical SSL subchannels to protect users' login credentials. They simulated the design completely in the normal world. We have developed a practical solution on the real hardware instead of using simulations, and have identified significant challenges in the *split SSL* design (Section 5.2.2) that would not be discoverable only via simulations. DroidVault [66] requires app-specific logic in the secure world to manage the sensitive data from the server and requires the server to adopt the mutual authentication protocol to communicate with the secure world. This dissertation proposes a design that requires no app-specific logic in the secure world and is transparent to existing network protocols. Furthermore, neither DroidVault nor TruWalletM considers the ease of adoption to apps; they both require significant modifications to the app logic. KNOX [83] requires apps to use new KNOX APIs, which directly link the normal-world application logic with TrustZone components in the application layer. KNOX integration requires introducing new TrustZone components in the application layer and cannot reuse any existing Android components to leverage TrustZone. Unlike KNOX, this dissertation proposes a design that allows developers to use the standard Android APIs because the modifications are done inside the existing Android components. Since the data in the secure world cannot be revealed to the normal world, all solutions require moving part of the data-sending logic into the secure world if the data sent to the server contains TEE-protected information. However, whether we can maintain the same APIs for the developers is decided by where the logic is split. All existing works perform the splitting at the application layer [42, 66, 83]. We work on the underlying layers, including HTTP and SSL, while leaving the interaction APIs between the application and these underlying layers the same.

TinMan [95] offloads the confidential data in mobile apps into a trusted node. TinMan synchronizes the SSL states between the client and the trusted node. TruWallet [95] and TruWalletM [42] protect the confidential data in the wallet. Both works create SSL proxy and use two SSL connections to isolate the untrusted connection from the trusted connection.

3.4 Android Privacy

Privacy enhancement. Researchers have proposed various solutions [46, 49, 50, 51, 68, 81, 87, 91, 99, 102, 103, 110] to prevent privacy leakage in Android. MalloDroid [50] detects potential vulnerabilities against MITM attacks for Android apps. Zhang et al. [103] prevents sensitive runtime information gathering by monitoring suspicious background processes. TISSA [110] proposes fine-grained user privacy access control during runtime. Screenpass [68], TIVO [51], Secure Input Overlay [87] and Guardroid [91] protect users' passwords by modifying the Android framework. Unlike these works that fail when the Android OS is compromised, This dissertation proposed the solution that can preserve the user privacy even when the OS is compromised.

Privacy leakage. Researchers also identified various attack surfaces in the Android framework to leak users' private information. Jin et al. [60] systematically studied the new JS code injection channels on mobile to steal users' private data. Zhang et al. [107, 108] discovered a new type of data residue vulnerability in the Android framework that can retrieve users' privacy footprint.

Android UI Security. Researchers have discovered various vulnerabilities [41, 43, 52, 62, 76] in Android UI system and proposed several solutions [29, 41, 77, 79] to mitigate the problems. Unlike 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 [43, 52, 76] to render phishing UI. Antonio et al. [41] and WindowGuard [77] proposed solutions to mitigate the UI task hijacking. Yanick et al. [52] and Luo et al. [70] 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 [29]. However, the Android secure window cannot protect the UI in the event of a compromised OS.

4. DESIGN MODEL OF TRUSTZONE TEE

As discussed above, current TEE solutions tend to have a large size of TCB and are difficult to integrate with third-party applications. To overcome these limitations, this dissertation proposes a new TrustZone TEE model to help applications protect their data on the mobile communication paths. This chapter explains the main idea of the proposed TEE design model, and systematically compares design trade-offs between the traditional TEE model and proposed TEE model. Two high-level design challenges are identified when the proposed model to protect the mobile communication paths is applied (described in Section 4.2). To evaluate the effectiveness of the proposed TEE model, this dissertation applies it to protect two representative mobile communication paths: the mobile-server communication (Chapter 5) and the User Interface interaction (Chapter 6).

Two directions are identified by this dissertation in TrustZone TEE design, as shown in Figure 4.1. The current design direction of the TrustZone TEE solutions [17, 75, 92] supports a fully functional code stack in the secure world mainly because these TEE solutions support TAs to build code logic in the secure world. Such a TEE design direction requires an individual code stack (i.e., framework and TA) to be presented in the secure world. This dissertation names such a design model the *self-contained TEE model* (see the left side of Figure 4.1). To secure data confidentiality and integrity of communication paths, the past research [63, 65, 83, 88] has built various security protection mechanisms (i.e., TAs) on top of the current TrustZone TEE solutions [17, 75, 92].

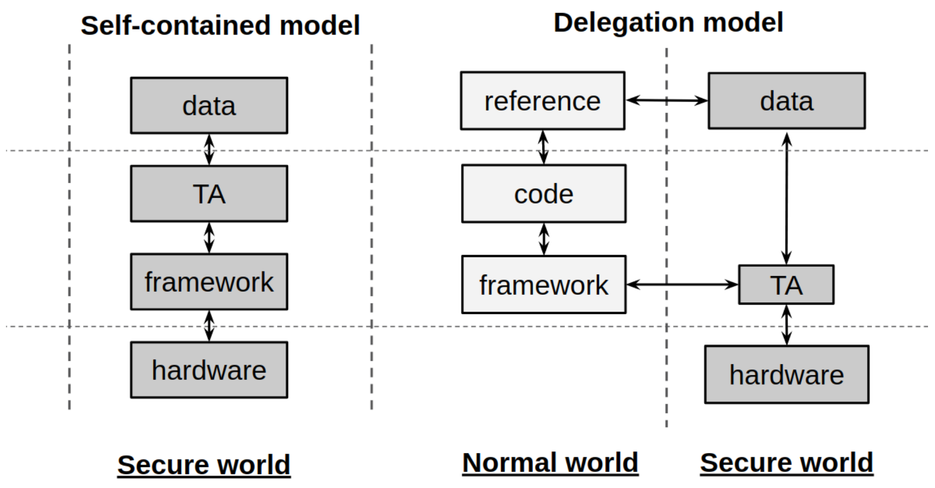


Fig. 4.1.: TEE Design Models

Despite security measures embedded in existing self-contained TEE model, current design has failed to manage two design principles listed in this dissertation: (1) a small TCB, which should be more easily verified in comparison to a rich OS; and (2) an easy-to-use TrustZone solution, which should be easily integrated by all third-party applications.

The need for a large size of TCB lies in the fact that the self-contained TEE model follows the same design direction as the rich OS, which requires an individual functional code stack in the secure world to support various code logic. This dissertation categorizes these TEE code logic into three groups. First of all, various protected data types require TAs to provide specific protection that meets the needs of each specific data type. Second, TAs need to leverage various protection mechanisms to protect data as well as to manage access control of the TrustZone-protected data. Third, TEE OSes need to include a separate running environment to support vibrant TAs running in it. These complex logics, thus contribute to a large size of TCB.

Another challenge self-contained TEE model often experience is how to enable vibrant third-party applications to use TEE. To use TrustZone protection, third-party application developers have to work with TEE vendors because the latter develops TEE frameworks including SDK (e.g. Trustonic SDK [19]) and TAM services (e.g. Intercede MyTAM [12]) with OTrP [13]. Furthermore, utilization of TEE requires collaborating with vendors to tailor applications for vendors' SDK. As a result, the majority of application developers find it difficult to utilize the advantage of TrustZone in protecting users' activities. Device and TEE vendors are reluctant to open TEE to any app because of the security concerns that vendors do not want any untrusted code to run inside the secure world. This restriction is necessary because allowing untrusted code to run is exactly what has contributed to the OS compromise in the normal world. This is also the reason why TrustZone is locked down in commercial phones before phones are shipped, so nobody (other than the vendors) can make changes to the code inside the secure world.

The main insight to the aforementioned problem is easy integration of applications with TrustZone support if the TrustZone support is at the normal-world OS level. In comparison to those who follow the traditional TEE integration approach, application developers who can reuse the existing OS interfaces are able to integrate their applications with TrustZone and conduct fewer changes. Furthermore, only the protection mechanism, which is the essential component to protect data on the communication paths, needs to be moved into the secure world. Other non-sensitive operations such as data-type dependent logic, data management, and the logic that uses the protection mechanism can be conducted by rich OS components. We can significantly shrink the size of TCB if the majority of the non-security related logic is moved to the normal world.

Based on the insight discussed above, this dissertation proposes *delegation TEE model* for TrustZone (see the right side of Figure 4.1). Delelgation TEE model provides generic protection mechanisms in the secure world, and integrates these protections with the normal-world OS. Application developers will have the option to use TrustZone protection when they write the program using the normal-world OS interfaces. The secure world protects data and never returns plaintext data to the normal world. To keep a small TCB size, the design delegates the application development and other non-sensitive operations to the normal-world framework. In delegation TEE model, normal-world applications obtain the reference to the TrustZone-protected data and run the data-dependent logic on the references. The secure world will replace the reference with the real data before data is put on communication paths. By leveraging the generic protection support integrated at the OS level, any application can use existing OS interfaces to securely transfer the protected data on the communication paths without exposing data to the risks in the normal world.

4.1 Design Trade-offs

To better understand the characteristics of different design models, this dissertation systematically compares trade-offs between the *self-contained TEE model* design and that of the *delegation TEE model* in three aspects, namely security, system design, and application impact. The comparison result is summarized in Table 4.1. Based on the comparison result, this dissertation further derives the design principles when the delegation TEE model to design TrustZone solutions is applied.

Table 4.1: TEE Design Models Comparison Summary

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

Security. TCB's size and isolation boundary are two essential attributes to measure the level of security of TrustZone solution. Compared to self-contained model that requires a separate code stack and has a large TCB, the delegation model reuses the majority of the normal-world code stack, and it only requires a small TCB in the secure world to support generic security mechanisms. As shown in table 4.1, the self-contained model could isolate two worlds completely as the code stacks in two worlds are indepedent from each other from a software engineering's perspective. On the other hand, to collaboratively protect data on communication paths, code stacks of delegation TEE model are coupled together. However, the closer coupled two worlds are, the less secure an isolation solution is. The TEE solution that applies the delegation model has to be a clean design from a software engineering perspective.

System design. Two critical measurements for TEE system design are code reusability and code modularity. The former implies that the software stack developed for one world can be reused for the other world, while the latter suggests that system update of one world should not affect the other world. When it comes to code reusability, software developed based on the self-contained model always suffers low code reusability because different OSes have different interfaces and development standards. The delegation model, on the other hand, reuses the majority of the normal-world software stack including the OS level and application level. Therefore, the delegation model can obtain high code reusability. Nevertheless, it cannot complete an indepdent system update because software stacks of both worlds are coupled closely, which is not an issue for self-contained model whose code stacks in two worlds are completely separate. To resolve this very issue, TEE design that applies the delegation model has to provide an insight to preserve modularity between two worlds.

Application impact. Transparency, consistency, and rich usability are considered three important attributes in assessing TrustZone design impact on applications. Transparency means the efforts that application developers need to make to integrate the TrustZone solution with their applications. In the self-contained model, application developers work with vendors to develop logic in the secure world, and such an approach demands application developers to great extent to leverage TrustZone solution. Moreover, placing application-specific logic in the TEE is not secure because doing so broadens the normal-world attack surface. The delegation model has high transparency thanks to its capacity to reuse normal-world application logic to leverage TrustZone protection and does not put any application-specific logic in the TEE. Secondly, consistency in TEE design focuses on two perspectives: (1) same user experience is provided across worlds, and (2) application developers maintain the development environment across worlds. Take UI as an example – users usually experience inconsistency when interacting with the self-contained TEE model because separate UI stacks present different screen images.

Furthermore, to develop logic for TEE, the self-contained TEE model requires application developers to use vendor's SDK, which is not the UI development environment for the normal world. The delegation model can provide consistent UI experience across worlds because a single UI stack produces screen data for both worlds. The UIs from both worlds will be constructed in the normal-world development environment. Third, the delegation TEE model delegates the rich functionalities (e.g., animation, etc) into the normal world. The users only invoke the secure world when they need to protect sensitive data on communication paths. The self-contained model can support rich usability depending on how large its TCB is.

Delegation TEE model design principles. As discussed above, this dissertation summarizes design principles of the delegation TEE model to demonstrate its difference from its counterpart. These principles, when delegation model is applied to design TrustZone solutions, will serve as guidelines to preserve the aforementioned properties: (1) maintain a small TCB in the secure world and reuse the rich OS for non-sensitive operations as much as possible; (2) find a clear boundary between two worlds from a software engineering perspective; (3) require a minimum effort for applications to integrate with TrustZone integration and for system to update; (4) offer consistent user experience and a consistent development environment across worlds.

4.2 TEE Design Challenges

This section discusses challenges in applying delegation model to design TEE. The secure world loses many capabilities during this TCB reduction process because the delegation model pushes the majority of the code stack out of the secure world. However, some lost capabilities are paramount to protect data on communication paths in the secure world, and hence, have become challenges for my TEE design.

First of all, the secure world is no longer able to develop data protection logic independently. The new design model delegates code development to the untrusted normal world in which all code and data processed by the normal world are considered untrusted. How can the secure world leverage the **untrusted** normal-world code stack to protect data for the secure world?

Secondly, the secure world loses all runtime states because the delegation model solely provides protection mechanisms in the secure world while all runtime states are maintained by the normal world. Some runtime states are important to define how to protect data on communication paths. This dissertation suggests a way to recover **lost runtime states** in the secure world.

This dissertation applies the delegation TEE model to protect the mobile-server communication and protect the User Interface Interaction. The solutions to overcome these design challenges will be explained in Chapter 5 and Chapter 6.

5. SPLIT SSL AND TRUZ-HTTP

Nowadays, smartphones support numerous functionalities including banking, online transactions, and other financing activities, which relies on efficient and confidential communication between applications and their backend servers. In a compromised OS, malware can act as the medium between applications and servers, and steal users' sensitive information such as bank passwords and credit card numbers. A compromised OS can even spoof actions such as acting as bank customers to transfer money out of user's bank accounts without being caught by bank servers. This security loophole is caused by servers' inability to distinguish a real user from malware. To address this security risk, TrustZone provides applications with an isolated mobile-server communication path to help applications protect users' sensitive activities on their smartphones.

The self-contained TEE model (chapter 4) presents two primary types of integration methods for mobile-server communication. First of all, device vendors proactively develop trusted applications that help applications communicate with servers. For instance, Samsung Pay works with major financial services such as Visa and MasterCard, and place their payment services in the secure world so users' payment information can be protected in TrustZone. Second, non-vendor applications can work with TEE vendors and place their code in the secure world. For instance, Alipay works with T6 [17] and secures users' payment information in the secure world. However, both approaches require applications to set up their logic in the secure world and to heavily tailor their existing applications. Deploying application logic in the TEE is not only insecure but also difficult-to-use for a mass of application developers, preventing them from utilizing TrustZone to secure mobile-server communication path.

Allowing third-party apps, such as banking, shopping, and medical apps, to use TrustZone can significantly benefit users. Apps commonly transfer users' sensitive data from/to their backend servers. Taking mobile banking as an example, we have identified three potential risks when a banking app does not use TrustZone. First of all, when users login bank's server, their passwords are exposed to theft during mobile-server data transmission when OSes are compromised. Second, clients can sometimes pay using payment barcodes they received from the server. An untrusted normal world has full access to payment tokens downloaded from servers and can use tokens for other transactions. Third, when the user conducts a monetary transaction, a compromised OS can replace the receiver's account number with the one belonging to an attacker, and leads to substantial financial loss. If such an app uses TrustZone, the aforementioned risks would be minimized. TrustZone allows applications to send passwords form TrustZone to the server without exposing it to the untrusted normal-world OS. TrustZone can also receive a payment token from servers without letting the normal world creep into the actual data. Moreover, before an important transaction is committed, TrustZone request confirmation from users, so that the transaction can be attested and its integrity can be preserved.

It is important to allow apps to use TEE via existing normal-world OS APIs without having to install app-specific TA in the secure world. This is a challenging requirement.

Without such support, developers need to make significant changes to their apps to use TrustZone, discouraging them from using it in their apps.

This dissertation presents a design that applies the *delegation TEE model* described in Chapter 4 to secure server communication. The design allows normal-world apps to leverage TrustZone to communicate with the server via existing OS APIs. By incorporating the generic TrustZone support at the OS level, normal-world apps can use TrustZone without building their own code into the secure world. Based on this idea, this chapter delineates the design of the *TruZ-HTTP* and *split SSL* that can transfer the TEE-protected data between the authorized server and TrustZone. We have implemented this system in the Android OS, running on a prototype that uses the TrustZone-enabled HiKey board. Moreover, we perform an evaluation of the system using real-world apps, including both open-source and closed-source apps. The evaluation results show that the system allows third-party apps to leverage TrustZone for a variety of use cases with minimal changes to apps. The performed usability study of the system shows that users can make the right access control decision to protect their security sensitive activities using the system.

5.1 Problem and Ideas

In this section, we discuss problems and constraints in providing TrustZone support to third-party apps and our ideas on how to solve these problems.

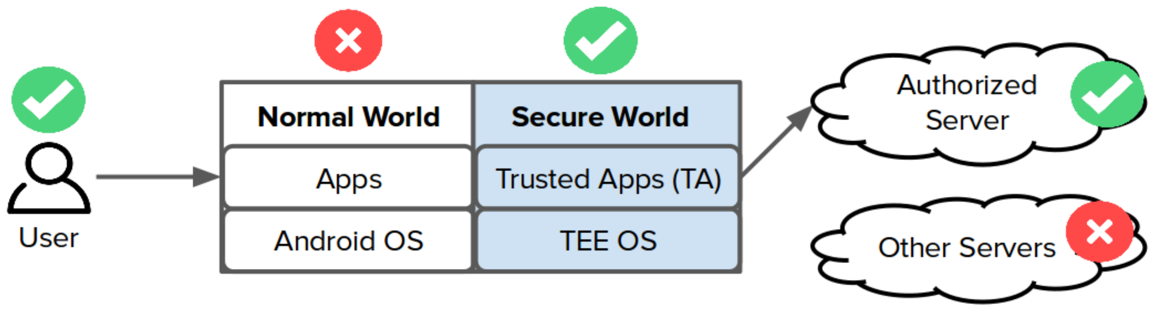


Fig. 5.1.: Threat Model

5.1.1 Threat Model

We assume the adversary model as shown in Figure 5.1. The user of the mobile device is trusted. The normal world that includes applications and the Android OS is untrusted. They may attempt to steal the user's secret data and spoof an unauthorized action on the user's behalf. The secure world that includes Trusted Applications (TA) and the TEE OS is trusted. It will protect the user's confidentiality and integrity when the normal world is compromised. We assume that the server is trusted after it is authorized by the user. The authorized server wants to protect the user's confidential data and verify the integrity of the user's request; other unauthorized servers are considered untrusted and they may collaborate with the normal world to steal the user's secret data.

5.1.2 Problem

We state the following problem: *How do we enable third-party apps to reuse the existing OS interfaces to leverage generic TrustZone support to protect user's private data without putting app-specific code in the TEE?*

We further break down the problem into protecting user's sensitive data and transferring it between the server and TrustZone. Users type sensitive inputs when using Android applications. In order for an app to protect such inputs using TrustZone, users should be able to type a secret without allowing the compromised OS to see the secret. This dissertation will explain the protection of user interaction in Chapter 6; this chapter focuses on asking how to transfer the TrustZone-protected data between the authorized server and TrustZone. Given a protected secret, the app should be able to send the secret to the authorized server without leaking the secret to the compromised OS. Applications should also be able to load protected data from a server without leaking the content to the normal world. In order for an app to enforce the user's intention using TrustZone, users should be able to confirm an action (e.g., money transfer) and the compromised OS should not be able to modify the user's confirmed action. The user's confirmation should be attested (signed) using TrustZone. The attested confirmation should allow the receiving server to verify that the action was confirmed by the user.

The problem of protecting the user's sensitive data and the user's intention has been solved by TrustZone, but the current solutions [66, 73, 80, 83] do not satisfy the following constraints: (a) normal-world apps can reuse existing OS interfaces to leverage the TrustZone support, (b) there is no app-specific logic in the secure world, and (c) the minimizing of Trusted Computing Base (TCB) while providing generic TEE support. In order to allow an app to protect user's activities and data with minimal changes, the developer should be able to use existing Android components and APIs, and still be able to leverage TEE support. If an app is required to replace Android components to integrate TEE support, it would result in a significant change to the app. An example of this is

Samsung KNOX [83], which provides a vendor SDK that allows app logic to be integrated with TrustZone components in the secure world. It is important to not require app developers to write code in the secure world, which minimizes the risks to the secure world. Research works [66, 73, 80] allow application-specific logic to run in either TAs or TEE OSes [18]. Such an approach restricts the number of apps that can use these research works and increases the risk in the secure world.

Given the identified constraints, we further elaborate on the problems of secure server communication.

The secret data typed inside the secure world need to be used by applications. If the data are to be processed on the client side (i.e., by apps), it would be difficult not to reveal the data to the normal world. However, in most applications, these secret data, such as passwords and credit card numbers, are only processed on the server side, so they need not be revealed to the normal world, as long as they can be sent from the secure world to the server securely. In another scenario, users want to securely display the data downloaded from the server. For example, users can download a payment barcode from a server and display it to a merchant. As long as they can display the protected data in the secure world, the data need not be revealed to the normal world. Applications commonly conduct these activities through the HTTPS protocol. Given that the normal-world OS is untrusted, we need a way to *allow apps to reuse existing HTTP interfaces to transfer the TEE-protected data between the authorized server and TrustZone without leaking the content to the normal world.*

5.1.3 Design Trade-off

In Chapter 4, we did a high-level comparison between the *self-contained model* and the *delegation model*. To better understand the characteristics of different design models when applying them to secure server communication, we systematically compared the design trade-off between the self-contained model and the delegation model in three aspects, namely security, system design, and application impact.

Security. TCB size and isolation boundary are two essential attributes to measure the security of server communication. The self-contained model requires an individual network stack (i.e., TCP, IP, data link layer and physical layer) for securing the communication, which requires a large TCB in the secure world. By contrast, the delegation model reuses the network stack in the normal world and only conducts security sensitive operations in the secure world. As a trade-off for TCB size, the self-contained model provides a complete isolated communication with their servers, while the delegation model requires the untrusted normal-world network stack to communicate with the server. Therefore, the secure network design that applies the delegation model has to answer this challenging question.

System design. Code reusability and modularity are two critical measurements for the TEE system design. First of all, code reusability implies that the server communication logic developed for one world can be reused for another world. The logic developed based on the self-contained model always suffers low code reusability because different OSes have different network stacks and different interfaces. In contrast to the self-contained model, the delegation model reuses the majority of the normal-world software stack including the OS level and application level. Therefore, the delegation model can obtain high code reusability. Secondly, modularity suggests that the network stack update of one world should not affect the other world. The self-contained model can complete an independent update for a network stack because both worlds' network stacks are entirely separated. As a trade-off for code reusability, the delegation model cannot merely modularize two worlds into two modules. The design that applies the delegation model has to provide an engineering insight to reduce dependency between two worlds.

Application impact. We use transparency, consistency, and rich usability to assess the TrustZone design impact on the applications. Firstly, transparency is how many efforts application developers need to make to integrate the TrustZone solution with their applications. The self-contained model requires application developers to work with vendors to develop server communication logic in the secure world, and such an approach requires much effort of application developers to leverage TrustZone. Moreover, putting application-specific logic in the TEE is not secure. The delegation model has high transparency to the application because our model reuses the normal-world network stack to develop the secure server communication and does not put any application-specific logic in the TEE. Secondly, consistency means whether or not applications can use the same network protocols to communicate with servers across worlds. TAs usually develop tailor-made network protocols to communicate with their servers because servers need to differentiate two worlds by using different network protocols. However, the tailor-made protocols are not well studied and usually contain security issues. The delegation model uses the standard network protocols (i.e., HTTP and TLS) for both worlds, which are well studied. We do not want applications to reinvent the wheel and will provide our insight to let server differentiate the TEE and normal world using standard network protocols. Thirdly, rich usability refers to whether applications can generate and process network packets in the secure world. For example, the secure world can process payment requests and do authentication. As a trade-off for TCB size, the delegation model cannot allow applications to process the packets inside the secure world but can only protect data on communication paths. The self-contained model could support such functionalities depending on how large its TCB is.

Design principles. To reach a conclusion on the comparison, we summarize the design principles of the delegation model for secure server communication. When we apply the model to protect the communication path, these principles will help us preserve aforementioned properties of the delegation model: (1) maintain a small TCB in the secure world and reuse the normal-world network stack for server communication; (2) from an engineering perspective, find a clean cut to reduce dependency between two worlds; (3) require a minimum effort from the TrustZone integration for applications and system updates; and (4) use standard network protocols for both worlds.

5.1.4 Design Challenges for Secure Server Communication

In this section, we discuss high-level design challenges for secure server communication by applying the delegation model. The secure world loses many capabilities during this TCB reduction process because our design model reuses the normal-world network stack. However, some of the lost capabilities are important for the secure world to protect the mobile-server communication and become challenges to the design. Moreover, there are challenges particular related to integrating TEE with the network stack; we will discuss them in Section 5.2.1 and Section 5.2.2.

First of all, the secure world loses the capability to communicate with servers independently. Our design model delegates the communication to the untrusted normal-world network stack. All the data processed by the normal world is considered as untrusted. How can the secure world leverage the untrusted normal-world network stack to communicate with the server?

Secondly, for performance reasons, our solution should only be used for communication involving TEE-protected data; other data should simply be transmitted by the normal channel between the app and its server. This requires changes at multiple layers, including HTTP and SSL. Developers know whether TEE-protected data is involved or not, but since they can only tell their intention to the highest layer (the HTTP layer), we need to find ways to convey the developer's intention through multiple layers inside the OS. Moreover, based on the server specific logic, the attacker can trick the server to return a protected secret back to the normal world or post the secret to a website's public field like a Twitter post or a Facebook post. We need to find a way to extend the TEE protection to servers without being intercepted by the normal world.

Third, the secure world loses all network states used to protect the data on the mobile-server communication path because all network states are maintained by the normal-world network stack, while some of network states define how to protect the data on the communication path. From an engineering perspective, we need to find a way to synchronize minimum states between two worlds in order to reduce the dependency between two worlds.

5.1.5 Our Main Ideas

Our main idea to solve the problem is to provide TrustZone support at the OS level, so apps can reuse existing Android components to integrate with TrustZone support with minimal changes. To reduce the risk to the secure world caused by the app-specific logic, we provide generic TAs.

We work on the network protocol layers, including HTTP and SSL, while leaving the interaction APIs between the application and these underlying layers the same. We split the logic in the network stack in a way that apps can reuse the same HTTP interfaces to send TrustZone-protected data without leaking to the normal world. Because the data in the secure world cannot be revealed to the normal world, existing solutions [42, 66, 83] require moving some of the data-sending logic into the secure world if the data being sent to the server contains TEE-protected information. They perform the splitting at the application layer, this forces the app developers to rewrite their app logic to run on the TEE OS. To be transparent to existing network protocols, we do not modify the protocols but move only the security sensitive logic in these two layers into the secure world. The secure world will encrypt the secret using a one-time key and SSL session key before giving it back to the normal world for sending. The server will decrypt the secret using the one-time key only when the server expects to use the secret in the HTTP request.

5.2 TrustZone-Enabled Interaction with Server

Normal-world applications can leverage TrustZone UI to allow users to enter sensitive information such as a password, payment information, etc. The secret data are not revealed to the normal world, but they need to be sent to servers. It is important for users to confirm the server hostname inside the secure world; we will explain the importance in Section 6.5. All the design related to the UI will be described in Chapter 6.

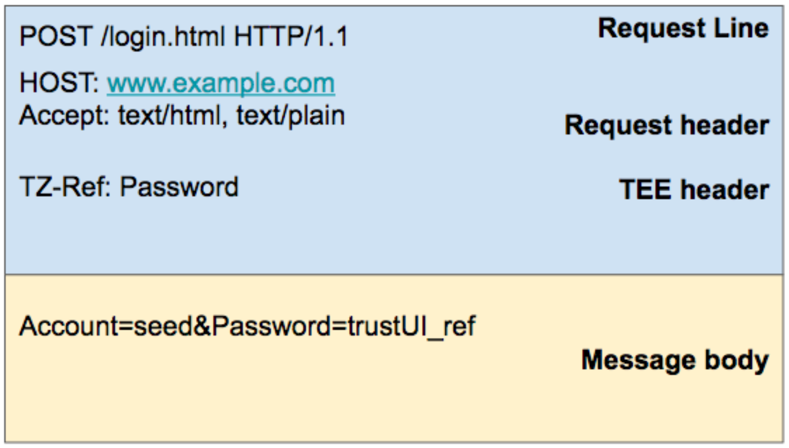
In our design, users make the final access control decision through the TrustZone UI described in Chapter 6. The OS depends on the user's action to decide how to provide confidentiality and integrity protection for user intended activities. For instance, when a user types a password, he/she depends on the OS (based on the app picked) to provide confidentiality, i.e., the password should go to the right app and its corresponding server. When a user confirms an action in an app, he/she expects the OS to maintain the integrity of the action, i.e., the action that the user confirmed is sent to the server, without being modified. The OS provides confidentiality and integrity guarantees by enforcing access control based on a policy. Part of this policy is decided by the OS, but the other half comes from the user and is derived from the user action. When the user types a password, the OS depends on the user's app selection to decide which app gets the password. When a user confirms an action for a server, the OS can only guarantee that the context of the action will not be modified after the user's approval; the main job of the user is to proofread and ensure that the context of the action indeed matches the user's intention. In our threat model, the normal-world OS fails to provide such security guarantees for users when it is compromised. The only solution for users to protect their security sensitive activities is to convey their intentions to TrustZone to leverage its confidentiality and integrity guarantees. That is why we ask users to confirm the confirmation message and the hostname inside the secure world.

Given that the normal-world OS is untrusted, we need a way for applications to send the TEE-protected data to the authorized server and receive TEE-protected data from the server without disclosing the content to the untrusted normal world.

5.2.1 TruZ-HTTP

Most apps interact with their servers through HTTPS [11], especially when the data being sent contains secret information, such as passwords and credit card numbers. HTTPS is basically HTTP running on top of the SSL protocol. When an app uses HTTPS, six primary steps are involved: (1) the app provides the URL of the web server to HTTP; (2) the app provides HTTP headers (if needed); (3) the app provides data to HTTP; (4) based on the URL, HTTP invokes SSL to establish a secure channel with the server; (5) HTTP constructs an HTTP request based on the data provided by the app; (6) HTTP gives the completed HTTP request to the SSL layer for sending.

As shown in Figure 5.3, Steps 1 to 3 involve the app, so changes in these steps should not be at the interface level in order to maintain the same interface. Since not every HTTP request contains TEE-protected data, for those that do, the secure world should be involved in the sending process; other non-secret-bearing requests should be sent out entirely from the normal world to avoid the overhead introduced by TrustZone. HTTP does not know whether the payload involves TEE-protected data or not; only apps know



POST /login.html HTTP/1.1
HOST: www.example.com
Accept: text/html, text/plain

TZ-Ref: Password

Account=seed&Password=trustUI_ref

Fig. 5.2.: TrustZone-enabled HTTP request

that. The question is how to enable apps to inform HTTP about this without changing the way in which they interact with HTTP. We use HTTP headers to solve this problem. We create a new HTTP header field that allows developers to tell HTTP whether the payload contains a reference to the TEE-protected data or not, and if so, where the reference is in the payload. The Figure 5.2 is an example of HTTP request that contains the new HTTP header. In the case of exchanging an attestation key with the server, the app creates an HTTP header with an empty value. The app developers can decide the keep-alive time (for a login session or always alive) for the attestation key. The secure world will fill in the attestation key value for the app in our modified SSL layer (Section 5.2.2). This solution does not change how apps interact with HTTP and it only adds an extra task in Step 2. We introduce a plugin called *TruZ-HTTP* for the HTTP engine to parse the new HTTP header.

To convey the TrustZone information to the SSL layer, TruZ-HTTP adds additional TrustZone logic to Steps 4,5, and 6. If an HTTP request involves TEE-protected data, the SSL channel used to send the request must be established by the secure world, so that the

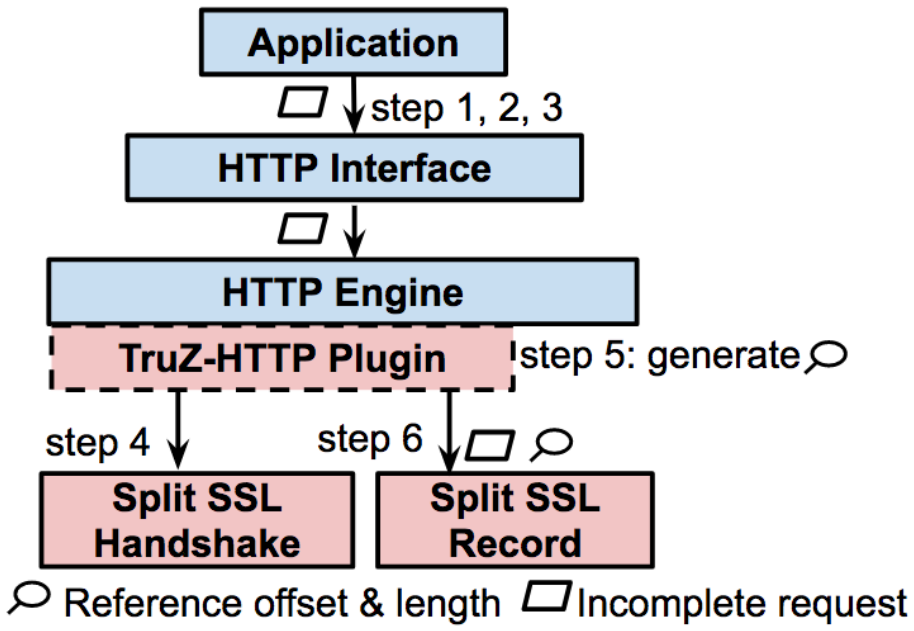


Fig. 5.3.: TruZ-HTTP Plugin Design

encryption keys used by the SSL channel are not revealed to the normal world. Therefore, in Step 4, HTTP will invoke our modified SSL library so that the secure world is involved in establishing the SSL connection. We provide a detailed discussion on this part in Section 5.2.2, where we discuss our split SSL design.

After the SSL connection is established, HTTP can give a completely constructed HTTP request to SSL. However, at this point, the secret data inside the request are still represented by their references, not by the actual content. SSL needs to replace the references with the actual content, requiring SSL to know where the references are in the request. This information is known to the HTTP engine after it parses the additional HTTP header provided by the app (the header will be removed after the parsing). Therefore, the HTTP engine has to convey the reference information to SSL. However, SSL is not supposed to understand the logic in the layers above it (e.g., HTTP format).

TruZ-HTTP parses the additional header in Step 5 and generates offset and length for each reference. TruZ-HTTP converts the HTTP format specific information into generic string offset and length so that SSL can replace all the references in the request with their actual content without parsing the HTTP request. The additional reference offset and length information is handed over along with the HTTP request to the SSL layer in Step 6. The TruZ-HTTP design assumes that there is no app specific integrity check (e.g., hash) in the HTTP payload. Based on our randomly collected app sample in Section 5.4.2, most apps meet our assumption.

5.2.2 Splitting SSL

When apps send sensitive data to their servers, they either go through HTTPS, which is built on top of SSL, or they go directly through SSL. If the sensitive data is in the secure world, part of SSL has to be carried out in the secure world as well.

The main challenge to run SSL in the secure world is to maintain the SSL state information between the two worlds because SSL is a stateful protocol and the TCP/IP stack is still in the normal world. Simply running the entire SSL implementation in the secure world requires maintaining the SSL connection in the secure world. However, our current design only focuses on protecting a secret on the mobile-server communication path while the encrypted data transferred between the server and the mobile should be performed in the normal world entirety. Simply running two copies of SSL in both worlds and synchronizing the SSL states between them is also not a viable solution. The data types of many SSL states are dynamically cast during runtime. Without knowing the actual data types, the states are not serializable between worlds. Our first design moved part of the SSL steps into the secure world and updated the states that have static data type between the worlds. This design was extremely complex because of the complexity of the SSL states and required significant modification of the SSL implementation. From an engineering perspective, we discarded this approach for a cleaner design. A clean splitting SSL design should be stateless across the two worlds and should only run security sensitive logic that deals with the secret in the secure world.

We decided to split SSL between the normal world and secure world to keep all the SSL states in the normal world. SSL can be further divided into SSL session layer and crypto layer. All SSL states are maintained in the SSL session layer. The crypto layer conducts all the crypto operations for SSL and no SSL states are updated in this layer. As shown in Figure 5.4, we decided to split under the interface of the crypto layer and moved the crypto layer into the secure world. Using our splitting approach, the normal world transfers only essential SSL states to the secure world. Upon exiting from the SSL TA, the crypto return value will be sent back to the normal world. It should be noted that a subset of the crypto return value contains sensitive information that should only be known to the secure world (i.e., the keys used to encrypt the data to be sent to the server). For these crypto return values, the references of keys are returned back to the normal world. The SSL TA stores the actual keys in the secure-world memory.

SSL contains two sub-protocols: the handshake protocol and the record protocol [40]. Our design involves splitting these two protocols. For the handshake protocol, if an app needs to use SSL to send data to its server (when part of the data is stored in the secure world), the app, either directly or via HTTP, must go through our split SSL to conduct the

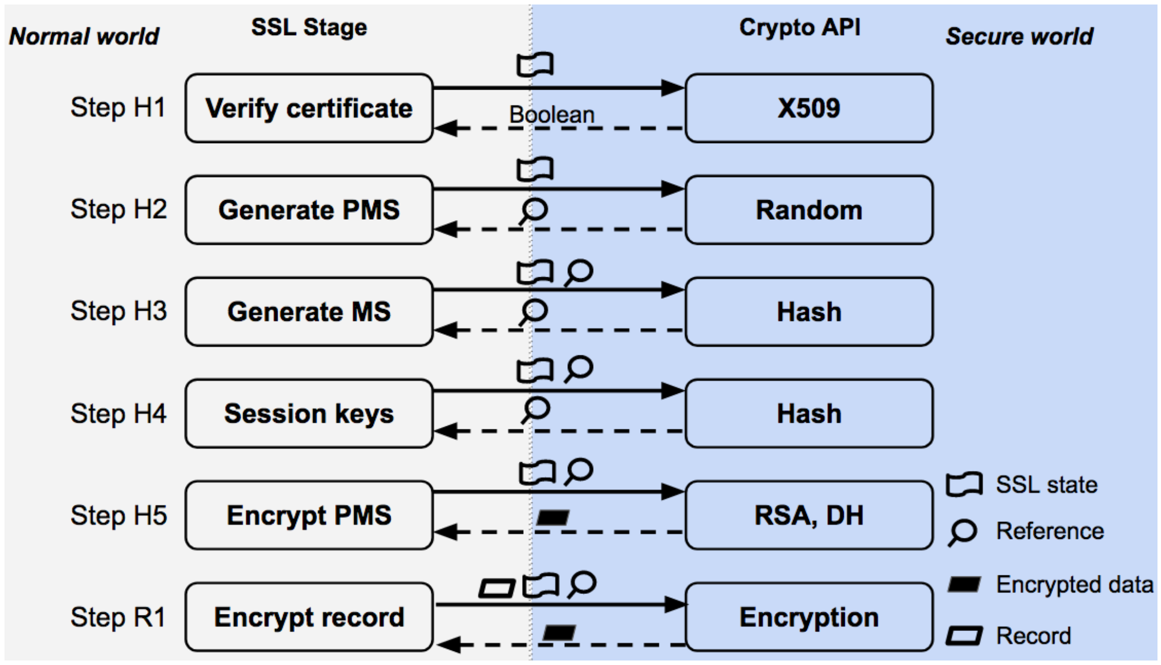


Fig. 5.4.: Splitting SSL Design

handshake with the server. To keep all the handshake SSL states in the normal world, we keep most of the handshake logic in the normal world, except the crypto logic of certificate verification and key exchange. As shown in Figure 5.4, we further divide the handshake protocol into five main stages (marked as H1 - H5). For each stage, the actual crypto operation is done in the secure world; either the crypto result or the reference of the key is returned back to the normal world. The actual SSL keys are saved in the secure-world memory. Each SSL session is bound with the hostname extracted from the server certificate; the importance of doing so will be discussed later in our security analysis (Section 5.3.1).

At Step H1, we transfer the server certificate to the secure world. Our modified X509 certificate verification crypto verifies the server certificate using the trusted certificates pre-installed inside the secure world. The verification result is returned back to the normal world. The SSL random number function asks the secure world random device to produce pre-master-secret (PMS) at Step H2. We modified the hash APIs that derive the master secret (MS) at Step H3 and session keys from PMS at Step H4. The normal world only gets the references of PMS, MS and session keys as we store the actual secrets in the secure world memory. At Step H5, the SSL TA encrypts the actual PMS using the server's public key, and gives the encrypted PMS back to the normal world for transmission to the server. Table 5.1 summaries all crypto operations in each stage.

Table 5.1: Split SSL Step Detail

Step	**SSL Stage**	**Crypto in TEE**	**Return Value**
H1	Verify certificate	X509	Boolean
H2	Generate PMS	Random	Reference
H3	Generate MS	Hash	Reference
H4	Session keys	Hash	Reference
H5	Encrypt PMS	RSA, DH	Encrypted data
R1	Encrypt record	Encryption	Encrypted data

SSL record protocol involves two parts: record sending (SSL Write) and record receiving (SSL Read). This section focuses on explaining how to protect the data generated from the user device and send it to the server. We will explain how to download the protected data from server to TrustZone in Section 5.2.3. SSL has separate keys for SSL Read and SSL Write. Both keys are kept in the secure world. We modified the encryption logic of SSL Write, as shown in Figure 5.4 Step R1. When an app or HTTP invokes SSL Write to send data that contains a secret in the secure world, either the app directly or HTTP engine passes the offset and length of the reference as part of SSL states to the SSL TA. The SSL TA replaces the reference with the real secret. In case of an attestation key needed to attest user confirmation, the SSL TA generates a random key in the secure world. The SSL TA will encrypt the payload using the Write session key. The encrypted SSL record will be given back to the normal world, which will send it out as TCP data.

For record protocol, the SSL TA replaces the reference with the real secret. The SSL TA will XOR the secret using a one-time session key that is generated using TEE random device and encodes the XOR result using base64 encoding. The one-time key is inserted in the HTTP header. The server can extract the one-time key from the HTTP header, decode the secret value and XOR the secret data with the one-time key only when the server needs to treat the data as secret. We will discuss the importance of the one-time key in our security analysis (Section 5.3.1). In case an attestation key is needed to attest user's confirmation, the SSL TA generates an attestation key with random value and inserts it into the HTTP header. The SSL TA saves the key and binds it with the hostname as an index. The SSL TA encrypts the record using the Write session key.

5.2.3 Protect Data Downloading

In this section, we discuss the additional challenges of how we protect the data during receiving the protected data. Our system allows developers to use standard network protocols (e.g., HTTP, SSL) to download the confidential data from the server to TrustZone.

High-level solution. The high-level idea of data protection is shown in Figure 5.5. 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.

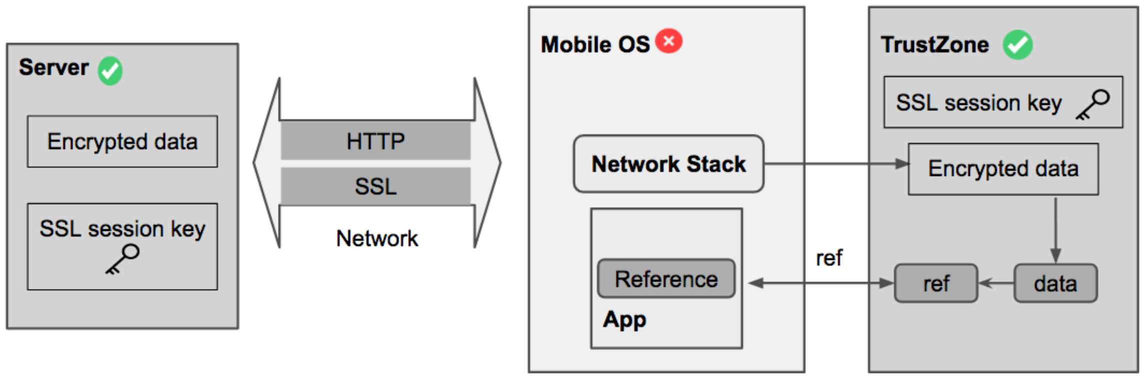


Fig. 5.5.: Data Protection High-level Solution

Engineering Challenges. The previous section 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. Figure 5.6 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 5.6 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 5.6 ❹.

Our solution manages all the partial references and their corresponding data in a reference management table. Table 5.2 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 the pieces, our solution concatenates all parts into one buffer and saves it in the secure world.

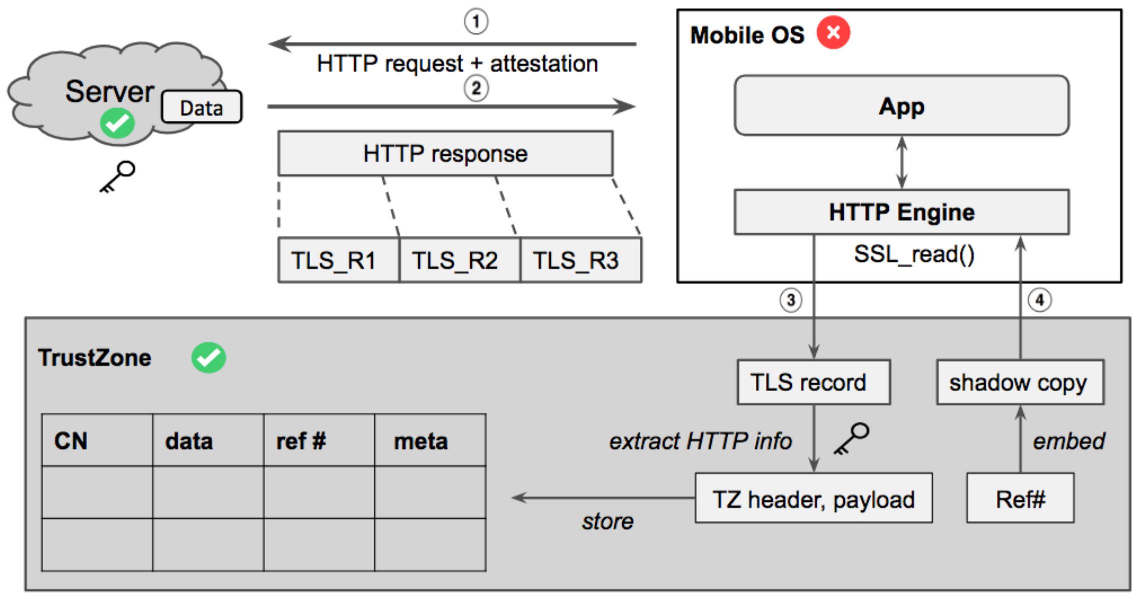


Fig. 5.6.: TrustZone-enabled HTTPS Downloading

Table 5.2: 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

5.2.4 Data Management

In this section, we discuss how application developers use the protected data. During the offline data usage, we allow normal-world applications to display the data stored in TrustZone without it leaking into the normal 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 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. The application developers can display confidential data using both types of reference.

5.2.5 Access Control Policy

This section summarizes all these access control policies. The normal world can use the data stored in the secure world through the reference. For instance, the normal world has access to the reference of the encryption keys and the reference to the protected data that is only sent to the server. Our system enforces a default access control policy when the normal world accesses the protected data through the reference.

Access control for accessing encryption keys. Here are the access control policies enforced on the TLS encryption keys:

- Every TLS encryption key is bound with a server hostname. The key can only be used to the encrypted data that is sent to the corresponding server. The normal world can use a session id to refer to all the encryption keys. The session id is also bound with the hostname.

- During the client key exchange message, the secure world verifies that the PMS is encrypted with the same public key extracted from the server certificate. We compare the hostname in the certificate with the encrypted PMS hostname and make sure they are the same before sending the encrypted PMS to the server.

- When users type the secret, they already confirm the hostname of the secret. When the secure world encrypts the data, we compare the server hostname of the data match with the encryption key hostname before using the key to encrypt the data.
- The encryption keys are bound with a counter to prevent plaintext brute-force attack.

Access control for accessing the data. Here are the access control policies enforced on the TLS encryption keys:

- The owner of the data has to match with the hostname of the server certificate. We return the encrypted data to the normal world only when these names match.
- Our solution inserts a one-time key in the HTTP header. The server will ensure that the HTTP request does need to have the secret data and then decrypt the secret using the one-time key.
- The secure world disables all the weak ciphers to prevent MITM attack.

5.2.6 Implementation

TruZ-HTTP adds 595 LOC to the HTTP engine (`OKHTTP`). The size is quite small compared to the total code size of the HTTP engine, which has 74,290 LOC. It is easy to incorporate split SSL design in both worlds. For the normal world, our prototype adds 1012 LOC in 10 functions in Android's SSL library (`boringssl`); for the secure world, our SSL TA consists of 1093 LOC. The crypto part of the `boringssl` library is also ported to OPTEE, but only minimal changes are made. It is also easy to migrate our design for a system update. We migrated our design from android 7.0.0_r1 to 7.0.0_r34. It took 5 hours for TruZ-HTTP plugin migration and 6.5 hours for split SSL migration.

5.3 Security Analysis

In this section, we present the security analysis of our system. Our design can enforce users' intentions in the presence of either a malicious app or a malicious OS. We pick the stronger attack model and consider a malicious OS as the attacker. Our analysis assumes that the TrustZone hardware platform is trusted and the secure boot process has initialized the integrity-verified OPTEE OS. Hardware attacks, crypto attacks, side channel attacks, and DOS attacks are considered out of scope.

5.3.1 TruZ-HTTP and Split SSL Security Analysis

In Section 5.2, we discussed how TruZ-HTTP and split SSL enable apps to send TEE-protected data to the server without the normal world knowing the actual content. To defeat this protection, adversaries can attempt two types of attacks: (1) stealing the encryption keys from the secure world, and (2) launching man-in-the-middle (MITM) attacks. The normal world with a compromised OS can trick the secure world into sending the secret data to a malicious server owned by the adversary or trick the TA to send the secret to a website's public field. The normal world can modify the plaintext handshake message to use a weak cipher or a vulnerable SSL version. All attempts are defeated by our security properties.

The first security property is that the key used to encrypt TEE-protected data is never visible to the normal-world OS. We use the key lifecycle to analyze the security of our key management. We can divide the key lifecycle into (1) generation (2) exchange (3) storage (4) usage and (5) destruction.

In the key generation, the SSL master secret (MS) is generated from the pre-master secret (PMS), client random number and server random number. Although the normal world can know the client random and server random numbers in plaintext, the PMS is generated in the secure world using a TEE-random device and is stored in the secure-world memory only. The normal world cannot infer the MS without knowing the PMS. The SSL key material is derived from MS. Without knowing the MS, the attacker cannot know the key material. Half of the key material is used for the SSL Read key and another half of the key material is used for the SSL Write key. Although we return the Read key back to the normal world, there is no correlation between the Write and Read keys because the key material randomness is based on the PMS randomness. Our TEE-random device ensures the enough entropy before generating the PMS. The PMS is generated right after the server certificate verification. We bind the PMS with the server hostname to prevent the normal world from misbinding the PMS with other servers.

During the key exchange, the secure world encrypts the PMS using the server public key before giving it to the normal world for sending. The normal world cannot know the plaintext PMS because the normal world does not know the server's private key that is protected by the server. The normal world cannot send the PMS to the wrong server because the PMS is bound to the server hostname, and the SSL TA refuses to encrypt the PMS using the wrong public key extracted from the certificate.

When the secure world stores the key, we store the PMS, MS, key material, and SSL Write key in the secure-world memory only. So the normal world cannot access the keys stored in the secure-world memory.

The normal world can provide the SSL session id that the SSL TA returns after certificate verification to use the SSL Write key for encryption. The encryption key is never returned to the normal world. The key is bound to the hostname. To prevent the normal world from using the wrong encryption key (known to the malicious server and stored in the secure world) to encrypt the TEE-protected data and sending it to the malicious server, the SSL TA ensures the hostname of the TEE-protected data matches with the key's hostname before doing the replacement and encryption. Each key is also bound with a counter. The counter is used to limit the time that the key can be used to encrypt the data.

We destruct the SSL Write key in the secure world after the secure world encrypts the packet that contains the secret. To avoid the normal world using the plaintext to brute force the key, we will also set the key value to zero when the key's counter reaches its limit. That way we limit the footprint of the key in the secure-world memory.

The second security property of our design is that the TEE-protected data is only sent to the authorized server. An adversary in the normal world with a compromised OS can steal the reference for a TEE-protected data, such as the password for Facebook, and ask the secure world to send the password to the adversary's server, which has a valid certificate. This attack is defeated by three critical decisions in our design. First, when getting a secret data item from the user inside the secure world, the user is presented with a message that clearly states the hostname of the server to which the secret belongs. Once the user approves it and types in the secret data, the data item is bound to the hostname. TrustZone will ensure that the data will only be sent to the server with that hostname. Second, during the SSL handshake protocol, after verifying the server certificate, the SSL TA extracts the common name from the certificate and binds the name to the current SSL session. This binding is saved in the secure-world memory, so the normal world cannot change this binding. Third, when the SSL TA sends out a protected data item to the server, it checks whether the hostname bound to the data item matches with the common name bound to the SSL session. If not, the secret data will not be sent out. The password for Facebook will never be sent to another server because of our protection.

Our design prevents the user's secret from leaking within the authorized server. For example, the attacker can trick the SSL TA to send out the Twitter password as a Twitter post so the user's secret becomes a public visible message on the Twitter website. The attacker can also append the secret reference to a redirect URL. In a redirect URL, the secret will be sent to the authorized server first. The authorized server will echo the redirect URL that contains the TEE-protected data back to the normal world. Because the split SSL design returns the SSL Read key back to the normal world, the compromised normal world can get the secret data that gets echoed back from the authorized server. To prevent such data leak from the authorized server, the SSL TA encrypts the user's secret twice. The secret value is first XOR with a one-time key generated from the TEE random device. Because SSL TA encrypts the one-time key using SSL Write key that is protected by the secure world (1st security property), the normal world cannot know the one-time key. The server will XOR the secret with the one-time key only when the server treats the data as secret. The website will post the one-time key encrypted secret value to any public field when the server only treats the secret as data and the server will not decrypt the value. The server can decide not to echo the HTTP header back to prevent the normal world from getting the one-time key. It prevents the one-time key from leaking out of the server.

To prevent the normal world from using downgrade attacks [9, 14, 15, 35] for MITM, the SSL TA disables weak cipher suites and vulnerable SSL versions. The SSL handshake will fail if the normal world rollbacks the cipher suites or the SSL version because the SSL encryption is done in the secure world. The SSL TA rejects the use of a weak cipher suite or a vulnerable SSL version to establish the SSL connection. The repeatable Initial Vector (IV) or predictable IV are subject to the chosen-plaintext attack [21], which assumes that the attacker can choose random plaintexts to be encrypted and obtain the corresponding ciphertexts. To prevent the normal world from manipulating the IV, the secure world always randomly generates the IV and uses it to encrypt the data.

When the TrustZone-protected secret is sent to the server using split SSL, the SSL session still shares the same security properties as normal SSL. The normal SSL has two properties (1) the session keys are tied to the server certificate, and (2) the PKI infrastructure can validate a fake certificate. Further more, the SSL TA checks the server common name (CN) before returning the encryption result back to the normal world.

TEE takes two inputs that are related to the server from the normal world: server domain and server certificate. In Table 5.3 we list all the attack vectors malicious apps can utilize and how our access control can defeat them. If malicious apps provide a malicious server certificate signed by the trusted CA, the common name of certificate would not be consistent with the user approved host name. The TEE SSL sending stage will refuse to return encryption data to the normal world. If a malicious app provides a malicious server certificate that has user approved common name, the certificate is definitely a fake one. The PKI logic in TEE verifies the certificate using pre-installed CA and rejects continuing SSL handshake. Our analysis result shows that we can defeat all the combinations of attack vectors and enforce the real user's intention.

Table 5.3: Malicious Server Attack Vectors Analysis

Server	Certificate	PKI Verify	CN Check	User Intent
Authorized	Real	Success	Success	Yes
Unauthorized	Real	Success	Fail	No
Authorized	fake	Fail	Sucess	No
Unauthorized	fake	Fail	Fail	No

5.3.2 Data Download Security Analysis

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

To prevent the server from leaking the confidential data, when the 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 all the servers that users sign in through the secure-world UI. The secure world refuses to save data that comes from the 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.

5.3.3 User Decision Security Analysis

In our design, users make the final access control decision when interacting with our system. The only choice for attackers to break the system is to trick users into making the wrong decisions. For instance, attackers can provide a fake hostname or spoof a fake secure-UI window in the normal world when the user types a password. Attackers can provide a fake confirmation message and ask users to confirm the wrong message. We consider all these possible attacks and conduct a usability evaluation in Section 5.4.3.

5.4 Evaluation

In this section, we evaluate our design from five aspects, namely, effectiveness, ease of adoption, usability, TCB reduction, and performance. We tested a variety of use cases using real-world applications and measured the ease of adoption for the developers. We tested whether or not users can make right decisions when interacting with our system. We also measured overhead imposed by our design and suggested future improvements.

5.4.1 Effectiveness: Applications

To demonstrate the effectiveness, we added new security features to open-source applications by making changes on the client side and server side (if needed). We modified seven open-source applications, including Elgg [7] and Drupal [5]. To measure the effectiveness in the case of closed-source apps, we modified the OS only for evaluation purpose.

Sensitive file upload. In this case study, we demonstrated how our work can enable normal-world apps to upload a TEE-protected file (e.g., a tax file, a medical record that is only needed by the server, not the client) to the authorized server without adding any app-specific code in the secure world. In contrast, DroidVault [66] requires the app-specific code in the secure world. We use an open-source app called `Seafile` to act as the tax e-file server. The `Seafile` client allows a user to enter a secret (e.g., tax account) via EditText and save it in a file. The app can then upload the tax file to its server using HTTP/SSL. We modified the `Seafile` app to allow the user to enter the secret file content using a secure EditText. The user types the file content using the secure keyboard, and the file content is saved in the secure world. The normal world gets a reference, which is saved in a file. When the user asks for the file to be uploaded to the server, the app issues an HTTP request using the normal-world file content (containing the reference). This triggers our modified HTTP engine, which traps into the secure world, where the content of the actual protected file replaces the file content in the HTTP request. The file upload request is then sent to the server. Our TruZ-HTTP and split SSL allow the file to be uploaded successfully to the `Seafile` server.

In this case study, we assume that the file content will not be sent back from the server to the normal world. Our design does not solve the sharing of the file once it reaches the server. Due to the HTTP header from TrustZone, the server will be able to identify that the file is uploaded from TrustZone. The server could deny download of this file until the request comes from TrustZone.

TrustZone-enabled Android authenticator. To demonstrate that our design can support the Account Manager framework (used to manage Android passwords), we wrote an authenticator app for Elgg. When a third-party app needs to login to our `Elgg` server, it will ask the Account Manager, which invokes the authenticator app's login activity. This activity uses a secure EditText to trigger our secure keyboard in the secure world. Once the user types the password, a reference is given back to the `Elgg` authenticator. The `Elgg` authenticator then sends the reference to the server through our modified HTTP and SSL. The password reference is saved by the Account Manager, which is not even aware that what it stores is not the actual password. Our system allows Account Manager to manage the authentication requests for third-party apps without storing the actual passwords in the normal world. Our design requires no change to the Account Manager framework.

Attested post. We installed `Drupal` on an Ubuntu server and modified the handling of the post content type to verify attestation. We used the `Drupal` Editor app [6] as a client. We modified the app to have an attested post functionality, which allows the user to confirm the post in the secure world before it is sent to the server. We utilized the proxy activity for this test to integrate with the confirmation TA. The app sends the secure world attestation along with the post message to the server. The `Drupal` server verifies the attestation before it publishes the post.

Protecting secrets. Apps written today need to protect different types of user's secrets. Our system allows developers to protect any text-based secret that can be typed in apps. We evaluated this by using seven different open-source apps and made minimal changes to the apps corresponding to the secrets that needed protection. This involved modifying the layout file containing the EditText corresponding to those secrets and configuring them as secure. The types of secrets protected in apps during the tests included login credentials and payment information.

5.4.2 Ease of Adoption

We evaluate the ease of adoption of our design by measuring how much effort developers need to make to add TrustZone support to their apps. We conducted the evaluation using both open and closed-source apps. For open-source, we downloaded both the client and server code from public Github repositories [8]. For closed-source, we downloaded apps from Google Play. To ensure their diversity, we downloaded apps from different categories, including shopping, traveling, productivity, finance, medical, business, food, etc.

We totally modified 7 open-source apps by either adding new features to them (e.g., attestation) or leveraging TrustZone to protect their existing features (e.g., login). We recorded the time spent on the modification and the number of lines of code (LOC) modified for each app. Table 5.4 shows the results. 1 LOC for TruZ-HTTP, 2 LOC for secure EditText, 4 LOC for secure confirmation.

As shown in Table 5.4, for apps to leverage our system to protect their login credentials, only 3 lines of code are modified on the client side and the time spent on making the changes is within an hour. For server-side changes, we need 4 lines of code to extract the secret data from the HTTP request. In case of attestation, the attestation logic may vary depending on what to attest. The overall change on the server side is less than 20 lines of code.

Table 5.4: Evaluation Results for Open-Source Apps

Test Case	**Client**	**Server**	**Time Spent**
Drupal Attested Post	4 LOC	20 LOC	1 hour
Elgg Attested Payment	4 LOC	12 LOC	30 mins
Elgg Authenticator	3 LOC	4 LOC	30 mins
Drupal Login	3 LOC	4 LOC	30 mins
GNUSocial Login	3 LOC	4 LOC	40 mins
Kandroid Login	3 LOC	4 LOC	30 mins
Redmine Login	3 LOC	4 LOC	30 mins
Owncloud Login	3 LOC	4 LOC	40 mins
Seafile Upload	3 LOC	4 LOC	50 mins

To evaluate whether the system works for apps from the market, we enabled closed-source apps to leverage the features of our system. To protect a user's secret in the secure world, we modify the apps to protect user's sensitive data, including passwords, credit card numbers, and files containing a secret. We repackaged the closed-source apps by configuring some selected EditText in their layout files, so when sensitive data needs to be provided by users, our secure keyboard is invoked and the data are typed inside the secure world. To protect users' confirmation in the secure world, we hardcoded the confirmation UI name (activity or activity containing dialog) and the corresponding message in a configuration file. The system uses the file to get a message (corresponding to a confirmation UI request) attested by the user in the secure world. To verify on the server side, we set up a proxy server to verify the attestation. The secure world shares the SSL keys with the proxy server, so it can intercept all the SSL traffic.

In our design, apps need to tell the underlying HTTP and SSL layers that the data to be sent to the server contains the TEE-protected secret, attestation message or attestation keys. Since it is difficult to modify the code of these apps, we hardcoded the information in a configuration file and let our modified HTTP engine obtain the needed information from this file rather than from the app. All configuration files and the proxy server are only for demonstration purpose. If we can modify the app, such files are not needed.

We collected 31 apps, including Chase, Github, Southwest Airline, Piazza, Priceline, Box, Poshmark, Listonic, Dropbox, MediaFire, Applebee's, Discover, Secure Cloud Storage, etc. We used 15 apps for TEE-protected login, 5 for TEE-protected payment, 2 for TEE-protected file upload, and 9 for attestation. Our results are shown in Table 5.5. All the experiments were successful, except two cases in the login category. The reason for the failures is not representative; they calculate HMAC of the HTTP request inside the payload. If we have the source code for these failed cases, we can easily make them work with our system.

Table 5.5: Evaluation Result for Closed-Source Apps

Test Case	**Login**	**Payment**	**Upload**	**Attestation**
Success/Total	13/15	5/5	2/2	9/9

5.4.3 Usability Evaluation

Methodology. To study whether users can make the right access control decision when using our system, we conducted two rounds of online survey to study the usability of our system. We wanted to test the three concepts that we introduced: (1) an LED to identify different worlds, (2) identify the correct hostname before typing the secret, and (3) confirm the message in the secure world before the approval. The survey follows *think-aloud protocol* that requires the respondents to write down what they think when making the access control decision. After conducting survey one, we found that people don't pay attention when interacting with the plain text confirmation message in the secure UI. Therefore, we conducted survey two, which was very similar to the survey one but we improved our secure UI to highlight the confirmation message in the secure UI.

We invited various college students who have a CS background as respondents for survey one. We recruited the respondents of survey two from Amazon Mechanical Turk (MTurk) where workers had at least 90% task acceptance rate. We conducted survey one in November 2017 and received a total of 93 valid responses. The respondents completed tasks by both typing secrets and confirming messages. We also conducted survey two in November 2017 and received a total of 161 valid responses in which 70 respondents typed secrets and 91 confirmed messages. Although the respondents of survey one are younger and have more computer related working experience in comparison with those of survey two, the overall performances of the respondents of survey two were better than those of the respondents of survey one because of the improvement of our secure UI in survey two.

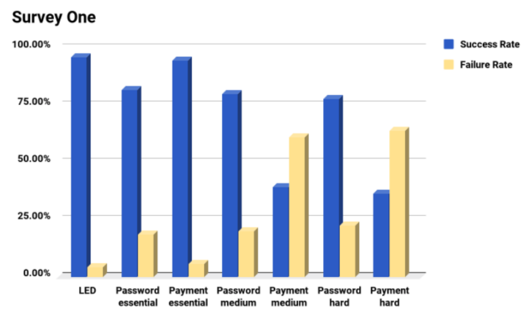
Survey Flow. In survey one, we provided a 3-minute video to define the secure world. We provided two use cases of our system (password typing and payment transaction) to explain how users should operate in the secure world. In survey two, we randomly assigned tasks to the respondents either to type a password or to approve a money transfer in the secure world. We provided a 2-minute video to define the secure world and used either password typing or a money transfer approval as an example to explain how to conduct the assigned task in the secure world. For typing a password in the secure world, we told respondents to pay attention to the LED light and the hostname before typing the password on the screen. For a payment transaction in the secure world, we told respondents to pay attention to the LED light and the confirmation message before approving the transaction on the screen. We then provided each respondent a phone screen image and asked them, "Based on the current screen, which mode are you currently interacting with?". For typing a password in the secure world, we then provided a secure UI for the Chase app with the LED light off and a correct hostname. We asked respondents, "Would you type your password on the current screen?". After respondents answered this essential question, they were randomly asked an optional question from three questions. The difficulty level of each question can be further divided into medium and hard. For the medium level questions, we asked respondents to type the password for Chase or Amazon. We provided a secure UI with an obvious wrong hostname, "aabbcc.com," with the LED light turned on. For the hard-level questions, we asked respondents to type the password for Amazon. We provided a secure UI with a phishing URL, "realamazon.com," with the LED light turned on. For all the questions, we asked them to provide open-ended answers to explain how they made the decision. We checked the answers and found their reasoning consistent with their decision, suggesting that the task of typing a password in the secure mode was understood correctly.

For payment transactions in the secure world, we provided a secure confirmation UI for the Chase app with the LED light on. The respondents were randomly asked an optional question from three questions. The difficulty level of each question can be divided into low, medium and hard. For the low-level question, we provided a secure confirmation UI with the right confirmation message and the LED light is on. For the medium level question, we changed the intended amount of $122.22 to be an obvious wrong amount of $10,000. For the hard level question, we changed the receiver's email from joe.banker@chase.com to a phishing email address of ali.banker@chase.com, which is not easy to differentiate with just a quick glance. For all the questions, we asked them to provide open-ended answers to explain how they made their decision. We checked the answers and found their reasoning consistent with their decision, suggesting that the task of approving payment transaction in the secure mode was understood correctly.

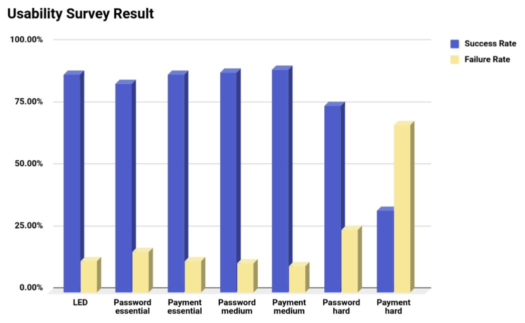
Improvement in survey two. Based on the payment transactions' feedbacks in survey one, in which the confirmation message is just a sentence, we reorganized the confirmation message and used the message highlighting mechanism. We highlighted the receiver's email and the amount. The new confirmation message is easier to read in comparison with a single sentence.

Data Analysis. We computed descriptive statistics of the quantitative data. We manually checked the open-ended answers by carefully reading through their answers. For the wrong decisions made by the respondents, we summarized the reason based on the open-ended answers. We wanted to know whether they could not use our secure mode or they could not catch the attack that we embedded in the survey questions.

Result. We now report respondents' quantitative results and qualitative feedback on access control decisions in both surveys.



(a) Survey One Result



(b) Survey Two Result

Fig. 5.7.: Usability Survey Result

In survey one, 70.89% of the respondents were male and 29.11% were female. In terms of age, 97.47% are among the age range of 18-34 and 2.53% are among the age range of 35-54. All respondents had either a degree or work experience in a computer-related field. All respondents were assigned to protect passwords in the secure world and to protect payments in the secure world.

Figure 5.7(a) shows the percentage of the respondents' access control decisions on our system. Respondents could make correct access control decision for all the essential tasks. For instance, 95.95% of the respondents could correctly identify which mode the device was in by identifying the LED light. 81.55% of the respondents didn't want to type the bank passwords when the LED light was off. 94.44% of the respondents could confirm the right transaction when the LED light was on. When we increased the task difficulty to the medium level, respondents could make right decisions for typing passwords but started to make mistakes for the payments. For example, 80.00% of the respondents didn't want to type Chase passwords for an obvious wrong hostname when the LED light was on, and 39.13% of the respondents didn't want to confirm the transaction with a wrong number when the LED light was on. When we increased the task difficulty to the hard level, the respondents still performed typing a password well but performed the payment confirmations in a less promising way. For instance, 77.78% of the respondents didn't want to type bank passwords for a phishing URL that was similar to the original URL, and 36.36% of the respondents didn't want to confirm the transaction with a wrong email address that was similar to the legitimate email.

In survey two, 60.25% of the respondents were male and 39.75% were female. In terms of the age, 64.59% of the respondents were among the age range of 18-34; 30.44% were among the range of 35-54; and 4.97% were among the range of 55 - 65. 57.14% of the respondents had either a degree or work experience in a computer related field. 42.86% of the respondents worked in a non-computer-related field. 43.83% of the respondents were assigned to protect passwords in the secure world, and 56.17% of the respondents were assigned to protect payments in the secure world.

Figure 5.7(b) shows the percentage of the respondents' access control decisions on our system. Respondents could make right access control decisions for all essential tasks. For instance, 87.58% of the respondents could correctly identify which mode the device was in by identifying the LED light. 83.95% of the respondents didn't want to type bank passwords when the LED light was off. 87.50% of the respondents could confirm the right transactions when the LED light was on. When we increased the task difficulty to the medium level, respondents could also make the right decision in the secure world. For example, 88.46% of the respondents didn't want to type bank passwords for an obvious wrong hostname when the LED light was on, and 92.86% of the respondents didn't want to confirm the transactions with a wrong number when the LED light was on. When we increased the task difficulty to the hard level, the respondents still performed typing password well but performed the payment confirmations in a less promising way for. For instance, 75% of the respondents didn't want to type bank passwords for a phishing URL that was similar to the original URL, and 31.82% of the respondents didn't want to confirm the transactions with a wrong email address that was similar to the legitimate email.

Next, we present respondents' qualitative feedback for all tasks of both surveys. The most common misunderstanding is that users think that the secure world is secure, and they are willing to type passwords without checking the host name or confirm transactions without reading the confirmation messages. Without users' assistance, our solution cannot work. This limitation also applies for all the TEE-based solutions whose objectives are to protect user-intended activities. Users also have trouble in identifying phishing attacks. For example, based on the feedback, many respondents did check both the LED and the message before confirming the transaction for the phishing email address. Respondents couldn't catch the difference in the phishing email address. We consider this as a limitation of our approach. We argue that our solution has tremendously increased the difficulty for attackers to launch attacks in real life because users are able to identify the obvious differences in the secure world based on the survey results, and getting a phishing email address for every receiver is a great challenge. The same limitation applies for the phishing hostname check. We argue that after two-minutes of instruction, people can guarantee above 75% accuracy. With more specific education on phishing URL, users will be able to make more accurate access control decisions. Existing work [53] also proves that users can identify a phishing URL with short-time education.

Conclusion. When users interact with our system, they know how to make the right decision to protect their intended activities in the secure world. Users can correctly identify different worlds using the LED light. Users can also make the right decision to only type the password for the right server in the secure world. In the case of the payment transaction in the secure world, users can understand how to confirm a message in the secure world and identify an obviously wrong confirmation message in the secure world. However, enabling users to make informed and appropriate choices is a hard research problem. We acknowledge that there is an entire research community of usable security researchers working on this challenging problem [44, 47, 61]. In our usability study, although only 32% of respondents correctly refused a confirmation with a spoofed email address, we believe that we can further improve the usability of our system by leveraging these usable research studies [44, 47, 61].

5.4.4 TCB Reduction

Our solution eliminates large swaths of code, compared with the existing secure network works [67, 80], which installed an individual network stack in the TEE. The result is summarized in Table 5.6. Take a linux network stack as an example – a separate TCP/IP stack would require 100K LOC and a separate NIC driver would require 1100K LOC.

Our TA code is only 1093 LOC and our solution only requires the SSL crypto library in the secure world for secure server communication.

Table 5.6: TCB Reduction

Component	**Existing TCB size**	**Our TCB size**
TLS	70K LOC	45K LOC
TCP/IP stack	100K LOC	0 LOC
NIC driver	1100K LOC	0 LOC

5.4.5 Performance Evaluation

In this section, we present the performance evaluation result for each major component in our design.

We also designed an experiment to evaluate the overhead of TruZ-HTTP plugin and split SSL design. We measured the overhead (average of 20 trials) caused by our design. Our evaluation was conducted based on Google, Amazon, and Facebook web servers. The average overhead of TruZ-HTTP plugin is 6.3 ms. The split SSL adds 304.3ms to each HTTPS request. Although this seems to be high, we need to keep in mind that the overhead is only incurred if an HTTPS request contains TEE-protected data (e.g. passwords, credit card number). If not, then there is no overhead. Therefore, this cost is more of a one-time cost.

Our design requires 10 rounds of interaction between two worlds for every HTTPS request. Each transition between these two worlds only takes 8 ms. The reference replacement overhead in SSL is almost 0 ms because they are all TEE memory access. The average overhead of split SSL is 304 ms. We further analyzed the cause of SSL overhead. When we port the SSL crypto library from Android to OPTEE, we can only port its C code, not the assembly code, because the boringSSL's assembly code is not compatible with the current OPTEE compiler. Based on our experiment in Android, C version of crypto has on average 1X time overhead in comparison with the assembly verison. Furthermore, the Android compiler optimizes code better than the OPTEE compiler. We observed on average 3X time overhead to run crypto in the OPTEE in comparison with running in the Android. Therefore, doing encryption inside OPTEE is significantly slower than doing so in Android. We believe that the encryption time can be reduced once these issues are solved.

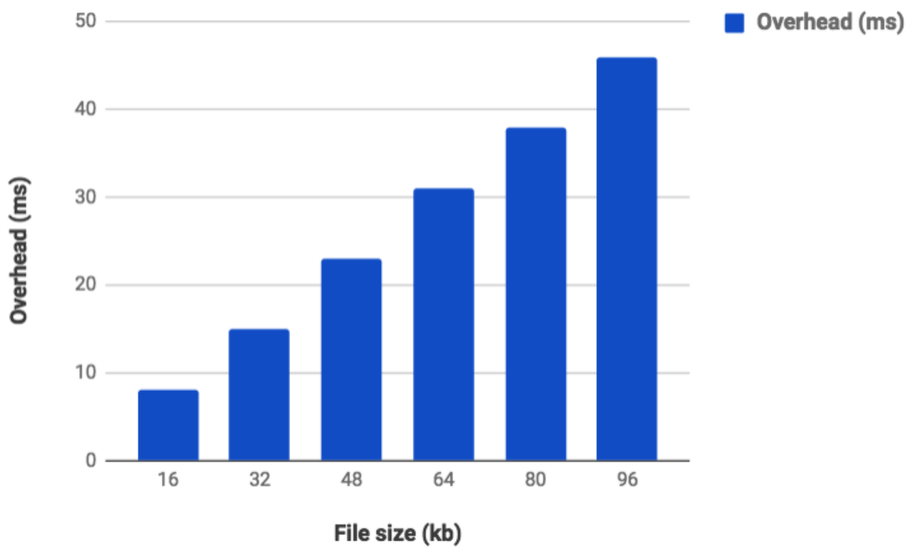


Fig. 5.8.: Data Download Performance Overhead

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 sizes (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 to calculate the overhead:

overhead = TZ download time - normal download time

Figure 5.8 summarizes our performance result. The main overhead is caused by the world switch of each TLS record decryption.

5.5 Summary

In this paper, we proposed a design to integrate TrustZone with Android that allows apps to leverage TrustZone to protect user's interactions and protect sending the user's secret to the authorized server. We implemented TruZ-HTTP and Split SSL components by modifying Android and OPTEE OS. We tested our system on the HiKey board. Through real-world evaluation, we have shown the effectiveness and ease of adoption of our design.

6. TRUZ-VIEW

Smartphone users heavily depend on User Interface (UI), the primary communication path between users and mobile devices, to conduct all sorts of daily activities. However, heavy reliance on this single interface has resulted in substantial security concerns due to mobile software stack control of the UI. Unfortunately, CVE results of mobile OSes are not positive [20]. The mobile OSes cannot prevent untrusted code embedded in applications from running, which then leads to a broad attack surface. Once the mobile OS is compromised, the last defense of UI is gone, and UI is then controlled by malware. What's worse, malware can spoof actions on behalf of users without their consent. To secure the communication between users and mobile devices, TrustZone UI provides applications an isolated user-mobile communication path to protect the sensitive UI interaction.

As described in Chapter 4, there are mainly two directions to design TrustZone UI (see Figure 6.1). The current design of TrustZone UI solutions [17, 67, 93] is based on self-contained TEE model and supports a fully functional UI stack in the secure world. These UI solutions support TAs to build isolated UI in the secure world. Past research works [16, 63, 88, 100] built various UI protection mechanisms (i.e., TAs) on top of current TrustZone UI solutions [17, 93]. Because of the complexity of UI stack (see Figure 6.2), the size of it is large and thus contribute a large TCB in the secure world.

We intend to reduce the TCB's size of TrustZone UI. The functionalities of the secure-world UI stack, according to our observation, can be further divided into three

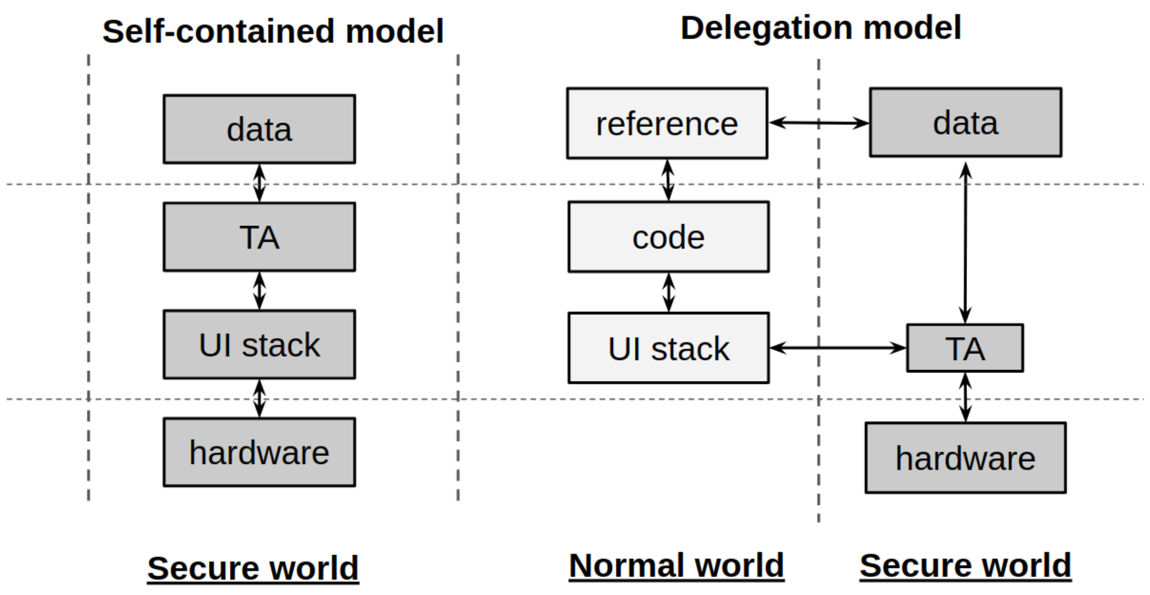


Fig. 6.1.: TrustZone UI Design Models

parts – UI development, UI services, and UI interaction (see Figure 6.2). When developing UI, developers construct the UI layer by layer, meaning that developers can first define the background layout (layer 1) and then place buttons (layer 2) on top of the layout (layer 1). Once UI layers are developed, UI services propagate touch events and compose screen content based on the multilayered UI structure defined by developers. These two components are both included in the self-contained UI model in the secure world. Because of the complexity of this multilayered UI processing, the TCB's size of UI development and UI services is large. Our goal in this dissertation, thus, is to reduce the TCB's size for these two components.

The main objective of TrustZone UI solution is to secure UI interaction, which involves displaying sensitive data on the screen and taking sensitive input from users. When users interact with the secure-world UI, they solely interact with a two-dimensional surface on the screen. On a two-dimensional surface, the process of protecting sensitive data display is equivalent to protecting a region on an image; the protection of users' sensitive input is equivalent to protecting users' touch coordinates on the screen. We can significantly reduce the size of TCB if the secure world only needs to protect the UI as a two-dimensional surface and leaves the three-dimensional UI processing in a rich OS.

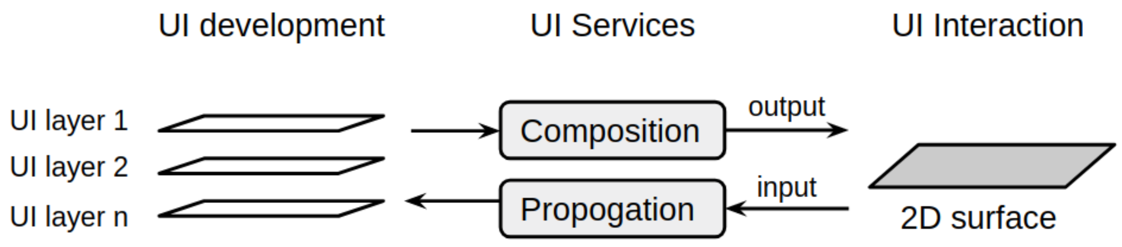


Fig. 6.2.: TrustZone UI Stack

The main idea of our model is to delegate UI development and UI services to a rich OS (see the right side of the Figure 6.1). The normal world conducts three-dimensional UI processing without accessing any sensitive data. The output from 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 all users' touch coordinates on the screenshot in the secure world. The protected data will never be leaked to the normal world. The new design significantly reduces the size of the secure-world UI stack by converting three-dimensional UI processing to two-dimensional UI processing.

This chapter systematically studies design challenges when applying delegation model to protect UI. Our new design based on delegation model is able to protect UI interaction when mobile OSes are compromised. Moreoever, we performed a thorough security analysis to prove the security of the delegation UI model and evaluated our system by using real-world applications.

6.1 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.

6.1.1 Threat Model

The user of mobile devices 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 Trusted Applications and TEE OSes 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 users' confidential data and verify the integrity of users' requests.

6.1.2 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.

6.1.3 UI Design Models Design Trade-off

In Chapter 4, we did a high-level comparison between the self-contained model and delegation model. To better understand characteristics of different design models to secure the UI, we systematically compare the design trade-off between two models in three aspects, namely security, system design, and application impact.

Security. TCB's size and isolation boundary are two essential attributes to measure the security of TrustZone UI. The self-contained model requires a separate UI stack to support TAs to write UI logic, which requires a large TCB in the secure world. By contrast, the delegation model moves the majority of UI stack in the normal world and requires a small TCB in the secure world. As a trade-off for TCB's size, the self-contained model provides a complete isolated UI stack for Trusted Applications to build UI, while the delegation UI model requires the untrusted normal-world UI stack to build UI. Therefore, the UI design that applies the delegation model has to answer this challenging question.

System design. Code reusability and modularity are two critical measurements for TEE system design. First of all, code reusability implies that UI logic developed for one world can be reused for another world. The UI developed based on the self-contained model always suffers low code reusability because different OSes have different UI stacks and APIs. 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 high code reusability. Secondly, modularity suggests that the update of UI stack for one world should not affect the other world. The self-contained model can complete the update of UI stack independently because two worlds' UI stacks are entirely separated. As a trade-off for 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 impact of TrustZone design on applications. Firstly, transparency is how many efforts application developers need to make to integrate TrustZone UI solutions 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 TrustZone UI. Moreover, putting application-specific UI logic in the TEE is not secure. The delegation UI model has high transparency to applications because our model reuses the normal-world UI development environment to develop the secure-world UI and does not put any application-specific UI logic in the TEE. Secondly, consistency means whether users have the same UI experience across worlds. Users usually endure inconsistent user experience when interacting with the self-contained UI model because 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, rich usability refers to whether the secure-world UI supports rich UI functionalities such as animation and UI extensible services such as autocomplete and spell checker. As a trade-off for TCB's size, the delegation model cannot 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 the UI design principles. When we apply the model to design TrustZone UI, these principles will help us preserve aforementioned properties: (1) maintain a small TCB in the secure world and reuse the normal-world UI stack for non-sensitive operation as much as possible; (2) from an engineering perspective, reduce the coupling between the normal-world UI stack and the secure-world TAs; (3) require a minimum effort for applications to integrate with TrustZone and for system to update UI stack; (4) maintain consistent UI experience across worlds.

6.1.4 UI Design Challenges

In this section, we discuss UI design challenges when applying the delegation model. The secure world loses many capabilities during this TCB reduction process because our design model pushes the majority of 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. Our design model delegates UI development to the untrusted normal world. All 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 multi-layered information of UI because the delegation model solely protects a 2-dimensional image in the secure world while the UI layer information is processed in the normal world. However, some of the multi-layered information of UI is important to define how to protect UI display and UI input. We need to find a way to recover the **lost UI layer information** in the secure world.

6.2 Idea

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

6.2.1 Splitting the UI Rendering Process

Our main idea of protecting the secure-world UI is to split the normal-world UI rendering process at the last step and move this step into the secure world. We categorize all existing TrustZone UI solutions as splitting the UI rendering process at different layers. The Figure 6.3 is an abstraction diagram of the splitting design options. The first option [17, 67, 93] is to split the UI rendering at the application layer and to move all three layers into the secure world. The second option [100] is to split the UI rendering process at the framework level. Such a splitting option leaves 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. Our design follows the same approach described in the paper [100] to overcome the binding challenge. Although these two existing splitting options provide UI security measures, such splitting approaches require a separate UI stack to support the secure-world UI.

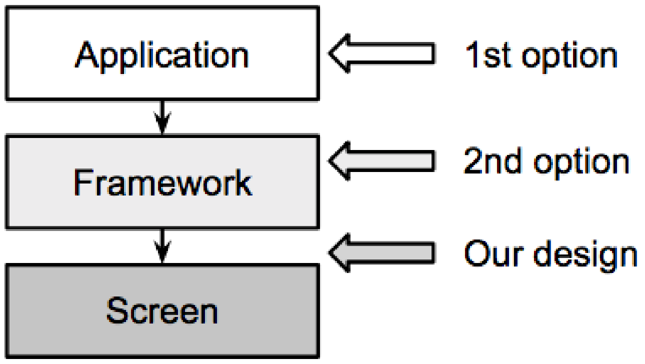


Fig. 6.3.: UI Rendering Split Options

We suggest splitting UI rendering process at the last step. The normal world produces 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 display of sensitive data by overlapping the protected data on top of the screenshot and protects users' input by keeping touch coordinates inside the secure world. Our splitting option allows us to protect the UI interaction on a 2-dimensional surface.

6.2.2 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 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 a 2-dimensional surface to overlap the sensitive data on top of an image, thus protecting a 2-dimensional surface is sufficient to protect the display of sensitive data.

Next, we describe why securing users' sensitive input on a 2-dimensional surface is adequate. First of all, the initial form of touch events 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 the 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 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 6.5 to prove the security of the 2-dimensional UI protection.

6.2.3 Recover UI Layer Information

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

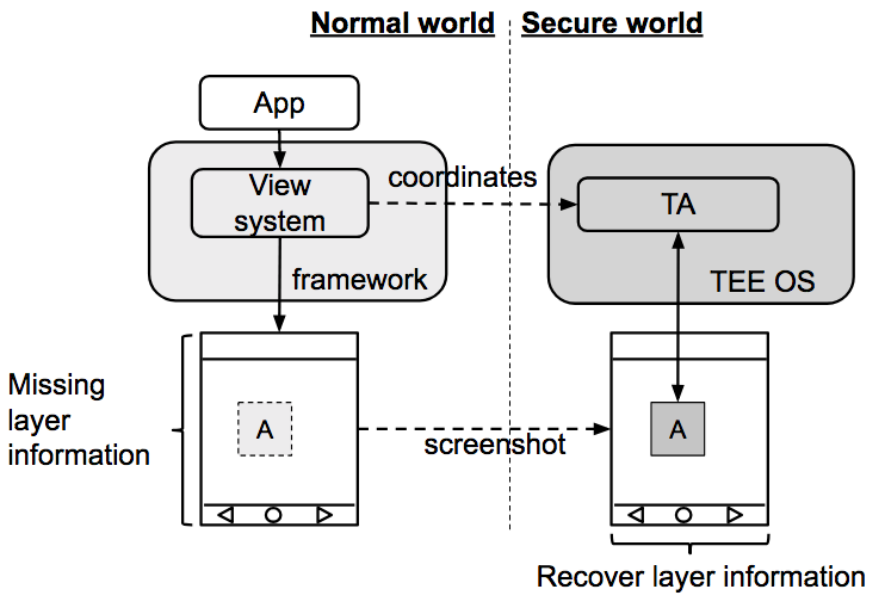


Fig. 6.4.: Recover UI Layer Information

Our main idea to recover the multi-layered information is to let the view system send views' coordinates to TrustZone, as shown in Figure 6.4. We observe that UI layer information is initially defined by UI views that are basic building blocks to construct UI in the normal world. Our design allows developers to label certain views as TrustZone-protected when they develop UI. We allow application developers to follow the same workflow to develop UI and create a TrustZone tag in the view system. Our modified view system sends all marked view coordinates to the secure world. Based on these coordinates, our design can recover UI layer information on a 2-dimensional surface. We have conducted a thorough security analysis in Section 6.5 to prove that the normal world cannot misuse 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, and (3) integrity-preserved interaction. In Section 6.3, we will describe our UI building blocks design in detail.

6.3 UI Design

In this section, we discuss our UI design based on three categories: (1) confidential display, (2) confidential input, and (3) integrity-preserved UI interaction.

6.3.1 Confidential Display

Application developers can protect users' confidential information displayed on the screen by adding TrustZone-enabled views. We refer to such views as *TrustZone-enabled UI building blocks*.

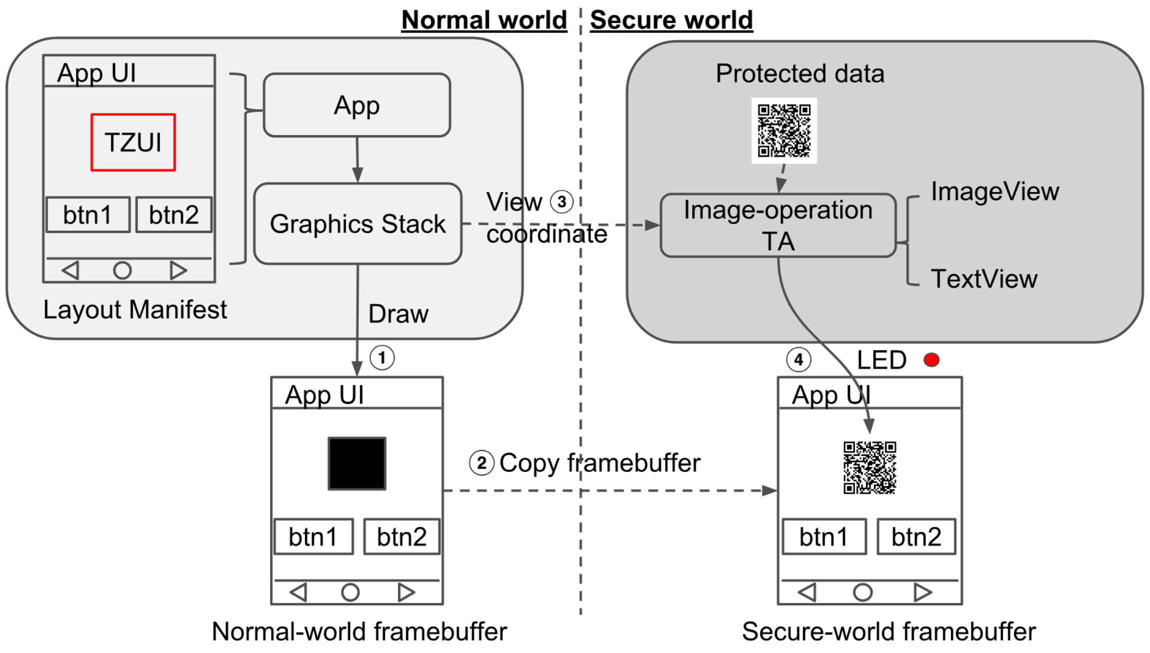


Fig. 6.5.: Confidential Display Design

The design of confidential display enables the normal-world graphics stack to recognize the TrustZone-enabled views and to work with TrustZone to display the protected data stored in the secure world. The normal-world graphics stack composites app's UI into an image but leaves the TrustZone-enabled views' area blank, as shown in Figure 6.5 ❶.

Users first see the normal-world UI, as shown in Figure 6.6 WeChat example. To view the TEE-protected data, users click on the TrustZone-enabled view 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 6.5 ❷. Our design further transfers the coordinate and the size of TrustZone-enabled view to the secure world, as shown in Figure 6.5 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 in the screenshot and displays the complete UI to users, as shown in Figure 6.5 ❹. Throughout this process, users see a familiar app's UI with the protected data filled in. We use the WeChat barcode payment UI to illustrate our UI in the secure world (Figure 6.6 WeChat secure world). Users scan the barcode in the secure world and can click the back button to switch the control of screen to the normal world. When in the normal world, the protected data disappears and the TrustZone-enabled view area is shown as blank on the screen (Figure 6.6 WeChat normal world). In Section 6.5.1, we explain why users' confidential information will not be leaked during the process of confidential display.

We totally developed two types of confidential display UI building blocks, namely ImageView and TextView. Developers can simply embed confidential display UI into their

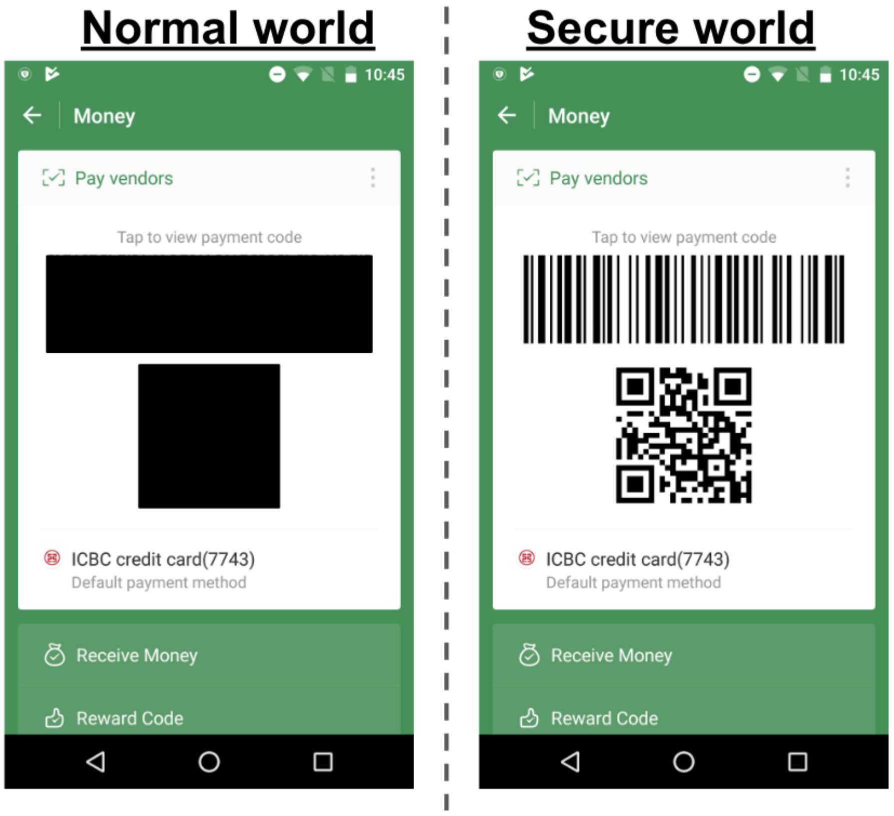


Fig. 6.6.: WeChat Barcode Payment UI

applications using the code in Listing 6.1. The confidential display only requires the secure world to perform image operation (cheap operation) to render the protected data. 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 have discussed our implementation for data downloading and reference management in Section 5.2.3.

Listing 6.1: Application change for confidential display

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

6.3.2 Confidential Input

Application developers can protect users' confidential input by embedding a TrustZone-enabled 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 their smartphones 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' keystrokes inside the secure world. To give feedback on users' inputs, an app, which uses the TrustZone-enabled keyboard, needs to display all 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 app's UI including the TrustZone-enabled keyboard and the EditText into an image. When the normal world displays the UI, our design takes a screenshot and then sends the screenshot to the secure world. Then the secure world takes over the control of screen and lights on a LED to notify users that they can securely enter their confidential input on the screen. As explained in Chapter 5, our access control model requires users to make the final decision. Users need to confirm the hostname of server inside the secure world before typing their secret. Our design displays the hostname on top of the screen.

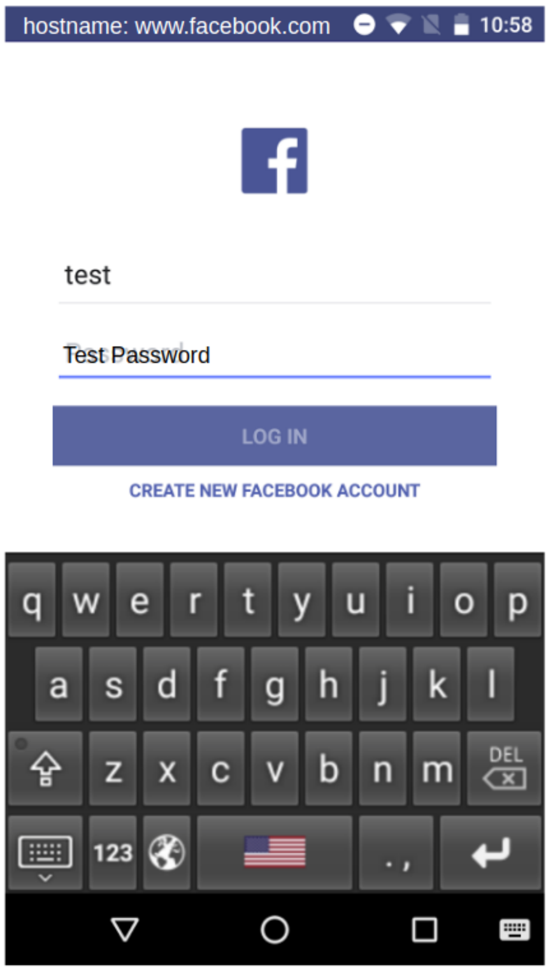


Fig. 6.7.: Facebook Password Login

Users see the same app's UI in the secure world, as shown in Figure 6.7. To allow the secure world to handle users' clicks on the keyboard, our design sends the coordinates of TrustZone-enabled 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 keyboard in the secure world. When users type 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.

Developers can simply embed a confidential input UI into the app using the code in Listing 6.2. We assume that users' input is processed only by the server, not by the client. We have explained our implementation of secret transmission in Section 5.2.3. In Section 6.5.2, we explain why the users' confidential information will not be leaked during the confidential input process.

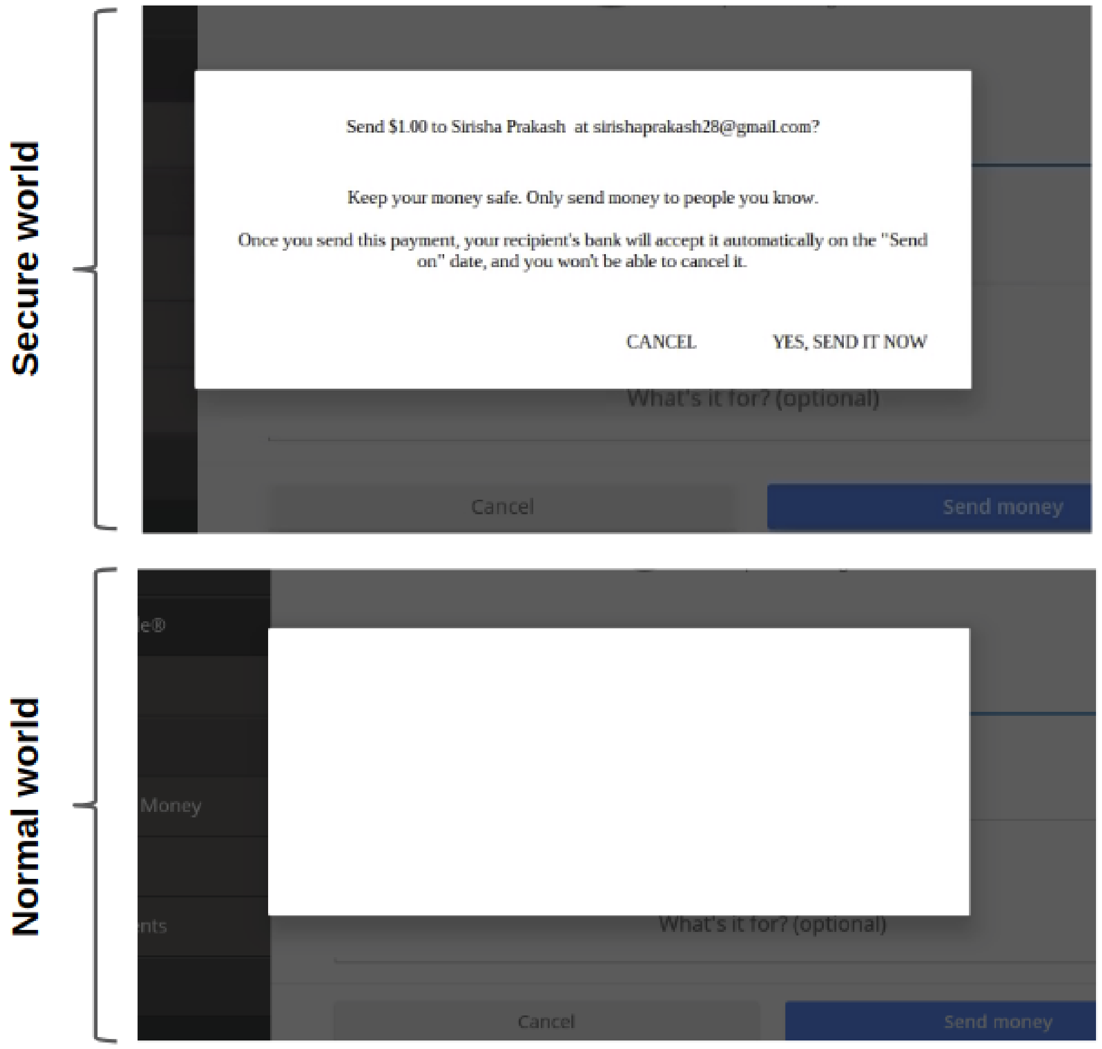
Listing 6.2: Application change for confidential input

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

6.3.3 Integrity-preserved UI Interaction

Application 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 users and to request 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 6.8. To display the confirmation message, app developers can embed a TrustZone-enabled TextBox. To obtain user's decisions, application developers can embed



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.

Fig. 6.8.: Chase Payment Confirmation Dialog

two TrustZone-enabled buttons, one for a positive decision and the other for a negative decision. Our design recognizes these specialized views and asks the normal-world graphics stack to composite app's UI including the TrustZone-enabled view region, as shown in Figure 6.9 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 views' coordinates, the confirmation message, and properties of buttons (i.e., positive or negative) to the secure world, as shown in Figure 6.9 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 6.9 ❸. Two options guarantee the integrity of the text: (1) TrustZone provides the text content and (2) servers attest 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 6.5.3.

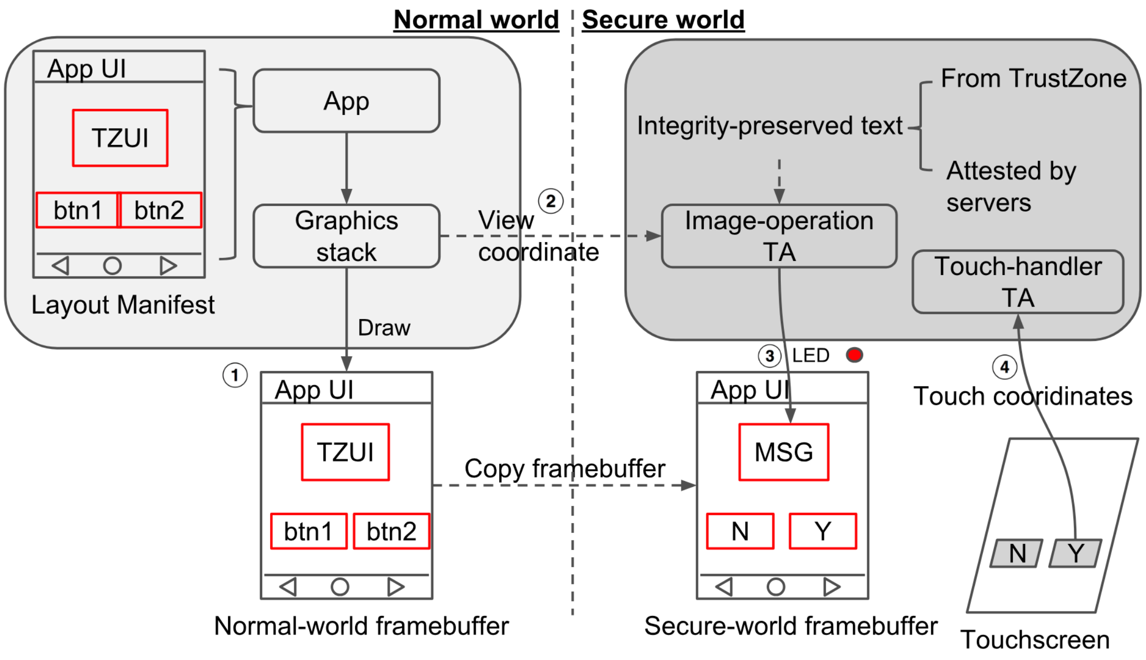


Fig. 6.9.: Integrity-preserved UI Interaction Design

Users see familiar app's UI with the integrity-preserved data filled in. When users confirm the message, our design generates an attestation of the confirmed message inside TrustZone, as shown in Figure 6.9 ❹. We have explained how TrustZone and servers exchange an attestation key in Section 5.2.3. Once users click 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 an app using the code in Listing 6.3. In Section 6.5.2, we explain why users' confidential information will not be leaked in our injecting click design decision.

Listing 6.3: Application change for integrity-preserved UI

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

6.3.4 Hardware Design

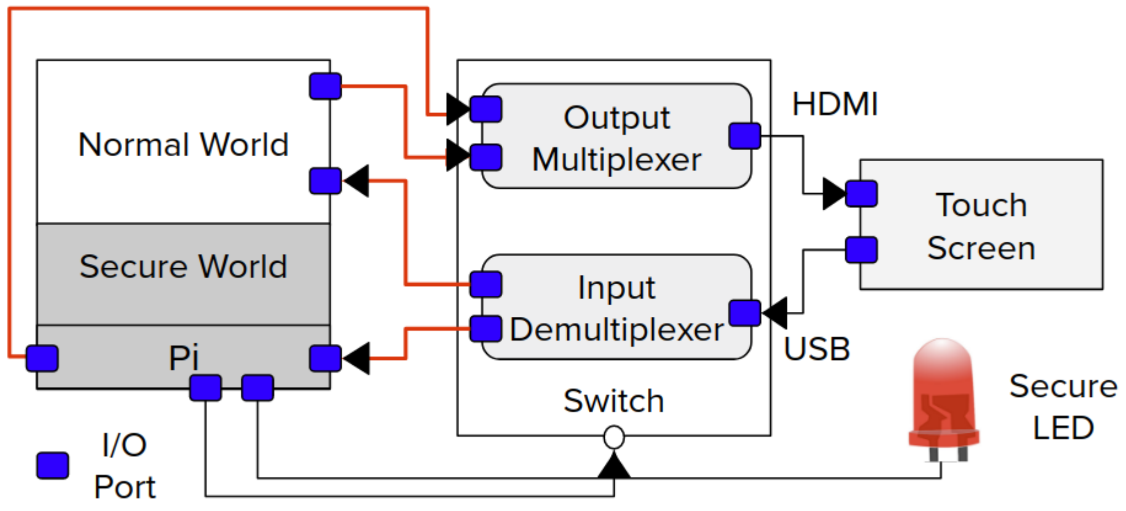


Fig. 6.10.: Hardware Design

Our hardware platform runs Android 7.0 in the normal world and OPTEE [75] in the secure world. We built our prototype using the HiKey board as our base platform [10]. We used a TFT LCD panel as the screen. The screen uses HDMI interface for display and USB interface for controlling touch.

As shown in Figure 6.10, to allow the secure world to drive screen, we introduce an additional board (i.e., Raspberry Pi) controlled by the secure world. The secure world communicates with Raspberry Pi through serial communication (i.e., UART) and the secure world has the exclusive control of UART. We achieve screen isolation at the circuit level. We connect the screen I/O to the multiplexer/demultiplexer. The multiplexer takes HDMI signal from both worlds and outputs the signal from one of worlds to the screen. The demultiplexer takes the touch input from screen and gives it to one of worlds. The secure world controls the switch of multiplexer/demultiplexer. Each world has separate I/O ports that are connected to multiplexer/demultiplexer. The 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, the LED indicator, and the UART. Our hardware implementation is shown in Figure 6.11.

The main reason to use a separate Raspberry Pi is that OPTEE does not have a series of device drivers for display and input. Developing such drivers requires accessing design reference documents of the hardware platform (e.g., HiKey). However, 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 all driver stacks for their own hardware. If vendors adopt our design, they can eliminate the Raspberry Pi in our design by directly deploying 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 functionality in Raspberry Pi. We consider that our hardware design provides a reference for research communities that desire to conduct a hardware-related research but are discouraged by device-driver accessibility in the secure world.

We provide a YouTube video to demo how our system can protect payment barcode [26].

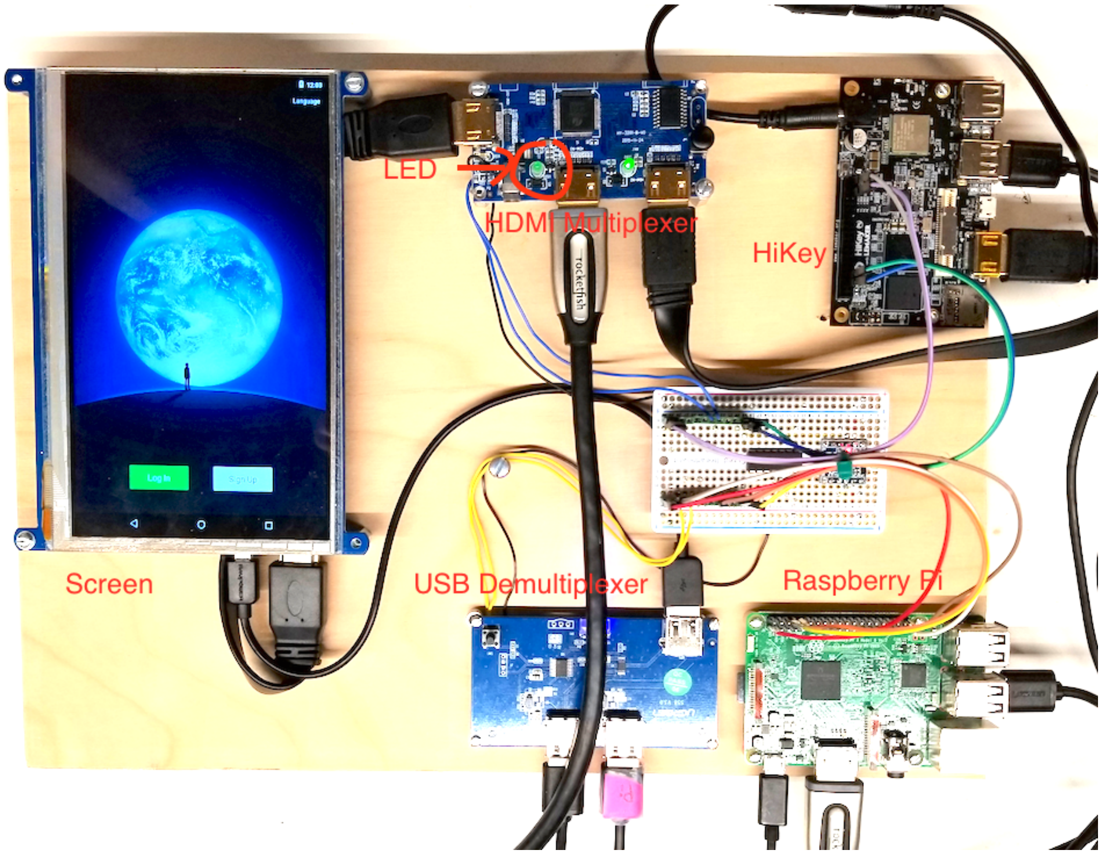


Fig. 6.11.: Hardware Implementation

6.4 Implementation Details and Case Studies

In this section, we describe our design details and use cases of our UI building blocks that we designed for TrustZone. Developers can adopt our solutions with few changes to their applications.

6.4.1 Confidential Display

We modify the view system to recognize the special labels in manifest file and the `onTouchEvent()` in view classes to handle the user's click on the TrustZone-enabled view region. When users click on the TrustZone-enabled view, our design transfers the view's coordinate and the screenshot to the secure world. Our approach to taking a screenshot is to copy the normal-world `framebuffer` to a piece of secure-world memory. The framebuffer is a piece of memory that contains a bitmap that displays on the screen. Each pixel in the bitmap is represented by RGBA (red, green, blue, alpha) values. We implemented a *bitmap-operator TA* to convert protected data into RGBA values on the bitmap. Our design further divides protected data into image and text. By default, the image data is a bitmap or can be easily converted into a bitmap. Thus the secure-world TA can fill in the protected image on the TrustZone-enabled view area. To render the text on the screenshot, our design leverages a tiny font rendering library [25], which can convert the text into a bitmap. Our bitmap rendering design has low performance overhead and does not rely on any sophisticated graphics logic. The design can be easily deployed into every TEE environment.

TrustZone-enabled ImageView and TextView. Apps usually display text-based secrets, such as credit card number and SSN, and image-based secrets, such as ID card and payment barcode. We designed the TrustZone-enabled ImageView and TextView to help apps secure the rendering of these secrets. We explain the workflow of ImageView and the TextView works the same way. Taking WeChat payment barcode UI in Figure 6.6 as an example, developers embed a TrustZone-enabled ImageView in the app's UI by adding a `tzSecure` flag under the ImageView tag in the layout manifest file. The source attribute of the image serves as the reference to the TEE-protected barcode stored in TrustZone. When users want to display the barcode, they click on the ImageView. Our modified `onTouchEvent` function for ImageView handles this click and switches the screen to the secure world. The secure world displays the complete payment barcode UI. After that, users can click the back button to switch to the normal world. In our design, developers do not need to insert code into the secure world to secure the secret rendering.

6.4.2 Confidential Input

UI design for confidential input. We override the `onWindowFocusChanged` in the view classes to recognize these specialized views. Our modified views transfer the views' coordinates, layouts of views and the screenshot to the secure world when such views appear on the screen. The secure world obtains the exclusive control of touchscreen to protect the click event generated from the hardware and we have described how we secure the hardware using TrustZone in Section 6.3.4. We implemented a *touch-handler TA* to handle user's clicks. The secure-world TA extracts the touch coordinates from the click.

We identify two types of common confidential input use cases, namely EditText with KeyboardView, and SignaturePadView. Our current touch-handler TA handles two types of touch action: 1) ACTION_DOWN and 2) ACTION_MOVE. Each action is used to support one particular UI building block.

TrustZone-enabled EditText and KeyboardView. We designed the TrustZone-enabled EditText and KeyboardView for confidential input through a keyboard. In Android, keyboard is an independent application called Input Method Editor (IME). Android prebuilds a KeyboardView as a baseline for IME developers to draw their customized keyboard layouts. To declare a TrustZone-enabled keyboardView, developers set the `tzSecure` flag in the KeyboardView in the layout manifest file. We override the KeyboardView `onWindowFocusChanged` callback function to work with TrustZone. When the keyboard appears on the screen, the `onWindowFocusChanged` will be triggered and the keyboardView will send the view coordinates and keyboard-layout XML to the secure world. Android requires each Keyboard app to have a XML layout file and our design reuses this file to gain the layout of the keyboard in the secure world. To display users' inputs in the secure world, we implemented a TrustZone-enabled EditText. Our modified *InputMethodManagerService* can use our TrustZone-enabled keyboard when the EditText `tzSecure` flag is true. When users click the TrustZone-enabled EditText in the normal world, the TrustZone keyboard will come out and the LED light will be on. The touch-handler TA handles the user's click based on the keyboard layout and informs the bitmap-operator TA which character is pressed on the keyboard. The bitmap-operator TA then renders the corresponding character to the EditText field. When users finish typing, the input is saved in the secure world and the IME app gets just a reference to the stored confidential input data. Android has already had a standard flow to return data from IME to the EditText. Our design follows this existing flow and returns a reference to the EditText for apps to use.

TrustZone-enabled SignaturePadView. We designed the TrustZone-enabled SignaturePadView to protect users' hand-drawn signature. In Android, developers commonly customize the view class and build their own signature pad. We follow the same development pattern and create an additional option in the view class for the TrustZone-enabled SignaturePadView. We overrided the SignaturePadView `onWindowFocusChanged` callback function. When the signature pad is shown on the screen, the secure world listens to the ACTION_MOVE on the SignaturePadView region. When the user signs on the screen, the secure-world TAs record the touch coordinates and render black dots of the touch coordinates on the screen. We create an API in the view to allow the app to get the reference to signature coordinates.

6.4.3 Integrity-preserved UI Interaction

Integrity-preserved UI design

We modify the view system to recognize these special marks in the manifest file. Because the integrity-preserved UI commonly requires multiple view elements to accomplish the task, our design allows the root view of activity to scan all children and identifies all the TrustZone-enabled views in the view hierarchy. All TrustZone-enabled-view coordinates, properties of view, the confirmation text and the screenshot are bundled together, and are sent to the secure world. To ensure the integrity of the displayed content, the bitmap-operator TA first cleans the view region by setting the region's color white and then fills in the text on the view. Users can approve or deny the message in the secure world. The touch-handler TA captures the user's consent on the TrustZone-enabled views. We identify serveral UI building blocks for integrity-preserved UI, such as Confirmation Dialog or Activity, Pattern Locker, PIN pad, Password Locker. We explain the design details of the first three cases and others follow the same design patterns.

TrustZone-enabled Dialog and Confirmation Activity. Apps usually take users' consent, such as approval of monetary transaction, through a Dialog or a Confirmation Activity. Apps can gain the user's consent from the secure world and send the proof of user's decision to the app server. We designed the TrustZone-enabled Confirmation Dialog and the Confirmation Activity to protect users' consent. We explain the workflow of Confirmation Activity and the Dialog works in a similar way.

TrustZone-enabled Confirmation Activity allows developers to put as many TextViews as they need to construct the confirmation UI. To prevent the normal world from fooling users by adding phishing content, the bitmap-operator TA blurs all the non-sensitive region on the screenshot by applying blur algorithm. Users thus see only the message that needs to be confirmed in the TrustZone-enabled activity. We assume that the server and TrustZone have exchanged an attestation key. When users click on the OK button to confirm the message, the touch-handler TA handles the touch on the positive button region and generates an HMAC of the confirmation message. Under the condition of multiple

TextViews in the Confirmation Activity, the TA combines all the text into a single attestation message based on the order of left to right and top to bottom on the screen. To return the attestation result to the normal world, we modify the `MotionEvent` class to carry the attestation from the secure world. We inject the user's click event to the normal-world OK button. The developers can override the `onTouchListener` and extract the special MotionEvent injected by our design. Users can also click the cancel button. In such case, no attestation is returned.

TrustZone-enabled Pattern Locker. In Android, app developers can request `KeyguardManager` to authenticate the user before an app continues to the next step. Android `KeyguardManager` provides the option to set the description of the action on the LockPatternView so that users can confirm the action on the LockPatternView before the app moves forward.

We allow both client and server developers to use secure LockPatternView. First, client developers can use our LockPatternView in a similar way of the Confirmation Activity. LockPatterView confirms the user's consent and also allows the app to check the authenticity of the user in the secure world. Second, app-server developers can ask the secure world to authenticate the user before it shows any protected data. For example, the secure world authenticates the user by using LockPatternView before showing the ID card. The server can further provide the metadata of the protected data to help users check the identity. For example, the server can provide the description of the payment barcode to help users identify which barcode is shown in the secure world because users cannot understand the actual meaning of the barcode by just seeing the barcode itself.

6.5 Security Analysis

In this section, we present the security analysis of our TrustZone UI design. Our design can guarantee the confidentiality of the UI display and the UI input, and the integrity of the UI interaction when the OS is compromised. We also analyze the security of the secure downloading feature that we built on top of the Split SSL [100]. 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.

6.5.1 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 the 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 6.3.4, 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.

6.5.2 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 6.5.1, 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 the users into typing the secrets. Although the normal world can fake the same UI look and trick the users to type the secrets, we use an LED light to indicate to users if they are typing in the secure world or not. The secure world obtains the exclusive control over this LED, so the normal world cannot control the indicator. The existing work [100] 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 the buttons.

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

6.5.3 Integrity-preserved UI Security Analysis

The objectives of the adversary include fooling the 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 view, the TA cleans the secure view region to prevent the normal world from displaying untrusted contents in the secure view region. The TA fill 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 the views to the secure world. For example, the normal world can swap the positive and negative button coordinates and send them to the secure world. However, the secure world renders all the texts including the 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 keys. Furthermore, we append a nonce when computing the attestation to avoid replayability.

6.6 Evaluation

In Section 6.3, we have demonstrated the applications of our design (i.e., ImageView, KeyboardView, etc.). In this section, we further evaluate our design from three aspects, namely, 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 [10], which runs both Android 7.0 in the normal world and OPTEE OS [75] in the secure world.

6.6.1 TCB Reduction

Our solution eliminates large swaths of code, compared with the existing TrustZone UI works [17, 67, 93], which installed an individual UI framework in the TEE. The comparison result is summarized in the table 6.1. Take a standard UI stack as an example - Xlib [34] is used for the UI display and the UI input. Xlib has more than 100k LOC. The widget toolkit called tk [31] that runs on top of Xlib contains half a million LOC.

We installed a modified font rendering library [25] 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 the display and the input. All existing works [17, 67, 93] also installed drivers in the TEE to keep a robust TCB.

Table 6.1: TCB Reduction

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

6.6.2 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 [8] 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 our TrustZone-enabled UI building blocks to eliminate the UI risks. For open-source apps, we count the LOC changes and modification time in order to integrate our solution into the app. For closed-source apps, we count the success rate of our integration. We initially implemented our solution on Android 6.0 and recorded the time to incorporated our solution on Android 7.0.

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 view resource ID 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 the repackaging.

For use cases, we let apps display various types of sensitive data in the secure world by using our ImageView and TextView. The apps can further use our LockPatternView to display the data identity or authenticate the 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 the results are shown in theTable 6.2. It takes fewer than 5 minutes to modify all the 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 the Dialog and the Confirmation Activity need 1 LOC to extract the attestation result from the `MotionEvent`.

Table 6.2: 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

Table 6.3: Evaluation Result for Closed-Source Apps

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

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 6.3. All the experiments were successful.

We migrated our design from Android 6.0 to 7.0. 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.

6.6.3 Performance

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

Table 6.4: Framebuffer Transfer Performance Overhead

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

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 6.4, the major impact is caused by our modification in the view system and by the memory access of the 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.

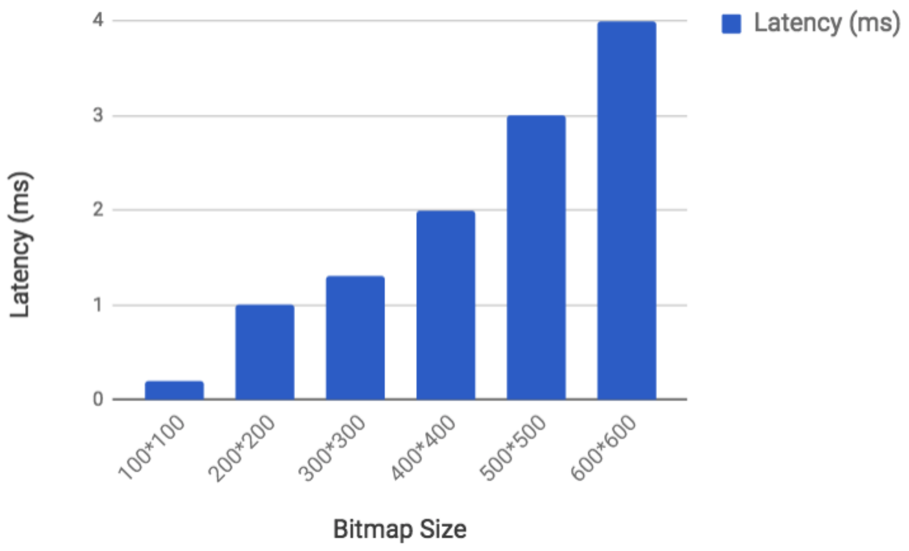


Fig. 6.12.: Image Rendering Overhead

Secure-world rendering overhead. Our image-operation TA renders the content on the secure-world framebuffer. We measured the performance of the bitmap operations for both image and text in OPTEE OS [75]. We measured the time needed to construct the final framebuffer for various image sizes, word lengths, and font sizes. Figure 6.12 shows the image rendering overhead with common image sizes from 100*100 to 600*600. The overhead is less than 4 ms. Figure 6.13 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.

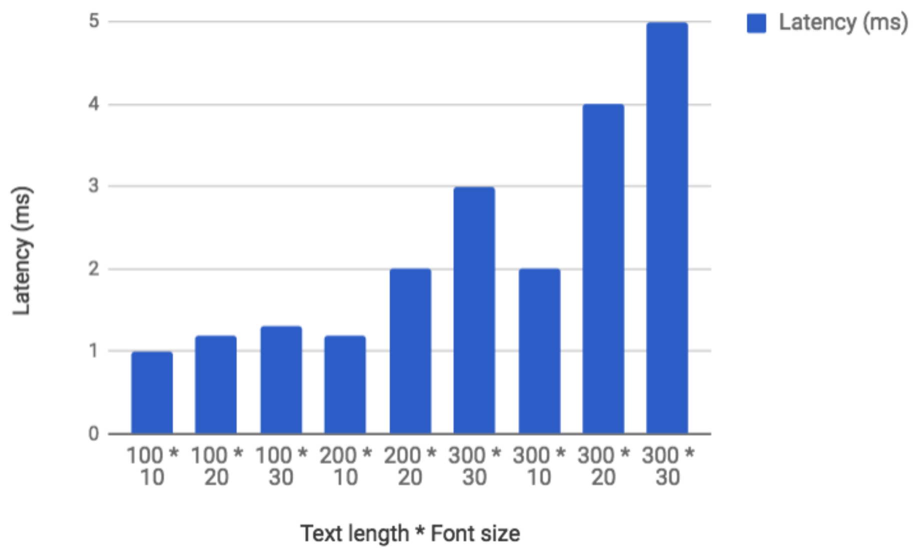


Fig. 6.13.: Text Rendering Overhead

Table 6.5: 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 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 the screen, the actual 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 the framebuffer (2 MB - 8 MB). The Table 6.5 shows both our framebuffer transferring overhead and the projected overhead. We argue that when the 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 the screen hardware, but rare research groups can obtain the access to drivers to a particular model of SOC.

6.7 Discussion

Future work. We implemented our design on Android OS and it can also be applied for iOS, Windows and web-based platforms. First of all, these platforms develop the UI layout in the same way. App developers of these platforms develop the UI layout on a 3-dimensional layout file (layout file for Android, storyboard for iOS, XAML for Windows, HTML for the web-based platform). They also have a similar set of building blocks such as ImageView, TextView, EditText, Dialog, etc. Our design of UI building blocks can also be applied to other platforms. Secondly, their rendering processes are similar and produce a framebuffer as the final result. Third, our design on TrustZone can be a potential reference design for other smartphone TEE solutions (e.g., Apple Secure Enclave).

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 the 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.

6.8 Summary

In this chapter, we proposed a TrustZone UI design model based on *delegation UI model*. Based on the delegation model, we developed the TrustZone-enabled UI building blocks including 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.

7. CONCLUSION AND FUTURE WORK

In summary, this dissertation provides TrustZone solutions to secure data on both mobile-server communication path and mobile-user communication path. The main objective of this work is to propose a new design that is easy-to-use and maintains a small size of TCB. First of all, this dissertation has proposed a *delegation TEE model* to secure data on both communication paths when mobile OSes are compromised. I systematically studied design properties of the delegation model and summarized design principles when applying the model. Second, based on the delegation TEE model, this dissertation proposed a secure server communication solution called *TruZ-HTTP* and *Split SSL* to leverage TrustZone to protect users' data sent to an authorized server without leaking data to an untrusted normal world. Third, based on the delegation TEE model, this dissertation proposed a secure UI solution called *TruZ-View* to leverage TrustZone to protect users' interaction with the mobile UI. All solutions are implemented on a HiKey board. The evaluation result demonstrates that the delegation TEE model can be easy-to-use and have small TCB, and at the same time can also protect the data on mobile communication paths. Future works can be focused on four aspects.

Ease of adoption for device vendors. To let the TEE solution described in this thesis benefits large scale of mobile applications, device vendors such as Samsung, Huawei will have to use our solution and thus, the ease of adoption for device vendors is also important. The approach taken in this thesis requires heavy modification of the Android framework. However, device vendors do not like to frequently update the Android framework, because updating OS requires vendors to retest most of the applications running on their devices so updating OS is a high-cost solution.

An ideal TEE solution for device vendors will require no modification in the Android framework and application developers can use generic TEE services without including their application logic in the secure world. The application-TEE integration should be conducted at the application layer. For instance, application developers can include a TrustZone-enabled HTTPS library in their applications to leverage secure server communication. To draw secure UI, developers can embed a TrustZone-enabled UI widget into application's UI by including a customized widget library in their applications.

This dissertation benefits both application developers and TEE OSes but has not taken device vendors into consideration. The ease of adoption for device vendors will add another dimension to the existing problem. Future research is to find the best middle ground that fits all requirement among three entities: application developers, TEE OSes, and device vendors.

Hardening broader communication paths. This dissertation evaluates one path from both mobile-public communication path and mobile-user communication path. With the evolution of mobile technology, the number of communication paths has been keeping increasing over the years. For instance, UI is not the only communication path for mobile to interact with users. Nowadays, users can send an audio instruction to Siri or Google Assistant. Mobile devices also provide multiple paths in order to physically connect to the public world. Users can use a mobile built-in camera to scan a barcode to gather information, and they can also use NFC to pay merchants. Users' sensitive data are transferred through these communication paths. This dissertation only shares an insight into how to protect the data on communication paths. A similar research problem is how to protect sensitive data on these communication paths by applying the delegation TEE model and what are unique challenges to protect these communication paths.

Split SSL as a building block to secure broader use cases. This dissertation secures the most popular network protocol HTTPS. However, there are various other application-level protocols that leverage the TLS layer as their secure communication protocol, such as the SMTP, POP3, VOIP, etc. Moreover, other network infrastructures, which are not currently using TLS, are also considering incorporating TLS layer into their infrastructures. For instance, the bitcoin payment is developing BIP-70 protocol [22] to secure merchant payment transactions by leveraging PKI infrastructure. To let broader application-level protocols benefit from the TEE, the *Split SSL* can be designed a basic building block to secure other network protocols.

Other approaches to secure data on communication paths. This dissertation focuses on the framework-level approach to secure data on communication paths. Other approaches are also worth exploiting, such as a hardware approach. The research problem will be how to build protection at the hardware level. For example, we can offload the SSL computation into a delegated chip controlled by the TEE or we can secure the screen control at the hardware-level to achieve a better performance than software solutions.

LIST OF REFERENCES

LIST OF REFERENCES

[1] ARM TrustZone Security Technology Whitepaper. https://web.archive.org/web/20170423173146/http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. [Accessed: 23 Apr 2017].

[2] Android Fingerprint HAL. https://source.android.com/security/authentication/fingerprint-hal, 2017.

[3] Android Developers: Android Keystore System. https://developer.android.com/training/articles/keystore.html, 2017.

[4] BitCoin Ledger. https://www.ledgerwallet.com/beta/trustlet, 2017.

[5] Drupal: Open Source CMS. https://www.drupal.org/, 2017.

[6] Drupal Editor - app by dissem on Github. https://github.com/Dissem/Drupal-Editor, 2017.

[7] Elgg: a powerful open source social networking engine. https://elgg.org/, 2017.

[8] F-Droid repository. https://f-droid.org/en/packages/, 2017.

[9] FREAK CVE-2015-0204 Detail. https://nvd.nist.gov/vuln/detail/CVE-2015-0204, 2017.

[10] Android Developers: Selecting Devices. https://source.android.com/source/devices.html, 2017.

[11] Common used Application Layer Protocols. http://www.informit.com/articles/article.aspx?p=169578, 2017.

[12] Intercede's MyTAM. https://www.intercede.com, 2017.

[13] Open Trust Protocol. https://elinux.org/images/2/20/A_More_Open_Trust_Protocol.pdf, 2017.

[14] This POODLE Bites: Exploiting the SSL 3.0 Fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf, 2017.

[15] Poodle Bites TLS. https://blog.qualys.com/ssllabs/2014/12/08/poodle-bites-tls, 2017.

[16] Samsung Pay. http://www.samsung.com/us/samsung-pay/, 2017.

[17] TrustKernel T6 Secure OS. https://www.trustkernel.com/en/products/tee/t6.html, 2017.

[18] SoC and CPU System-Wide Approach to Security. https://www.arm.com/products/security-on-arm/trustzone, 2017.

[19] Learn how to become a Trusted Application Developer with Trustonic. https://developer.trustonic.com/, 2017.

[20] Google Android: Vulnerability Statistics. http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224, 2018.

[21] Beast CVE-2011-3389 Detail. https://nvd.nist.gov/vuln/detail/CVE-2011-3389, 2018.

[22] BIP 70. https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki, 2018.

[23] Face ID. https://support.apple.com/en-us/HT208108, 2018.

[24] Google Assistant. https://assistant.google.com/platforms/phones/, 2018.

[25] MCUFont. https://github.com/mcufont/mcufont, 2018.

[26] Our hardware setup for payment demo. https://youtu.be/f1LJ63_6nug, 2018.

[27] U.S. Smartphone Use in 2015. http://www.pewinternet.org/2015/04/01/us-smartphone-use-in-2015/, 2018.

[28] Mobile Fact Sheet. http://www.pewinternet.org/fact-sheet/mobile/, 2018.

[29] Android WindowManager FLAG SECURE. https://developer.android.com/reference/android/view/WindowManager.LayoutParams, 2018.

[30] China Digital Payments. https://www.statista.com/outlook/296/117/digital-payments/china, 2018.

[31] Tk graphical user interface toolkit. https://www.tcl.tk/, 2018.

[32] TLS 1.2 RFC 5246. https://tools.ietf.org/html/rfc5246, 2018.

[33] XEN security vulnerability. https://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html, 2018.

[34] Xlib X Language X Interface. https://www.x.org/archive/X11R7.5/doc/libX11/libX11.html, 2018.

[35] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*, 2015.

[36] A. Amiri Sani. 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, pages 197–210, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4928-4. doi: 10.1145/3081333.3081346. URL `http://doi.acm.org/10.1145/3081333.3081346`.

[37] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI' 16, Savannah, GA, USA, Nov 2–4 2016.

[38] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS' 14, Scottsdale, Arizona, USA, Nov 3-7 2014.

[39] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI' 14, Broomfield, CO, USA, Oct 6–8 2014.

[40] K. Bhargavan, A. D. Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Beguelin, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *IEEE Symposium on Security and Privacy*, 2017.

[41] 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. 2015.

[42] S. Bugiel, A. Dmitrienko, K. Kostiainen, A. Sadeghi, and M. Winandy. Truwalletm: Secure web authentication on mobile platforms. In *International Conference on Trusted Systems*, Beijing, China, November 2011.

[43] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[44] B. Christina and R. Jean-Marc. Security and usability: The case of the user authentication methods. In *Proceedings of the 18th Conference on L'Interaction Homme-Machine*, 2006.

[45] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, , and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[46] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. Spandex: Secure password tracking for android. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[47] G. J. D. and L. Clayton. Designing for usability: Key principles and what designers think. *Commun. ACM*, 1985.

[48] A. Dmitrienko, Z. Hadzic, H. Löhr, A.-R. Sadeghi, and M. Winandy. Securing the access to electronic health records on mobile phones. In *Biomedical Engineering Systems and Technologies*, 2013.

[49] F. Earlence, Q. A. Chen, P. Justin, E. Georg, H. J. Alex, M. Z. Morley, and P. Atul. *Android UI Deception Revisited: Attacks and Defenses*.

[50] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[51] E. Fernandes, Q. Chen, G. Essl, J. Halderman, Z. Mao, and A. Prakash. TIVOs: Trusted Visual I/O Paths for Android. Technical report, University of Michigan, 2014.

[52] Y. Fratantonio, C. Qian, S. Chung, and W. Lee. 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)*, 2017.

[53] C. Gamze, V. Melanie, B. Clemens, and B. Roland. *NoPhish: An Anti-Phishing Education App*. 2014.

[54] R. Guanciale, M. Dam, H. Nemati, and C. Baumann. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 22-26 2016.

[55] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing ARM trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[56] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, USA, Nov 2–4 2016.

[57] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS' 16, CA, USA, Feb 21 - 24 2016.

[58] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang. Secret: Secure channel between rich execution environment and tee. In *NDSS*, 2015.

[59] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang. Privatezone: Providing a private execution environment using arm trustzone. *IEEE Transactions on Dependable and Secure Computing*, 2016.

[60] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. CCS '14, New York, NY, USA, 2014. ACM.

[61] R. Kainda, I. Flechais, and A. W. Roscoe. Security and usability: Analysis and evaluation. *2010 International Conference on Availability, Reliability and Security*, 2010.

[62] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han. 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, 2017.

[63] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. 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.

[64] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, Beijing, China, June 25-26 2014.

[65] W. Li, H. Li, H. Chen, and Y. Xia. Adattester: Secure online mobile advertisement attestation using trustzone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, NY, USA, May 2015.

[66] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena. Droidvault: A trusted data vault for android devices. *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*, pages 29–38, 2014.

[67] D. Liu and L. Cox. VeriUI: Attested Login for Mobile Devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, CA, USA, February 26-27 2014.

[68] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. Cox. ScreenPass: Secure Password Entry on Touchscreen Devices. In *Proceeding of the 11th annual international conference on Mobile systems, Applications, & Services*, Taipei, Taiwan, June 25-28 2013.

[69] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA, June 2012.

[70] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Proceedings of the 5th International Symposium on Foundations & Practice of Security*, 2012.

[71] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi1, Y. R. Choe, C. Kruegel, and G. Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS' 17, San Diego, CA, USA, Feb 26 - Mar 1 2017.

[72] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun. Secure enrollment and practical migration for mobile trusted execution environments. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices*, SPSM '13, 2013.

[73] S. Nuno, R. Himanshu, S. Stefan, and W. Alec. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[74] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Security Symposium*, Austin, Texas, USA, Aug 10-12 2016.

[75] OP-TEE. OPTEE OS, 2015. URL `https://github.com/OP-TEE/optee_os`.

[76] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[77] C. Ren, P. Liu, and S. Zhu. Windowguard: Systematic protection of gui security in android. In *NDSS*, 2017.

[78] B. Robert, V. Julian, and N. Jan. The threat of virtualization: Hypervisor-based rootkits on the arm architecture. In *Information and Communications Security*, 2016.

[79] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, 2013.

[80] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury. Automated partitioning of android applications for tee. In *Proceedings of the 38th International Conference on Software Engineering*, NY, USA, May 2016.

[81] C. Saksham, G. Nishad, H. Suhas, H. J. I., and A. Yuvraj. Does This App Really Need My Location?: Context-Aware Privacy Management for Smartphones. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2017.

[82] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. Guitar: Piecing together android app guis from memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 120–132. ACM, 2015.

[83] Samsung. KNOX White Paper, 2013.

[84] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, CA, USA, May 17-21 2015.

[85] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, Feb 26 - Mar 1 2017.

[86] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS' 17, San Diego, CA, USA, Feb 26 - Mar 1 2017.

[87] L. Sobel. Secure Input Overlays: Increasing Security for Sensitive Data on Android. Master's thesis, Massachusetts Institute of Technology, MA, USA, 2015.

[88] H. Sun, K. Sun, Y. Wang, and J. Jing. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Denver, Colorado, USA, Oct 12-16 2015.

[89] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *Proceedings of the International Conference on Dependable Systems & Networks*, Brazil, June 22-25 2015.

[90] C. Tian, Y. Wang, P. Liu, Q. Zhou, C. Zhang, and Z. Xu. 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.

[91] T. Tong and D. Evans. Guardroid: A trusted path for password entry. In *Proceedings of Mobile Security Technologies*, San Francisco, CA, USA, May 23 2013.

[92] Trustonic. Trustonic Secured Platforms, 2012.

[93] Trustonic. Trustonic TEE Trusted User Interface, 2012.

[94] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Proceedings of the 21st European Symposium on Research in Computer Security*, Heraklion, Greece, Sept 28-30 2016.

[95] Y. Xia, Y. Liu, C. Tan, M. Ma, H. Guan, B. Zang, and H. Chen. Tinman: Eliminating confidential mobile data exposure with security oriented offloading. EuroSys, New York, NY, USA, 2015.

[96] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P' 15, San Jose, CA, USA, May 17-21 2015.

[97] S. Yalew, G. Maguire, and M. Correia. Light-SPD : A Platform to Prototype Secure Mobile Applications. In *Proceedings of Workshop on Privacy-Aware Mobile Computing*, Paderborn, Germany, July 05 2016.

[98] B. Yang, K. Yang, Z. Zhang, Y. Qin, and D. Feng. Aep-m: Practical anonymous e-payment for mobile devices using arm trustzone and divisible e-cash. In *Information Security*, 2016.

[99] S. Yihang and H. Urs. Privacyguard: A vpn-based platform to detect information leakage on android devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015.

[100] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. Truz-droid: Integrating trustzone with mobile operating system. In *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, Munich, Germany, 2018. ISBN 978-1-4503-5720-3.

[101] K. Ying, P. Thavai, and W. Du. Truz-view: Developing trustzone user interface for mobile os using delegation integration model. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, pages 1–12, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6099-9. doi: 10.1145/3292006.3300035. URL `http://doi.acm.org/10.1145/3292006.3300035`.

[102] L. Yuanchun, C. Fanglin, L. T. Jia-Jun, G. Yao, H. Gang, F. Matthew, A. Yuvraj, and H. J. I. PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2017.

[103] N. Zhang, K. Yuan, M.Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security and Privacy*, pages 915–930.

[104] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou. CacheKit: Evading Memory Introspection Using Cache Incoherence. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*, EuroS&P' 16, Saarbrucken, Germany, March 21-24 2016.

[105] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, S&P' 16, San Jose, CA, USA, May 22-26 2016.

[106] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[107] X. Zhang, Y. Aafer, K. Ying, and W. Du. Hey, you, get off of my image: Detecting data residue in android images. In *Computer Security – ESORICS 2016*, 2016.

[108] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *NDSS*, 2016.

[109] X. Zheng, L. Yang, J. Ma, G. Shi, and D. Meng. Trustpay: Trusted mobile payment on security enhanced arm trustzone platforms. In *ISCC*, 2016.

[110] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag.

VITA

VITA

Kailiang Ying was born in Shanghai, China. Received a Bachelor of Science degree in computer science from Shanghai Ocean University (Shanghai, China), June of 2012. Received a Master of Science degree in computer science from Syracuse University (Syracuse, New York, USA), December of 2013. This dissertation was defended in November 2018 at Syracuse University.