December 2020

SECURING USER INTERACTION CHANNELS ON MOBILE PLATFORM USING ARM TRUSTZONE

Amit Ahlawat
Syracuse University

ABSTRACT

Smartphones have become an essential part of our lives, and are used daily for important tasks like banking, shopping, and making phone calls. Smartphones provide several interaction channels which can be affected by a compromised mobile OS. This dissertation focuses on the user interaction channels of UI input and audio I/O. The security of the software running on smartphones has become more critical because of widespread smartphone usage. A technology called `TEE` (Trusted Execution Environment) has been introduced to help protect users in the event of OS compromise, with the most commonly deployed `TEE` on mobile devices being ARM TrustZone.

This dissertation utilizes ARM TrustZone to provide secure design for user interaction channels of UI input (called *Truz-UI*) and Audio I/O for VoIP calls (called *Truz-Call*). The primary goal is to ensure that the design is transparent to mobile applications. During research based on `TEE`, one of the important challenges that is encountered is the ability to prototype a secure design. In `TEE` research one often needs to interface hardware peripherals with the `TEE` OS, which can be challenging for non-hardware experts, depending on the available support from the `TEE` OS vendor. This dissertation discusses a simulation based approach (called *Truz-Sim*) that reduces setup time and hardware experience required to build a hardware environment for `TEE` prototyping.

SECURING USER INTERACTION CHANNELS ON MOBILE PLATFORM USING ARM TRUSTZONE

by

Amit Ahlawat

B.Tech., Maharshi Dayanand University, 2010

M.S., Syracuse University, 2012

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering.

Syracuse University

December 2020

ACKNOWLEDGMENTS

There are several people who I would like to thank for the successful completion of my PhD thesis.

First and foremost, I would like to thank my advisor Dr. Wenliang Du for giving me the opportunity to work in his lab as a Research Assistant during my Masters degree and giving me a chance to work on a PhD degree under his guidance. I thank him for the training he provided related to formulating research problems, judging the merits of a formulated problem and applying constructive & critical thinking while doing research. I also thank him for the feedback he provided in all the project meetings we had over the years.

I would like to thank Dr. Jae Oh, Dr. Richard Tang, Dr. Fanxin Kong, Dr. Bryan Kim, and Dr. Carlos E. Caicedo Bastidas, for agreeing to be on my thesis committee. I am extremely grateful for their time in reading my dissertation and commenting on my views. I would like to thank the department chair Dr. Jae Oh for his continued support during my PhD study.

During my time in Dr. Wenliang Du's lab, I have been very fortunate to have had the opportunity to work with several colleagues, including Dr. Kailiang Ying, Dr. Xiao Zhang, Dr. Yousra Aafer, Dr. Paul Ratazzi, Xing Jin, Ammar Salman, Francis Akowuah, Yifei Wang, Haichao Zhang, Hanyi Li, Yuexin Jiang, Carter Yagemann, Zhenyu Wang, Amey Ashok Patil, Gautam Peri, Priyank Thavai and Bilal Alsharifi. I would like to thank Dr. Andrew Henderson for all the guidance he provided on various projects in my thesis.

I would like to thank my family for their support of my decision to pursue a PhD degree, and their continued support during my PhD study.

Finally I would like to thank the various staff members of the EECS department and other departments responsible for ensuring the success of PhD students at Syracuse University.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure | Page

1. INTRODUCTION

Smartphones have become a common tool in modern society. Based on recent statistics [78] from the US, as of Feb 2019 81% of adults own a smartphone. One of the popular mobile OS Android now has a majority market share [89]. The widespread adoption of smartphones makes the security of mobile OS extremely important. Unfortunately, the recent trend has not been promising. CVE numbers show that the number of disclosed vulnerabilities in Android has remained high [71]. A recent attack on Android could achieve arbitrary code execution in a privileged process by using a crafted image file [63]. Smartphones provide several interaction channels which can be affected by a compromised mobile OS. This chapter discusses the types of smartphone interaction channels and provides an overview of secure solutions for specific interaction channels.

1.1 Risks faced by Smartphone Channels

Smartphones provide several interaction channels (Figure 1.1), including user interaction channels (UI and audio I/O), context based channels (camera and GPS), inter-phone channels (bluetooth and NFC), and back end channels (network interface to communicate with server). A compromised mobile OS can affect various use cases for these channels. For user interaction channels like UI input and audio I/O, a compromised OS can steal user secrets (e.g. password). For context based channels like camera and

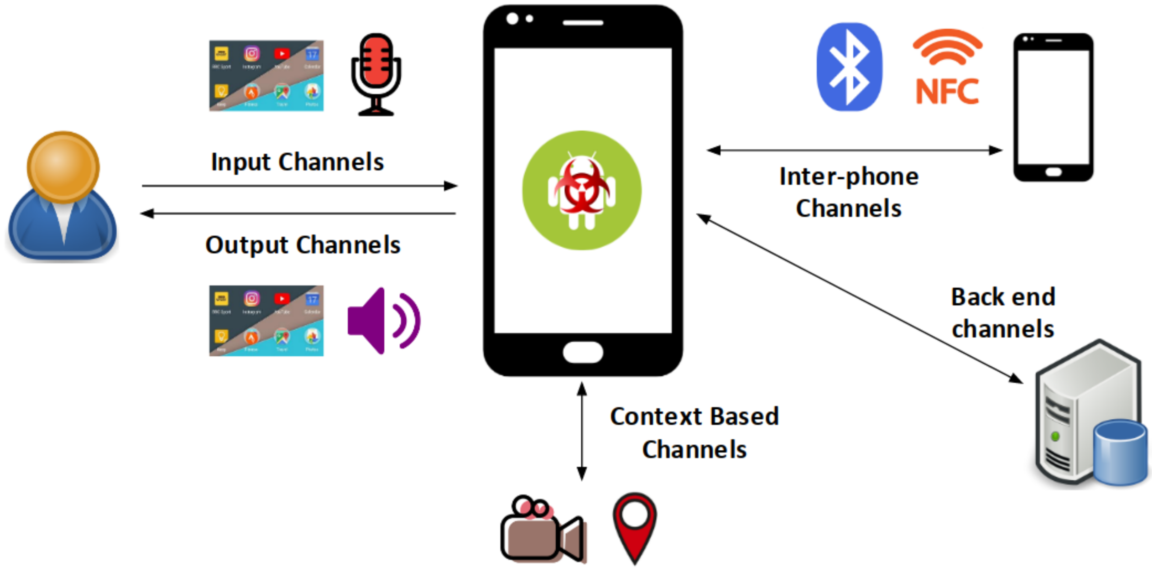

Fig. 1.1.: Smartphone Interaction Channels

location, a compromised OS can falsify environment (e.g. spoof location). For inter-phone channels like NFC and bluetooth, a compromised OS can steal data being exchanged between the devices. For back-end channels with servers, a compromised OS can steal data sent to a server or send forged data to a server.

This dissertation focuses on the user interaction channels of UI input and audio I/O. UI input is used to allow the user to enter a secret or to approve an action in a mobile application. For example, in case of a banking application, a compromised OS can steal user's secret information such as bank passwords, and spoof actions such as transferring money out of the user's bank accounts on behalf of the user. An important use case of audio I/O is user's ability to make phone calls. In recent years, VoIP apps such as Signal [25] and Whatsapp [14] have become popular ways for making a call. Recent survey [64] indicates top social apps used have VoIP calling support. A compromised mobile OS can listen to user's VoIP call. Today different types of users need to have a secure means of calling, including activists, journalists, government employees etc. Given the risk to user interaction channels, there is a need to design solutions utilizing features in mobile architectures that can provide security inspite of a compromised mobile OS.

1.2 ARM Architecture and Trusted Execution Environment

Majority of mobile devices use ARM architecture [26]. It is divided into two worlds [65] as shown in Figure 1.2. The *normal world* contains normal apps and mobile OS (like Android). As mentioned in previous sections, the mobile OS can be potentially compromised. To design secure solutions, one may look at utilizing the hypervisor which has higher privilege than the mobile OS. Existing research [155] shows vulnerabilities in hypervisor on the ARM platform. Given the normal world cannot be trusted to design secure solutions, ARM architecture contains a second world called the *secure world*.

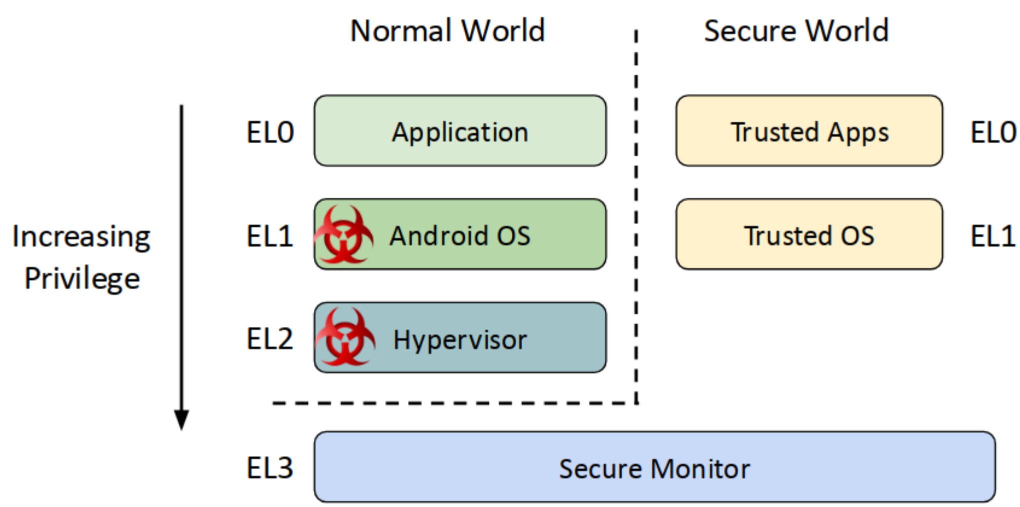

Fig. 1.2.: ARM Architecture Privilege Levels

The *secure world*, also referred to as trusted execution environment (`TEE`), provides an execution environment isolated from the normal world. The most commonly deployed `TEE` on mobile devices is ARM TrustZone [96]. Other architectures also support `TEE`, including AMD Platform Security Processor, Apple Secure Enclave and Intel Software Guard Extensions (SGX). A compromised mobile OS cannot access data in the secure world and cannot access hardware protected by the secure world. The secure world runs an independent trusted OS (will be referred to as `TEE` OS) with its own set of trusted applications (also referred to as `TA`). Popular examples of `TEE` include Samsung TIMA [91] which uses TrustZone to provide various security services (e.g. keystore, trusted user interface), and Trustonic [145] which uses TrustZone to provide security solutions to various vendors (e.g. mobile payment apps like WeChat and AliPay).

1.3 Component Binding Across OS

In a typical computing system, components in userspace, kernel and hardware interact with each other to form a *single OS context*. At userspace level, components can include processes, and at a finer granularity level, the various libraries (modules) used in the processes. At kernel level, components can include modules like various device drivers. At hardware level, components can include various peripherals being used by the system. Within an OS context, this dissertation uses the term *binding* to refer to interaction between two components via OS support. Example of binding can include application interacting with hardware, process interacting with another process via IPC etc. The key to the term is that some type of OS support is involved.

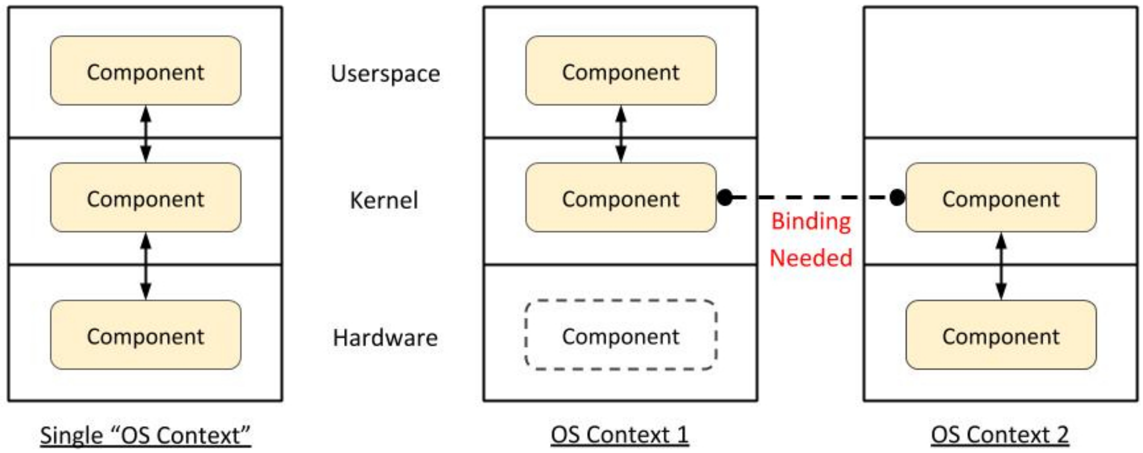

Fig. 1.3.: OS Context and Binding

There can be circumstances where components cannot exist in the same OS context, but rather exist across two different context. In such situations, if these components need to interact, an OS-level binding needs to be created (Figure 1.3). The binding can be created across two similar OSes (Figure 1.4). For example, an app on one Android phone using the hardware on a different Android phone. In case where the components exist in different types of OS, a *cross-OS binding* is needed (Figure 1.4). In this dissertation, two types of cross-OS bindings are introduced in the designs for secure input interaction (*Truz-UI*) and simulation platform for `TEE` prototyping (*Truz-Sim*).

1.4 Thesis Statement and Contributions

The thesis statement of this dissertation is that, **design solutions transparent to applications to protect user interaction channels on mobile platform using ARM TrustZone**. The dissertation focuses on the user interaction channels of UI input and

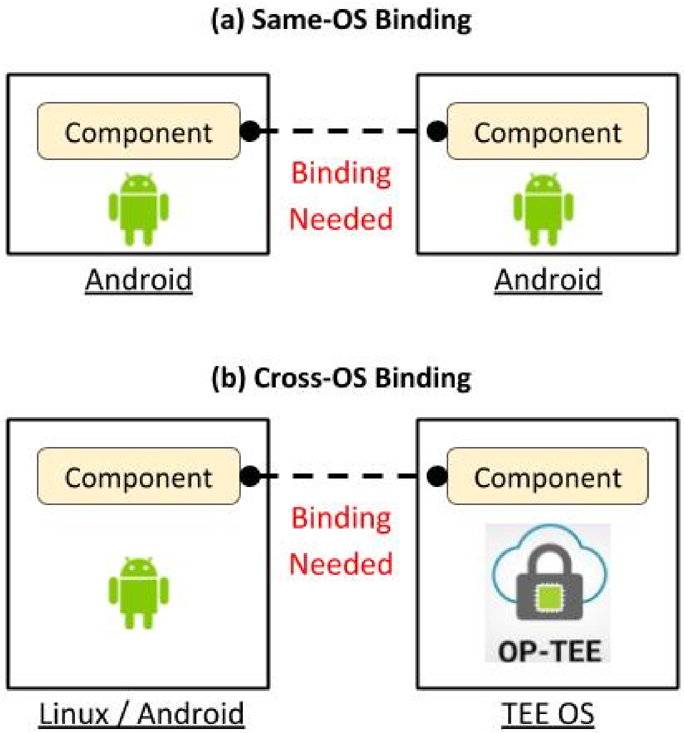

Fig. 1.4.: Types of Binding

audio I/O for VoIP calls. In support of this statement, this dissertation describes the following contributions:

1. **Truz-UI:** Users provide secret data to the smartphone via the interaction channel of UI input (touch input). To protect user's secret data, we need to protect the interaction between the user and the smartphone so that the secret data will be never given to the normal-world OS. Two common types of touch based interactions are typing text and confirming an action. A compromised normal-world OS poses a risk to such interactions. Taking mobile banking as an example, when a user logs in to the bank's server, the user needs to type a password, which can be stolen if the OS is compromised. Second, when the user conducts a money-transfer transaction, the compromised OS can replace the receiver's account number with the one belonging

to the attacker, leading to loss of money. TrustZone can be leveraged to protect such interactions because of the hardware level isolation it offers. It is important to allow apps to use `TEE` via existing normal-world OS APIs and without a need 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 transparent design that allows normal-world apps to leverage TrustZone via existing OS APIs to protect user interaction via UI input. The goal is achieved by incorporating generic TrustZone support at the OS level so that normal-world apps can use TrustZone without the need to put their own code inside the secure world. Reusing existing APIs can be achieved by moving the sensitive UI interaction into the secure world, while still maintaining the UI's functionality related to its corresponding code in the normal-world app. This is achieved by creating a *cross-OS binding* between the UI interaction in the secure world and the code in the normal-world app. Using this approach, the app developer requests a secure version of the UI and provides the code to be bound to this UI. When the UI in the secure world finishes collecting inputs from users, the bound code in the normal-world app is triggered. This design has been evaluated using both open and closed source apps in this dissertation. The design has been tested on the TrustZone-enabled Hikey development board. The performance evaluation shows that the overhead from Truz-UI is not noticeable to users.

2. **Truz-Call:** Users make end-to-end encrypted VoIP calls using various apps on mobile OS like Android. When the user initiates a VoIP call, the app uses OS APIs to fetch audio, processes the audio, and sends out the packet over the network (reverse flow for incoming packets). The app uses a VoIP protocol like `SRTP` to encrypt and calculate `HMAC` for the audio payload (in `RTP` packets), and send the encrypted payload to the callee device. With a compromised OS, the user's privacy is at risk during the call. TrustZone can be leveraged to protect user's voice interaction because of the hardware level isolation it offers. VoIP apps should be enabled to use TrustZone to protect the user's conversation while using the existing OS APIs and existing VoIP protocols, without a need to install app-specific `TA` in the secure world. The design should be transparent to developers and to the existing VoIP infrastructure. This dissertation presents a transparent design to protect user's audio I/O during a VoIP call by integrating `TEE` at essential stages in a VoIP app's audio pipeline. The design allows VoIP apps to leverage TrustZone while using existing OS APIs and VoIP protocol, and provides generic `TA` support so that no app-specific `TA` code is needed. The conversation audio during a VoIP call is protected from the normal-world OS. The design has been evaluated using an open source VoIP app Linphone on the TrustZone-enabled Hikey development board.

3. **Truz-Sim:** `TEE` research often involves interfacing different types of hardware peripherals with the `TEE` OS. This task can be challenging for non-hardware experts, depending on the available support from the `TEE` OS vendor. There is a need for a `TEE` prototyping environment that can allow researchers to interface

different category of hardware with the TEE OS irrespective of the available support from the vendor, and can best retain the quality of data needed for prototyping. To meet this requirement, this dissertation introduces a simulation based testing environment that allows reduced setup time and requires no hardware experience for setup. The idea involves creating a simulation driver in the TEE OS that facilitates a *cross-OS binding* between the trusted application in the TEE and hardware attached to a different OS, for example, on a different board like Raspberry Pi. This allows TAs in the TEE on a TrustZone-enabled development board like Hikey, to transparently access hardware attached to a binded board like Pi. The design has been evaluated for the use cases of a TA needing access to data from camera, GPS and UI hardware.

1.5 Organization of Dissertation

Chapter 2 provides background on ARM TrustZone and related development boards, text input & action confirmation in Android, and VoIP calling. Chapter 3 discusses `Truz-UI` to provide secure input interaction. Chapter 4 discusses `Truz-Call` to provide secure voice interaction for VoIP calling. Chapter 5 discusses hardware simulation to assist research related to TrustZone. Chapter 6 presents conclusion and future work.

2. BACKGROUND

2.1 ARM TrustZone

The TrustZone technology is a system-wide approach to security that allows building secure endpoints with a root of trust. Using TrustZone, a System-on-Chip's (SoC) hardware and software resources are partitioned to provide security, s.t. the resources exist in one of two hardware-separated worlds, the *secure world* for a security subsystem, and the *normal world* for everything else (as shown in Figure 1.2). The normal-world software is not allowed to access the secure-world resources. The concepts of normal and secure world are applied to various parts of the SoC, including memory, software, bus transactions, interrupts and peripherals.

The 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. The mechanism to context switch between them is known as monitor mode. The entry to the monitor can be triggered by software executing the Secure Monitor Call (`SMC`) instruction. The secure-world comprises of various software components, including trusted boot, the secure-world switch monitor, a small trusted OS and trusted apps (or `TA`). There are several trusted OSes currently in development, including OP-TEE [85], T6 [1], Trustonic [145], etc.

Secure Boot. As shown in Figure 2.1 [96], 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 completed, 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 secure world boot process. It aims to assert the integrity of the secure world software, preventing any unauthorized or maliciously modified software from running.

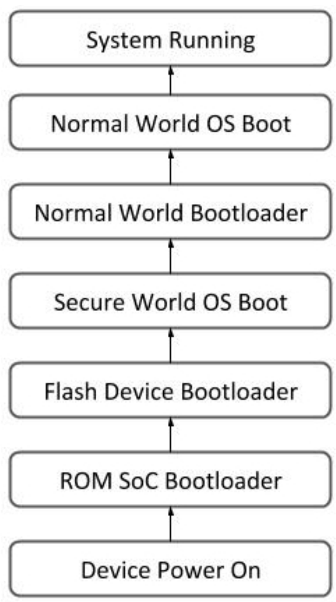

Fig. 2.1.: Secure Boot

OPTEE OS. This is an open-source TEE OS maintained by Linaro, based on the GlobalPlatform TEE system architecture specification [54]. It is designed to be compatible with any isolation technology suitable for TEEs, including TrustZone . In TrustZone , the OP-TEE 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 OP-TEE kernel forwards the normal-world request to a TA and returns the result back to the normal world.

2.2 TrustZone Development Boards

To conduct TEE research one needs select a device for testing. Commercial Android phones with the TrustZone feature have TrustZone locked down by the manufacturers. Researchers have to instead rely on development boards that can allow modifications to both normal world and secure world. Since the research done in this dissertation is focused on mobile OS (primarily Android), the board selected is the one recommended by Google to run Android upto the year 2020 [53, 106]. The board recommended by Google is Hikey [125] (shown in Figure 2.2). This dissertation relies on the Hikey 620 board for testing.

Fig. 2.2.: 96Boards Hikey 620 Development Board

In order to modify the secure world, the development board needs to be supported by a TEE OS vendor. The vendor would provide a patch to the Android source code released by Google, so that when the final version of the code is flashed on the board, both the normal world and secure world OS can be updated. Fortunately OP-TEE OS provides support [134] for the Hikey board. This allows a research environment where modifications can be made at the user and OS levels in both Android and OP-TEE.

2.3 Android Text Input

Android allows users to provide text input to applications. Android supports this via the input method framework [103]. It has three overall pieces as shown in Figure 2.3. Applications include UI elements to accept text input. User interaction with these UI elements requests Android framework to display a keyboard UI for input. The system displays a keyboard UI based on the currently configured keyboard app (also referred to as input method editor or `IME`). User can interact with the keyboard UI to provide text input to application UI element.

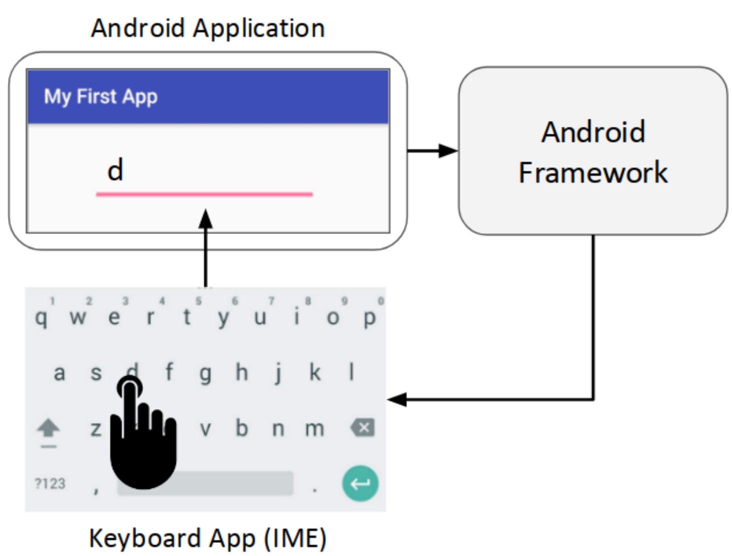

Fig. 2.3.: Android Input Method Framework Overview

2.3.1 Text Input UI Element

Android allows app developers to create user interface to allow touch interaction with users. Developers create app components called `Activity` which create windows in which developers can place their UI. An application's UI is represented in XML format. App developers use a UI element called `EditText` [101] to accept text input from users. Listing 2.1 shows an example of an application UI containing two `EditText` elements. The corresponding app UI is shown in Figure 2.4. In the example, several attributes are specified for `EditText`, including height, width and `inputType` [105]. The `inputType` attribute informs the system whether expected input is just text or special input like password. Other types include phone, time, date etc.

Listing 2.1: EditText 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"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.example.edittext.MainActivity"
    android:orientation="vertical">
    <EditText
        android:id="@+id/edittext1"
```

```
        android:layout_width="200dip"
        android:layout_height="50dip"
        android:inputType="text"
        android:layout_marginLeft="10dip"/>
    <EditText
        android:id="@+id/edittext2"
        android:layout_width="200dip"
        android:layout_height="50dip"
        android:inputType="textPassword"
        android:layout_marginLeft="10dip"/>
</LinearLayout>
```

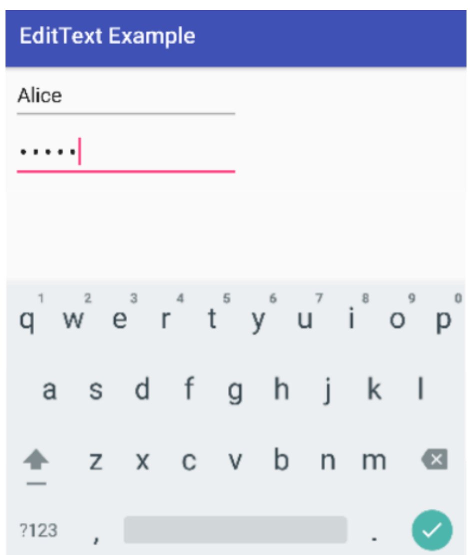

Fig. 2.4.: EditText UI Example

2.3.2 Text Input via Binding

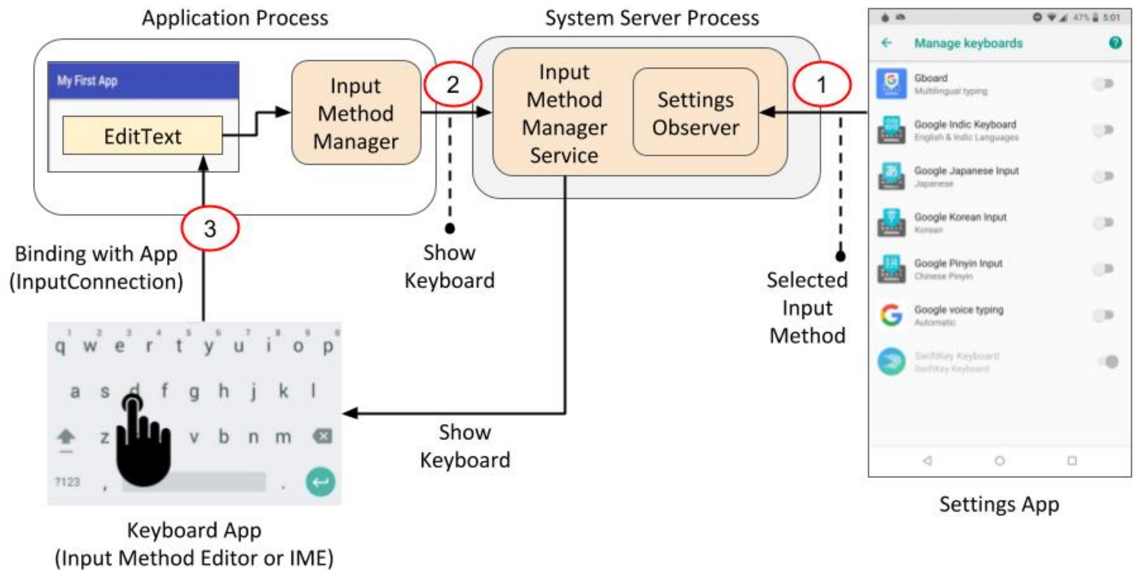

Fig. 2.5.: Keyboard Input via Android Input Method Framework

This section explains how the Android input method framework allows a keyboard app to provide input to an Android application. Figure 2.5 shows the overall flow. The figure is divided into three overall steps. Android allows users to install different keyboard apps and select which one to use via Android's settings app [61]. When user selects a particular keyboard app, a system service (running in a privileged process) called `InputMethodManagerService` (will be referred to as `IMMS`) is notified (step ①). When the user interacts with an `EditText` in an Android app, an in-app Android framework component called `InputMethodManager` sends an IPC request for the keyboard UI to `IMMS` (step ②). The `IMMS` requests currently selected keyboard app to show its keyboard UI.

Keyboard apps (also referred to as Input Method Editor or `IME` [102]) are developed by deriving the Android class `InputMethodService` [128]. Every `IME` app has a life cycle. The IMMS is responsible for managing the life cycle for the currently selected IME. One of the important steps in this life cycle is providing the current IME a binding (of type InputConnection [104]) to the current application. The binding allows the IME to send text input to the EditText in the app (step ③). Once the text input is completed, the EditText in the app can get the text entered by the user using the API getText() (as shown in Listing 2.2).

Listing 2.2: EditText getText() Example

```
public class MainActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      EditText editText
                = (EditText) findViewById(R.id.
                   edittext1);
      String str = editText.getText().toString();
   }
}
```

2.4 Android Action Confirmation

This section describes how Android app developers can ask user to confirm an action and how user interaction with the confirmation UI results in the corresponding code being triggered in the app. Two common Android components that can be used to ask for user confirmation are `AlertDialog` and `Activity`.

2.4.1 Using AlertDialog for Confirmation

Dialogs [100] commonly consist of a user message and a set of buttons. Listing 2.3 shows an example of how Android developers create a dialog using the `AlertDialog` class. Given a confirmation message (set using `setMessage()`), an app requests a dialog using the `show()` API while providing button code for UI buttons. The confirmation UI runs in the caller app process. The app gets a response upon user interaction with the dialog UI via Android's input event handling framework [30]. Figure 2.7 shows an example of a dialog UI.

Listing 2.3: AlertDialog Example

```
// "this" refers to the containing Activity
AlertDialog.Builder builder
  = new AlertDialog.Builder(this);
builder.setMessage("Confirm transfer of $50 to Bob ?")
 .setPositiveButton("OK",
   new DialogInterface.OnClickListener() {
   @Override
```

```
    public void onClick(DialogInterface dialog, int
       button)
    { /* Handle User Approval */  }
  })
 .setNegativeButton("Cancel",
    new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int
       button)
    { /* Handle User Cancellation */ }
  });
AlertDialog alertDialog = builder.create();
alertDialog.show();
```

2.4.2 Using Activity for Confirmation

As mentioned in Section 2.3.1, an `Activity` allows an Android developer to create a UI for user interaction. To get user confirmation, an app `Activity` can invoke a second `Activity` containing the confirmation UI. Based on the confirmation result, corresponding code can be triggered in the calling `Activity`.

The calling `Activity` specifies the confirmation message as part of an `Intent` and uses the `startActivityForResult()` API for confirmation UI invocation. The UI may run in same or different process and is invoked via `ActivityManagerService`.

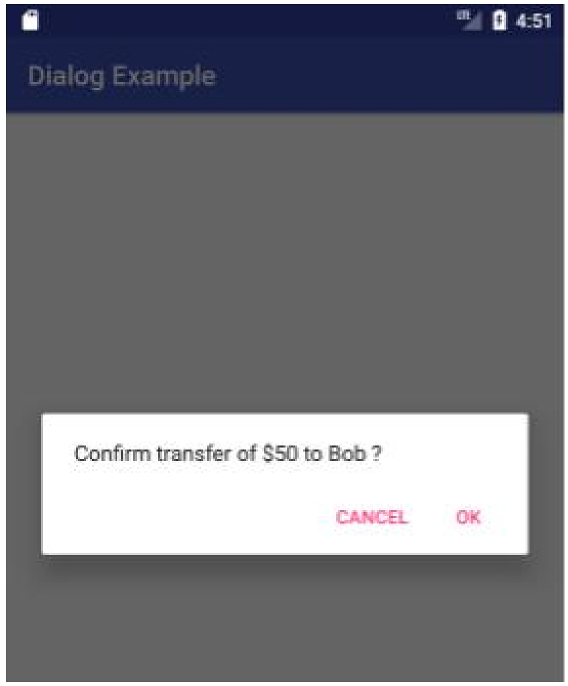

Fig. 2.6.: AlertDialog Example

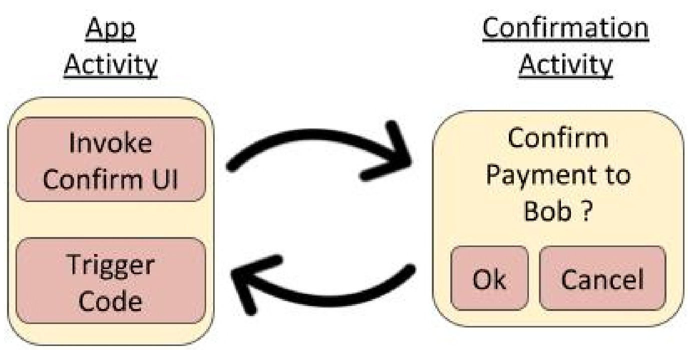

Fig. 2.7.: Confirmation Activity Flow

Upon user interaction with the confirmation activity UI, the activity constructs a result using `finish()` and sends to the caller app via the Intent IPC framework. The caller app gets the response from the confirmation UI via the callback `onActivityResult()`.

Listing 2.4: Confirmation Activity Request and Response

```
// Calling Activity
Intent intent = new Intent("com.example.ACTION");
intent.putExtra("msg",
        "Confirm transfer of $50 to Bob ?");
startActivityForResult(intent, confirm_request);


// Confirmation Activity
OkButton.setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View view) {
    setResult(Activity.RESULT_OK);
    finish();
    // Cancel button can use Activity.RESULT_CANCELED
    // Data can also be returned (not used in example)
  }
});


// Back to Calling Activity
protected void onActivityResult(int request_code,
        int result_code, Intent data) {
   if(request_code == confirm_request &&
```

```
        result_code == RESULT_OK) {
      // Handle user confirmation
   }
}
```

2.4.3 Trigger Confirmation Code via Binding

This section explains how user interaction with the confirmation UI shown via `AlertDialog` and `Activity` triggers corresponding app code via binding support provided by the OS.

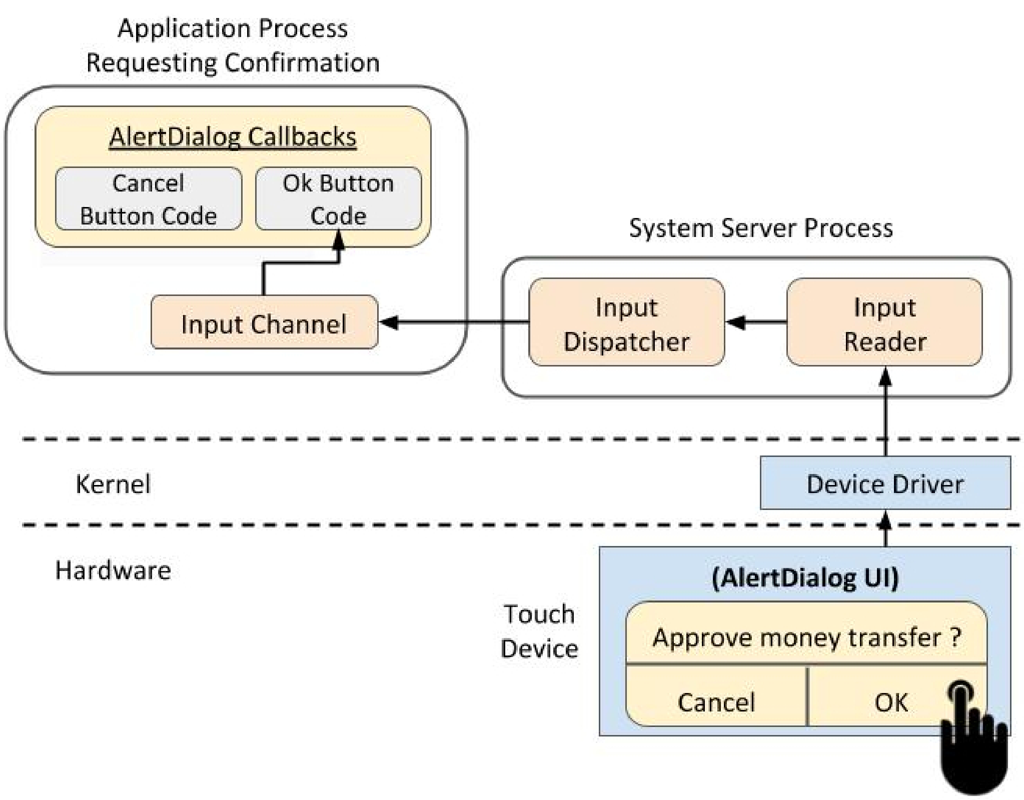

Fig. 2.8.: Trigger Dialog Button Code

When a user interacts with the dialog UI by pressing either the OK or the Cancel button, the event associated with user's touch interaction passes through several stages before it reaches the `AlertDialog` in the app. The binding between the UI interaction and corresponding app code is provided by the OS via the input event handling framework [30] (shown in Figure 2.8). The UI interaction event is captured by the hardware and passed onto the Linux device driver in the kernel. Android's system server process (a privileged process) receives the event in a component called `InputReader`. It forwards it to `InputDispatcher` which sends the event to the application via the `InputChannel` layer in the application process. Android app's UI is organized as a hierarchy of UI elements (also referred to as `Views`). The event is passed down the view hierarchy in the app. Eventually the event triggers the code associated with the clicked button via the `onClick()` callback function.

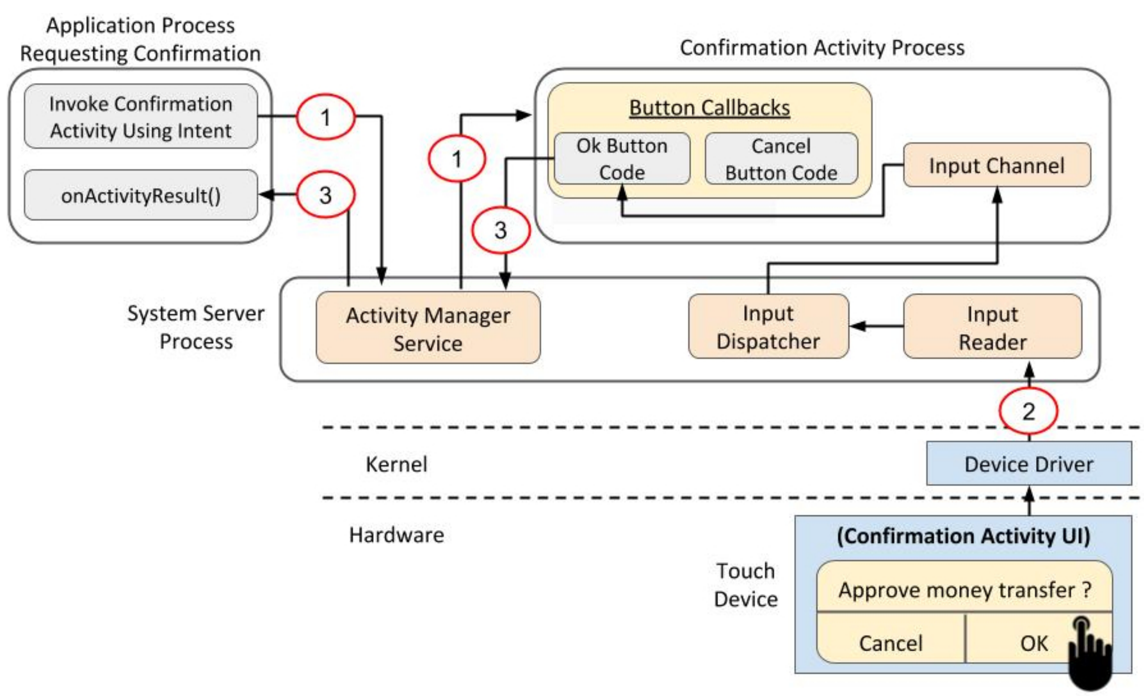

Fig. 2.9.: Trigger Requesting Activity Code

In the case of confirmation via `Activity`, the flow is similar upto the triggering of code in the confirmation `Activity`. Based on whether the user confirmed or denied the action, the confirmation `Activity` will use the `Intent` IPC framework to return the result to the app requesting confirmation via the system server process (as shown in Figure 2.9). The binding support provided by the OS thus has two stages, the input event framework to trigger the code in the confirmation `Activity` and the `Intent` IPC framework to trigger the callback `onActivityResult()` in the app requesting the confirmation.

2.5 Voice over IP (VoIP) Call

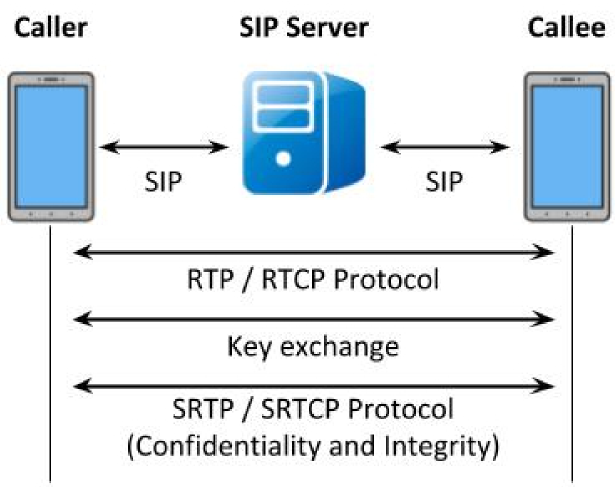

Fig. 2.10.: VoIP Call Flow

Voice over Internet Protocol (VoIP) [178] allows delivery of voice communications over Internet Protocol (IP) networks like the Internet. Common protocols used by VoIP software for secure calling using end-to-end encryption can be found at [174]. From the data available for protocols used by apps, a common protocol for VoIP with open source implementation is SRTP [10] using SIP [6] for call initiation. Figure 2.10 gives a high level view of the flow involved in connecting a VoIP call. If a caller wants to call a callee, they will first use the SIP application-layer protocol [6, 22] to exchange information. The information is exchanged using SDP messages [2] enclosed within SIP messages. The SIP protocol does not carry any audio data; it is used to initiate a session between the two end points. Once the connection is established, protocols like RTP [8] are used to deliver audio between the two end points. RTP is used alongside the RTP Control Protocol (RTCP). RTP is used to carry media streams, while RTCP is used to monitor transmission statistics and quality of service. SRTP is a profile of RTP that provides confidentiality, message authentication, and replay protection to RTP traffic. A sister protocol SRTCP provides the same features for RTCP. SRTP resides between the RTP application and the transport layer. It intercepts RTP packets and then forwards an SRTP packet containing encrypted payload and HMAC on the sending side, and intercepts SRTP packets and verifies HMAC and decrypts payload to provide an RTP packet up the stack on the receiving side. SRTP and SRTCP need keys for encryption and HMAC. These keys are derived from master keys which are set up using a key exchange mechanism. Protocols used by VoIP to setup master keys include DTLS [15, 19] and ZRTP [18].

3. TRUZ-UI: SECURE INPUT INTERACTION

3.1 Problem Overview

Users provide sensitive inputs when using Android applications. Two common types of input are text input and action confirmation (shown in Figure 3.1). In order for an app to protect text input, users should be able to type a secret (e.g, password) without allowing the compromised OS to see the secret. 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. TrustZone can allow users to type their secret in the right app without leaking to the untrusted normal-world OS (this dissertation does not cover the sending of secret to authorized server; covered in existing thesis [180]). In order for an app to enforce user's intention, 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. To protect this interaction, before an important transaction is committed, TrustZone can ask users for confirmation so that the transaction can be attested (signed using TrustZone) and its integrity can be preserved. The attested confirmation should allow the receiving server to verify that the action was confirmed by the user.

The problem of protecting user's sensitive data and user's intention has been solved by TrustZone, but the current solutions like [24, 163] do not satisfy the following constraints: (a) normal-world apps can reuse existing OS interfaces to leverage the TrustZone support,

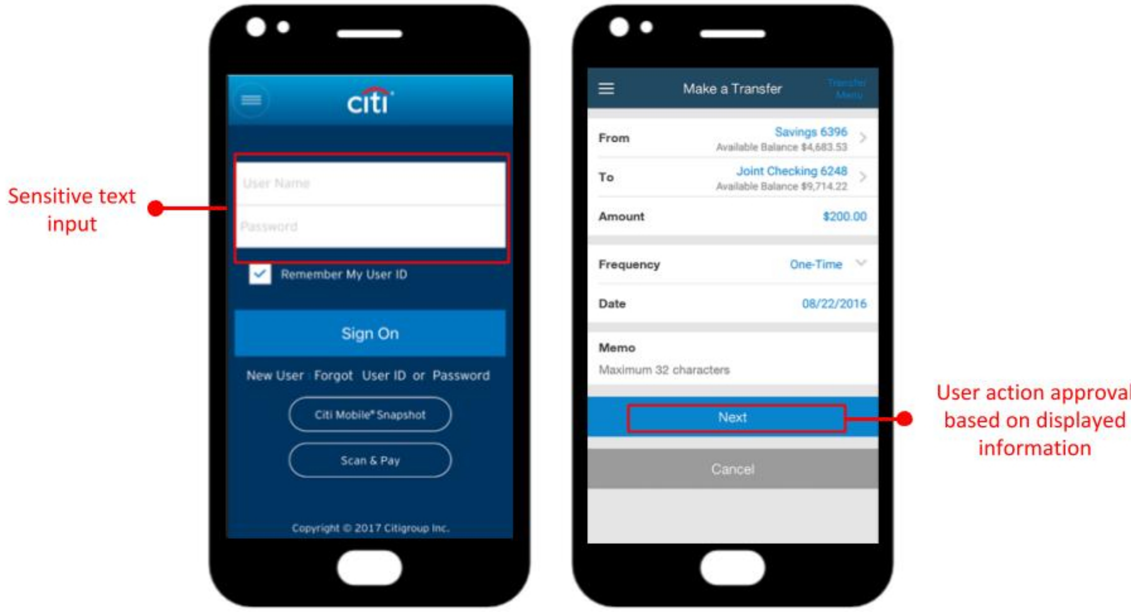

Fig. 3.1.: User Input Interactions

(b) no app-specific logic in the secure world, and (c) minimize Trusted Computing Base (TCB) while providing generic `TEE` support. In order to allow an app to protect user input interaction 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.

Threat Model. The adversary model is shown in Figure 3.2. The user of the device is trusted. The normal world that includes the apps and 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 the Trusted Applications (`TA`) and `TEE` OS is trusted. It will protect the user's confidentiality and integrity when the normal world is compromised. The server is assumed to be trusted after it is authorized by the user.

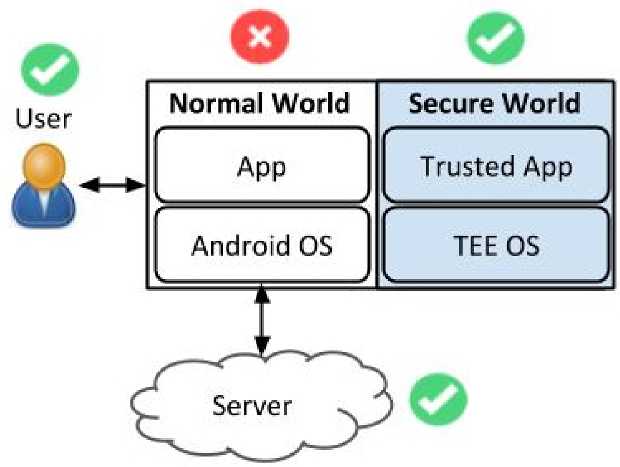

Fig. 3.2.: Input Interaction Threat Model

3.2 Broken Binding between Code and UI

Given the risks to user input interactions from a compromised OS, this dissertation states the following problem: *How to allow the normal-world apps to reuse existing APIs to protect UI interaction for text input and action confirmation using ARM TrustZone?*

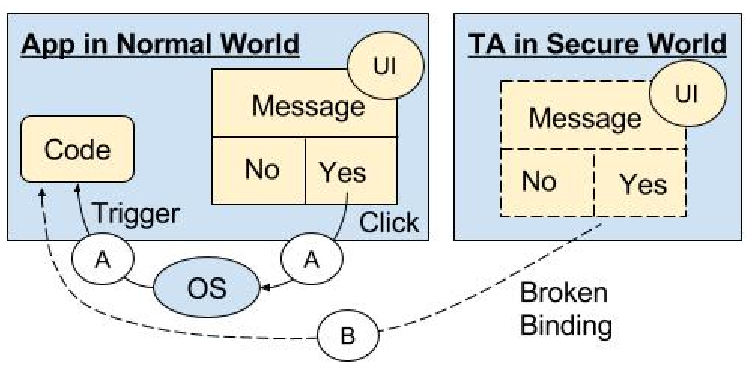

Fig. 3.3.: Binding Between Code and UI

Reusing existing APIs can be achieved by moving the sensitive UI interaction into the secure world, while still maintaining the UI's functionality related to its corresponding code in the normal-world app. Taking the example of Android dialog box for action confirmation, using a dialog box in an app involves two parts: a UI component and a code component. As shown in Figure 3.3 (path A), the OS provides a binding between the UI and code to be triggered. Moving the sensitive UI interaction into the secure world breaks the existing binding support provided by the OS, as shown in path B. To maintain the same API interface, we should allow the developer to leverage `TEE` support while using the existing dialog box component and should preserve the UI functionality of the dialog box. The UI's binding to its corresponding code in the app needs to be maintained. When the dialog button is clicked in the secure world, the code for the dialog button in the normal-world app should still be triggered.

3.3 Main Idea: Cross-OS Binding

The approach in this dissertation to achieve the required protection is to move the sensitive UI interaction into the secure world and to maintain the binding between the UI interaction and normal-world app code across OSes. This *cross-OS binding* allows the apps to leverage the UI in `TEE` by using existing APIs. In normal cases, an app developer requests a UI and provides the associated code to be triggered from the UI. Using the proposed approach, the developer will instead request a secure version of the UI and provide the code to be bound to this UI. To the developer, the way to request a secure UI is the same as other UIs, but to the system, when the secure UI needs to be displayed, the corresponding UI is displayed in the secure world. When the UI in the secure world finishes collecting inputs from users, the bound code in the normal-world app is triggered. This dissertation refers to this binding support as *TruZ-UI*. In order to have no app-specific code in the secure world, the proposed design provides generic TAs for keyboard and confirmation UIs.

In order to protect the user's interaction in the secure world, the hardware input (touch digitizer) and display (screen content) need to be protected. To protect the user's interaction when the device switches to the secure world, these peripherals should only be accessible from the secure world. Users also need an indicator to identify whether they are interacting with the normal world or secure world. The indicator should be exclusively controlled by the secure world. The proposed design leverages the TrustZone Protection Controller (TZPC) to allow the secure world to have exclusive control of I/O and the indicator. When the device is in the secure world, the indicator (LED light) is turned on and the secure UI is shown on the screen to accept input from the user without leaking data to the normal world.

3.4 Related Work

Several existing works [162, 163, 168, 169, 179] protect user's interactions by leveraging TEE. All of them move the UI interaction into the secure world, and overcome the broken binding between the UI and corresponding code by moving the code into the secure world as well (binding is maintained within the secure world). These works require the developer to provide the TA code to be executed, resulting in an app-specific TA. VeriUI [165] protects the login web page by porting the WebKit engine and GUI library into TrustZone. VeriUI is designed to protect the entire web page. However, TruZ-UI targets the granularity of UI view elements that build the entire `Activity`. The existing works require the developer to write TA code and change the app for the TA code invocation. This changes how developers write normal world apps, preventing them from leveraging TEE support by using existing Android components with minimal change to their apps. This dissertation presents a transparent design that allows normal-world apps to leverage TrustZone via existing OS APIs to protect user interaction without the need for app-specific TA code inside the secure world.

3.5 Securing Text Input

This section describes how the user's interaction for text input is protected by seamlessly integrating with the secure-world keyboard UI using a *cross-OS binding*. As described in Section 2.3, Android apps get user's text inputs using a UI element called EditText. When users interact with an EditText, the OS invokes a keyboard. The OS sets up a *binding* between the app and the keyboard. The binding allows the keyboard to send user's typed characters to the app's EditText.

To protect user's interaction with the keyboard, the keyboard UI is moved into the secure world and a binding is provided between the keyboard UI and app's EditText across OSes. Android allows developers to specify a keyboard type when using EditText. To allow the developer to use the existing EditText component to leverage the keyboard UI in the secure world, the design adds a special type called *secure*. The effect of requesting a *secure* keyboard type is shown in Figure 3.4. The app's secure keyboard request is relayed via the modified Android framework service (InputMethodManagerService or IMMS) to a new proxy IME system app (shown as step ①). The

OS sets up a binding between the proxy IME and the requesting app. This proxy IME app communicates with a generic Keyboard Input TA (step ②), resulting in a secure keyboard UI being displayed on the screen with the secure LED turned on. While the secure keyboard is displayed, the normal world does not have access to the screen display or input. In addition to the keyboard keys, the secure UI also displays a hostname (specified with the secure EditText configuration) that represents the destination server for the typed secret. The importance of the hostname is discussed in Section 3.8.

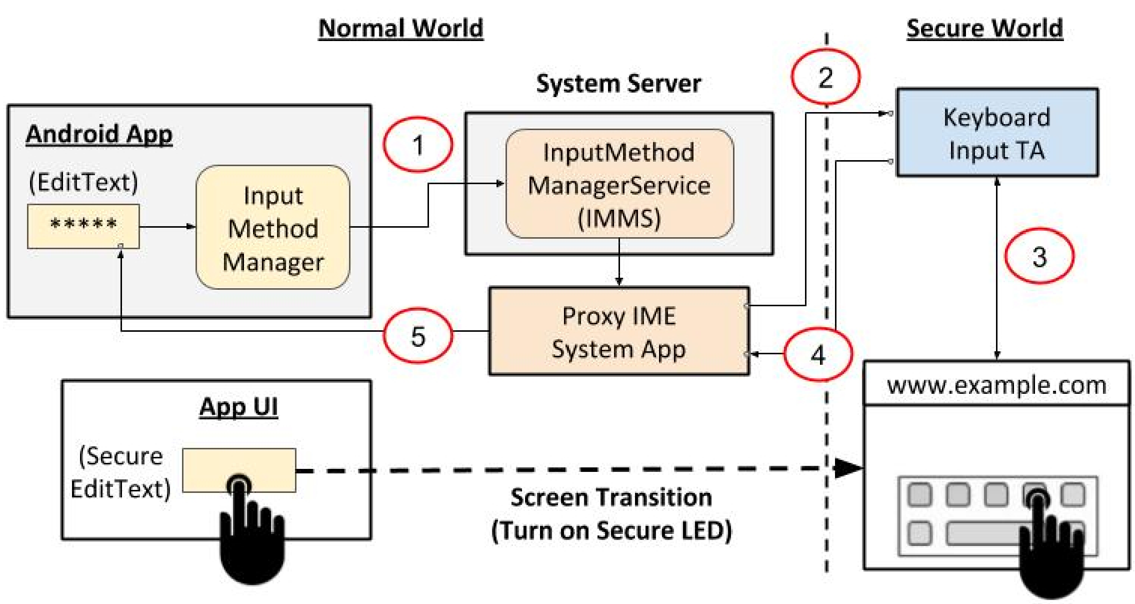

Fig. 3.4.: Seamless Keyboard Binding Across OS

The Keyboard Input TA communicates with the Keyboard UI (step ③) to get the user's input. Once the input capture has finished, the secret is saved in the secure-world memory, which the normal world cannot access, and a reference (corresponding to the saved input) is returned back to the proxy IME app (step ④). The reference is a random string of the same length as the user secret. The proxy IME app uses its binding with the app's EditText to return the reference (step ⑤), made accessible via EditText's standard

API getText() (normally used to get the text typed by the user). A visual feedback is shown in the normal-world EditText by displaying a set of stars. The reference returned from the secure world can support different formats for different scenarios such as passwords, credit card numbers, etc. The design added 1114 LOC in Android (including 634 LOC for a native bridge component to invoke the secure world) and 710 LOC in the TA. The following sections provide further details on the design on individual components in Figure 3.4.

Configuring EditText. As shown in Section 2.3.1, an Android developer can declare an EditText in XML with an inputType. To leverage Truz-UI, the developer will specify the inputType as secure (as shown in Listing 3.1). When using a secure type, the developer must also specify a hostname that indicates which server the secret is associated with. Once the user types a secret in the secure world for the specified hostname, the the secret is only sent to the corresponding server. This is further explained in Section 3.8.

Listing 3.1: Normal vs Secure EditText

```
// Normal EditText
<EditText android:inputType="textPassword" />
// Secure EditText
<EditText android:inputType="secure"
android:allowTo="www.example.com" />
```

Modifications to InputMethodManager. The Android app sends a request for keyboard display to the IMMS via the InputMethodManager. To accommodate the new secure type for EditText, InputMethodManager was modified s.t. it can inform IMMS whether the request was being sent on behalf of a secure or non-secure EditText.

Modifications to IMMS. In order to explain the modifications made to IMMS, this section first expands Figure 2.5 to show how the binding is provided to the IME app by IMMS. As shown in Figure 3.5, when an Android application's UI is initialized, the application process informs the IMMS regarding a window having gained focus. The IMMS creates a session with the InputMethodService [128] in the current IME app. This provides the IME a binding of type InputConnection [104].

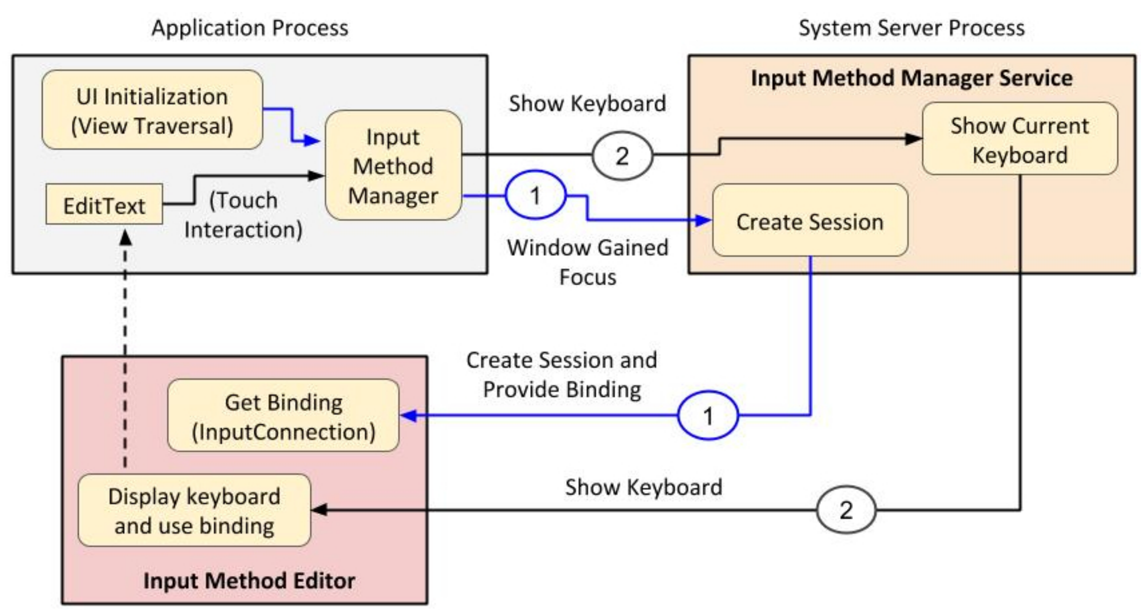

Fig. 3.5.: IMMS Providing Binding to IME

With the modification to InputMethodManager in place, when the user interacts with a secure EditText, the IMMS will be informed of the EditText type. In order to allow secure text input, the IMMS needs to interact with the proxy IME app. The IMMS is aware of one IME at a time (default IME is the one selected via Settings). When IMMS receives the secure request, the current IME known to IMMS is updated to the proxy IME app name. This is followed by re-triggering the window focus gain function in the IMMS. This forces a new session to be created with the proxy IME app, with it being provided a binding to the current application, as shown in Figure 3.6. Once the user is done with secure text input, the current IME in IMMS is switched back to the default IME which allows it to continue providing text input to the app via the provided binding.

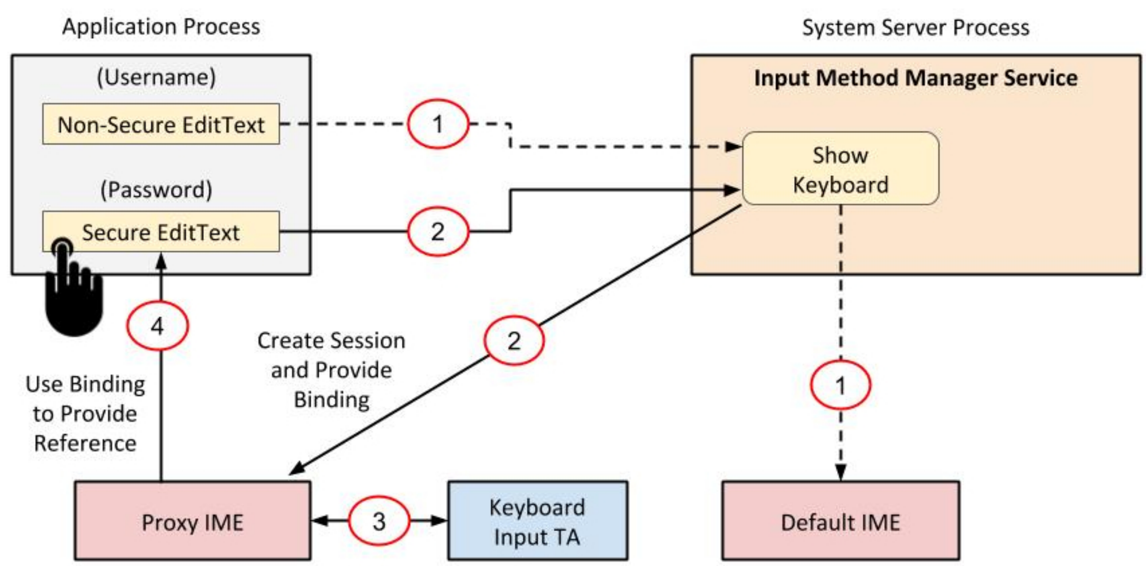

Fig. 3.6.: Switch to Proxy IME for Secure EditText

Proxy IME App. This section further explains how the proxy IME app provides a reference for a user secret typed in the TEE to the EditText in an Android app. As shown in Figure 3.7 (step ①), the IMMS creates a session with the proxy IME app. The

IMMS does this by invoking bindInput() in the InputMethodService of the IME app. This allows the proxy IME to get a binding of type InputConnection using the function getCurrentInputConnection() in onBindInput().

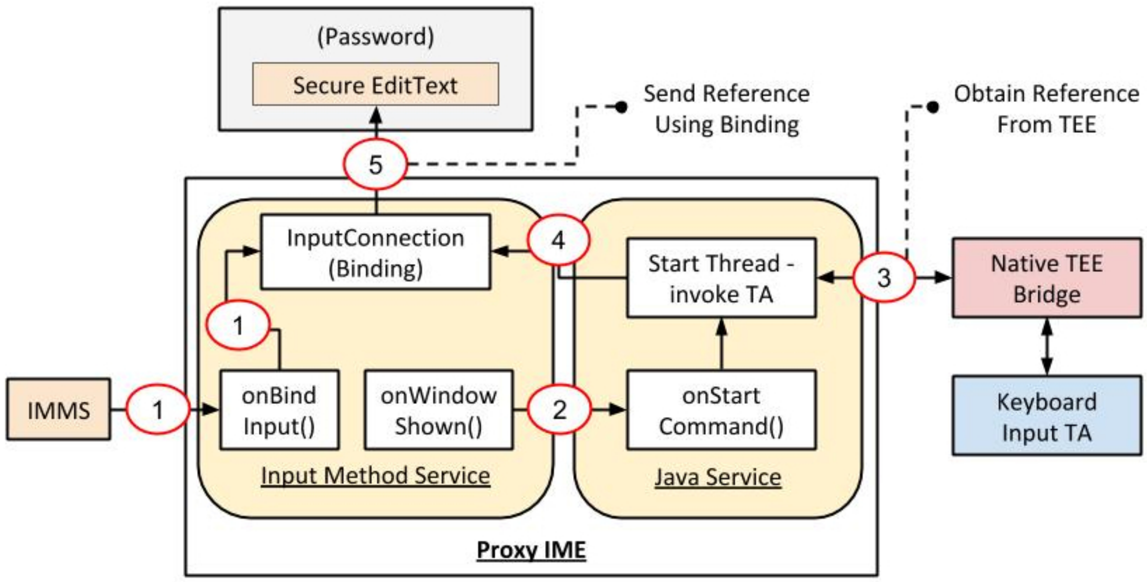

Fig. 3.7.: Proxy IME Commiting Reference Obtained from Keyboard Input TA

To trigger the invocation of the keyboard input TA for secure text input, the design uses the onWindowShown() function in the proxy IME's InputMethodService. onWindowShown() is called immediately before a IME window is shown to the user. Since secure text input does not require any IME UI in the normal world, when onWindowShown() is called in the proxy IME, a new thread is started in a separate Java service, which invokes the keyboard input TA via a native TEE bridge (native daemon process). The TA accepts user input, stores it in the TEE and returns a reference (corresponding to the user secret) to the Java service. The reference is then sent to the EditText in the application using the binding. The Keyboard Input TA will be further explained in Section 3.9.

3.6 Securing Action Confirmation

This section describes how the user's interaction to confirm an action via `AlertDialog` and `Activity` is protected and attested by seamlessly integrating with a confirmation UI in the secure world using *cross-OS binding*. As described in Section 2.4, app developers can ask users to confirm an action by showing a confirmation message and providing the code to be executed based on whether the user approves or denies the message. The OS provides a binding between the confirmation UI and the code provided by the app. Such user interactions face risk in case the normal-world OS is compromised, as the OS can confirm a request on behalf of the user or change the message confirmed before it is sent to the server. To allow the developer to leverage `TEE` support for user's confirmation while using existing components, cross-OS binding is provided along the existing paths for `AlertDialog` and `Activity` components.

3.6.1 Action Confirmation using AlertDialog

As described in Section 2.4.1, an app developer requests a dialog using the `show()` API by providing the message to be confirmed. The app gets back the result via Android input event handling framework which triggers the `onClick()` callback for the dialog button. Figure 3.8 shows the *TruZ-UI* design to allow secure confirmation UI integration for apps. The cross-OS binding is setup between the confirmation UI interaction in `TEE` and the `onClick()` callback in calling app. The design allows the developer to request a secure confirmation UI via `AlertDialog` using the existing API by adding a secure configuration. The secure confirmation UI request is sent by the modified `AlertDialog` class and relayed via a `TEEBridge service` (step ①). This causes the invocation of a generic confirmation `TA` , which results in the switching of the screen to show the secure confirmation UI. The normal-world OS cannot access the display or input at this stage.

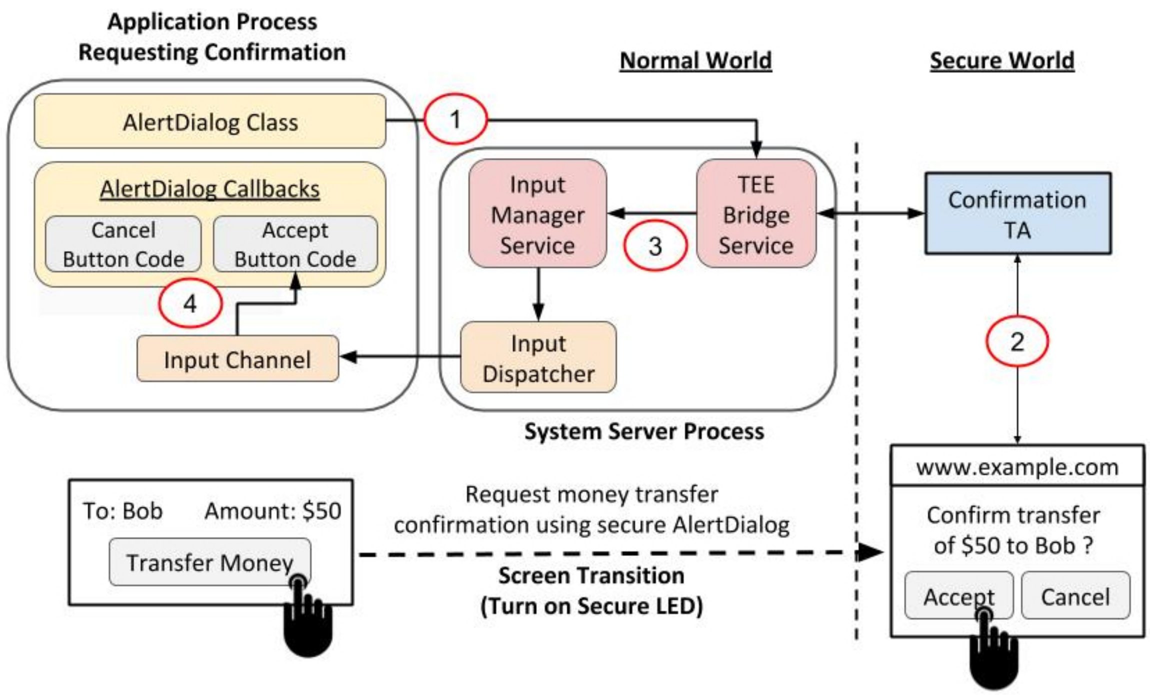

Fig. 3.8.: AlertDialog Confirmation using TEE

The secure confirmation UI allows the user to approve a message and get it signed by the secure world. As part of the secure configuration, the developer also specifies a hostname, which reflects the server for which the message is being attested, and is displayed in the confirmation UI along with the message. The hostname provides the user a context of the requested confirmation. The hostname serves as a reference to lookup the attestation key in the secure world. Each attestation key is bound with the hostname in the `TEE`. The key is setup in the secure world when the user first logs in to the app (discussed in Section 3.8).

Upon user's confirmation (step ②), the message is attested (HMAC signed). The attestation is generated using the key and displayed message, using a nonce to make it non-replayable. In order to improve the user's readability of the message, the developer is allowed to add additional formatting in the message to highlight sensitive fields (e.g., a destination account and amount in case of money transfer). On user's approval, the attestation is returned to the normal-world app. To ensure the confirmation attestation can be returned to existing component, the result is returned via existing callback (`onClick()`) for `AlertDialog`. To return the attestation to the dialog button code, the cross-OS binding uses the event handling framework via an existing service (step ③) in the system server process called `InputManagerService`. Using the API `injectInputEvent()`, a modified `MotionEvent` [133] is sent carrying an attestation (`MotionEvent` is extended to have an extra field called attestation). The event triggers the app button's `onClick()` callback (step ④) where the attestation can be retrieved.

Since the attestation obtained by the app does not contain any user secret, it can be sent to the server using normal-world HTTP/SSL flow. The server can use the attestation to verify the integrity of the request before taking action. The only difference is the addition of the attestation argument in the request. Since the message approved by the user using `TruZ-UI` consists of fixed and variable parts (for example, "transfer $500 to John" contains "transfer .. to .. " as fixed part and the arguments "$500" and "John" as variable parts), the request will have to indicate the fixed and variable parts to allow the server to regenerate the approved user message. By maintaining a strong mapping between the request URI and fixed part of the message, the server can recalculate the attestation based on arguments in the request, and verify it against the attestation in the request to approve the requested action. The presented attestation scheme currently only applies to user understandable message and cannot work for app-specific semantic like GUID, which users cannot understand. The design added 820 LOC in Android and 680 LOC in the TA (the LOC count includes changes for Section 3.6.2). The design used the native bridge mentioned in Section 3.5.

Configuring Secure Dialog Request. In case of dialog, configuring involves adding an additional configuration to provide a common name (as shown in Line 5 in Listing 3.2). Once the user has confirmed the action in `TEE`, the attestation will be returned to the `onClick()` callback and can be accessed as shown in Line 12-13 in Listing 3.2.

Listing 3.2: Secure Dialog Request

```
1  // "this" refers to the containing Activity
2  AlertDialog.Builder builder
3    = new AlertDialog.Builder(this);
4  builder.setMessage("Confirm transfer of $50 to Bob ?")
5      .attestTo("www.example.com")
6      .setPositiveButton("OK",
7       new DialogInterface.OnClickListener() {
8       @Override
9       public void onClick(DialogInterface dialog,
10                                int button)
11      {
```

```
            AlertDialog dialog = (AlertDialog) d;
            String attestation = dialog.getAttestation();
            /* Handle User Approval */
        }
    });
AlertDialog alertDialog = builder.create();
alertDialog.show();
```

3.6.2 Action Confirmation using Activity

As described in Section 2.4.2, an app developer requests a confirmation `Activity` using the `startActivityForResult()` API by providing the message to be confirmed in an `Intent`. The result is received back via the Intent IPC framework. This triggers the `onActivityResult()` callback. Figure 3.9 shows the *TruZ-UI* design to allow secure confirmation UI integration when apps use `Activity`. The cross-OS binding is setup between the confirmation UI interaction in `TEE` and `onActivityResult()` callback in calling app. The design allows the developer to request a secure confirmation UI via `Activity` using the existing API by adding a secure configuration.

To request a secure confirmation, the developer can configure the `Intent` as secure while using the existing Activity API. The request is relayed by a proxy `Activity` (step ①), which is provided as part of a system app. The proxy `Activity` doesn't have a UI; it instead allows transparency as the requesting app can use the existing Activity API.

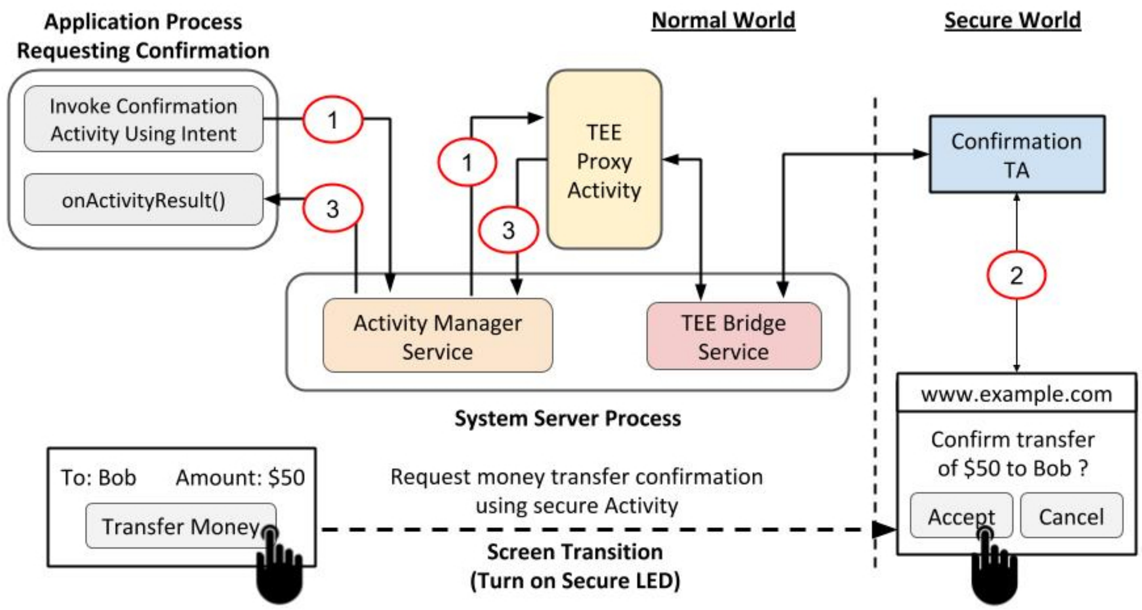

Fig. 3.9.: Activity Confirmation using TEE

The proxy forwards the request to the `TEEBridge service`, which invokes a generic confirmation `TA` . Once the user confirms the action, the message is attested (step ②). The attestation is returned to the caller app via the `ProxyActivity` (step ③), which returns the result to the caller by wrapping the attestation in an `Intent`. This triggers the `onActivityResult()` callback where the attestation can be retrieved.

Configuring Secure Activity Request. In case of activity, configuring involves using a different action in the `Intent`. The action will correspond to the `TEEProxyActivity`. Configuration will also involve setting the message to be attested and the common name as part of the `Intent` (Lines 2-5 in Listing 3.3). The design assumes that the `TEEProxyActivity` will be provided by vendors as part of a system app. The app developer can target an action (like the `SECURE_CONFIRM_ACTION` shown in Listing 3.3) that can be agreed upon by vendors to indicate the proxy `Activity`.

Vendors can ensure that the action is only received by `TEEProxyActivity` from their system app.

Listing 3.3: Secure Activity Request

```
Intent intent
    = new Intent("com.example.SECURE_CONFIRM_ACTION");
intent.putExtra("msg",
        "Confirm transfer of $50 to Bob ?");
intent.putExtra("attestTo", "www.example.com");
startActivityForResult(intent, secure_request);

protected void onActivityResult(int request_code,
        int response_code, Intent data) {
 if(request_code == secure_request &&
        response_code == RESULT_OK) {
   String attestation
        = data.getStringExtra("attestation");
 }
}
```

3.6.3 Attestation Using Android Keystore

In addition to the existing work discussed in Section 3.4, app developers today can use Android's keystore support to have a message attested (signed) using a private key stored inside the secure-world and have a policy that allows use of the key only when user authenticates with a fingerprint [62]. Since fingerprint hardware can only be accessed in the secure world [108], this allows the developer to get a message signed with a `TEE` protected key only when user authenticates in the secure world. This design assumes that the key material is generated before the normal-world OS is compromised. For an app relying on keystore and fingerprint, three problems can occur: (a) user cannot see what message is being confirmed and signed; normal world OS could alter the msg to be signed, (b) normal-world OS can fool the app and its server into thinking that the private key is hardware backed when generated, while keeping the key pair in the normal-world (c) when app requests user authentication for use of the private key, the normal-world OS could provide positive response to the app without asking the user.

In order to not face the above issues, the app's server needs a guarantee that the message is signed using a key visible only to the secure-world, and the user needs a guarantee that message is signed only if the user approves it in the secure world, with normal-world not having any way to alter the message once it is approved. The design discussed in this dissertation provides these features.

3.7 User Involved Access Control

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.

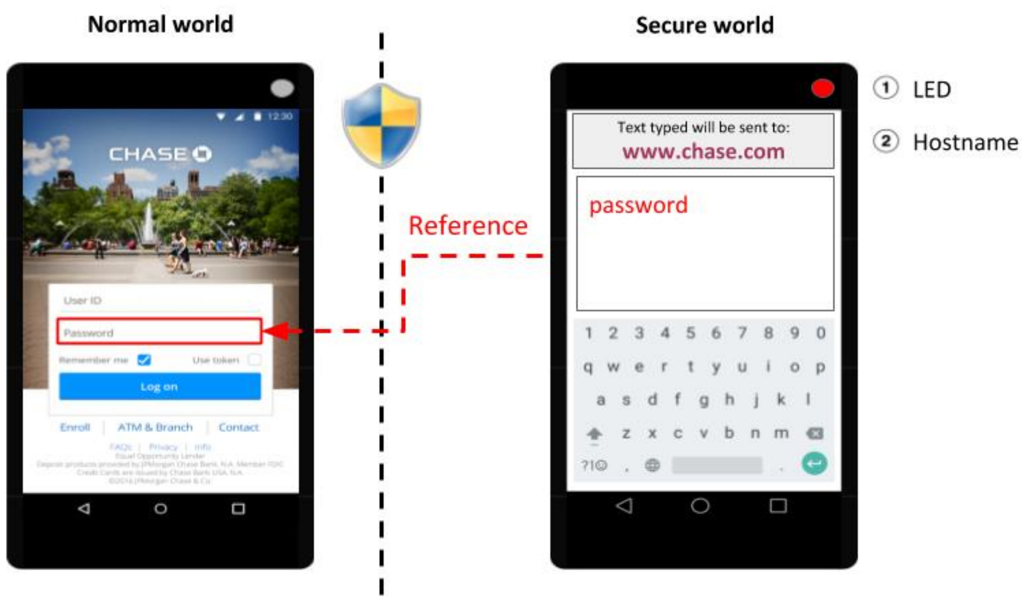

Fig. 3.10.: Truz-UI Context Verification by User

In `Truz-UI`'s 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. When the user needs to get secure text input or secure confirmation using Truz-UI, the user needs to verify two things (a) the secure LED is on, and (b) the common name of the website. This is shown in Figure 3.10. The effectiveness of using a secure LED and a common name has been measured via user evaluation in a related work [180].

3.8 Sending TEE Protected Data to Server

When an app wants to send TEE-protected data corresponding to reference(s) to the server, it will use the existing HTTP/SSL API. To use the secure world to construct an encrypted packet containing the user's secrets, one will need to integrate TEE with HTTP and SSL. This part is not solved in this dissertation, and instead is covered in a related work called TruZ-HTTP and Split-SSL [180]. Using this related work, the reference acts as glue among application, HTTP, and SSL layers. The normal world cannot see the user secret(s), and will get an encrypted packet constructed in the secure world. The packet will be sent to the server via the normal-world TCP/IP stack. This is shown in Figure 3.11.

As stated in Section 3.5, when getting user input in the TEE, the secure UI also displays a hostname (specified with the secure EditText configuration). By typing the secret for that hostname, the user acknowledges that the secret can only be sent to the server with the displayed hostname. The SSL TA enforces the policy of sending the secret only to the corresponding server. Section 3.6 states that when the user confirms an action in the secure world, the message is attested using an attestation key. The key is setup in the secure world when the user first logs in to the app. This is done by the SSL

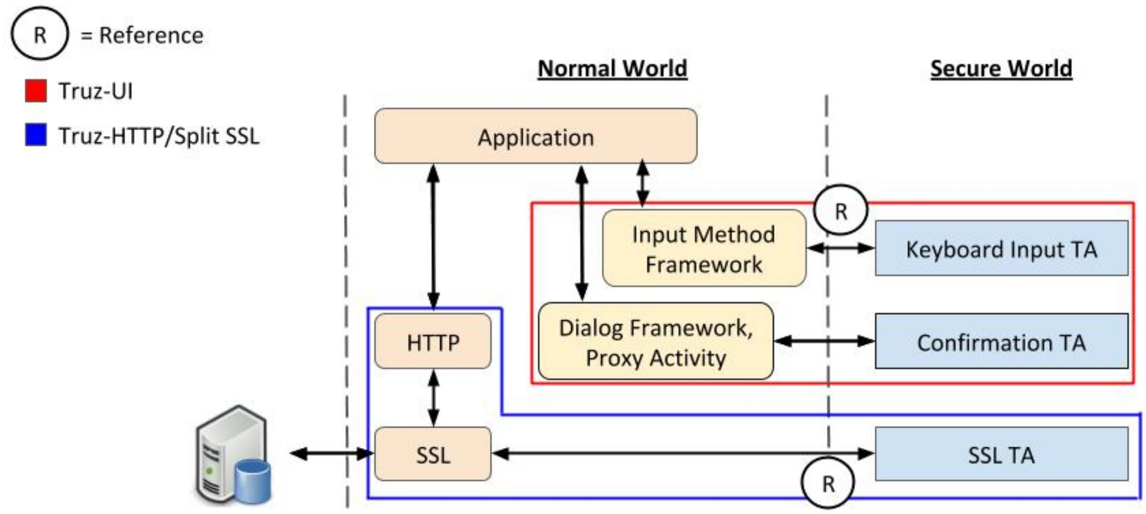

Fig. 3.11.: Connection Between Truz-UI and Truz-HTTP/Split-SSL Works

`TA`. The confirmation `TA` uses the displayed hostname as a reference to lookup the attestation key in the secure world. It can then use the key in the secure world to attest the message confirmed by the user.

3.9 Hardware Implementation

All the commercial Android phones with the TrustZone feature have TrustZone locked down by the manufacturers. In order to test `Truz-UI`, a TrustZone -enabled prototype platform was built that could run Android OS (version 7.0) in the normal world and run OP-TEE OS [85] (version 2.1.0) in the secure world. The prototype was built using the HiKey development board. The prototype uses a TFT LCD panel as the screen. The screen uses the HDMI interface for display and the USB interface for touch control.

Hardware Setup Overview. The hardware implementation provides isolation for the user's input and display. Even though both worlds share the same screen, when the secure world controls it, the normal world cannot access the I/O of the screen. The isolation is achieved at the circuit level. As shown in Figure 3.12, the I/O of the screen is connected to a multiplexer/demultiplexer. The multiplexer takes the HDMI signal from both the worlds and outputs one of the signals to the screen. The demultiplexer takes the touch input from the screen and gives it to one of the worlds. A switch is used to control the multiplexer/demultiplexer. Each world has separate I/O ports that connect to multiplexer/demultiplexer. The control of the switch is accessible to secure-world I/O ports only. To indicate to users which world they are interacting with, the secure world will turn on a LED when the device is in the secure world. The TrustZone Protection Controller (`TZPC`) is configured to allow the secure world to have exclusive control of the switch, LED indicator, and secure-world I/O ports.

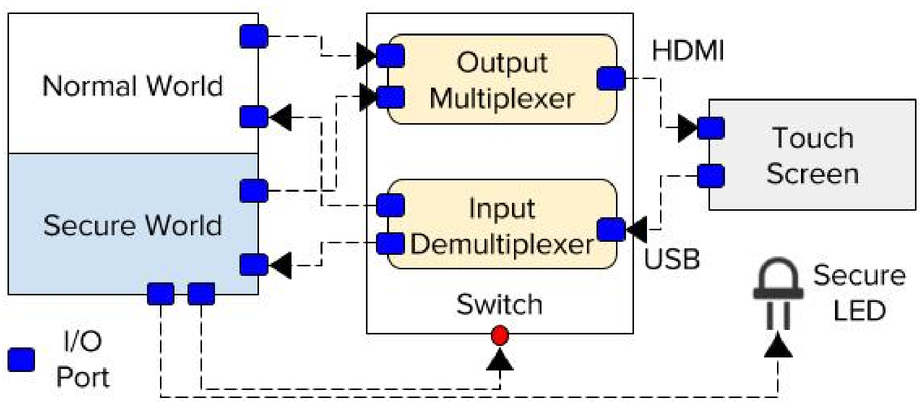

Fig. 3.12.: Hardware Setup Overview for Truz-UI

Hardware Setup Wiring. Figure 3.13 shows the wiring of hardware setup used for testing `Truz-UI`. It follows the overview diagram in Figure 3.12. The touch screen is connected to an input switch (input demultiplexer) and a HDMI display switch (output multiplexer). The normal and secure world run on the Hikey board. The HDMI and USB connected to the Hikey provide display output and input for the normal world. Due to lack of vendor driver support, the secure world cannot directly provide input/output for the touch screen. Due to this reason, the secure world relies on a Raspberry Pi board (interfaced via `UART`). The `UART` is only accessible to the secure world using `TZPC`. The HDMI and USB connected to the Pi board provide the display output and input for the secure world. The UIs for the keyboard input `TA` and confirmation `TA` are provided by Python code running on the Pi board. The `TA`s running in the secure world get results via `UART`. The input switch and the HDMI display switch are controlled by the Pi board, which in turn is controlled by secure world. There is an LED on each switch which indicates which world is in control.

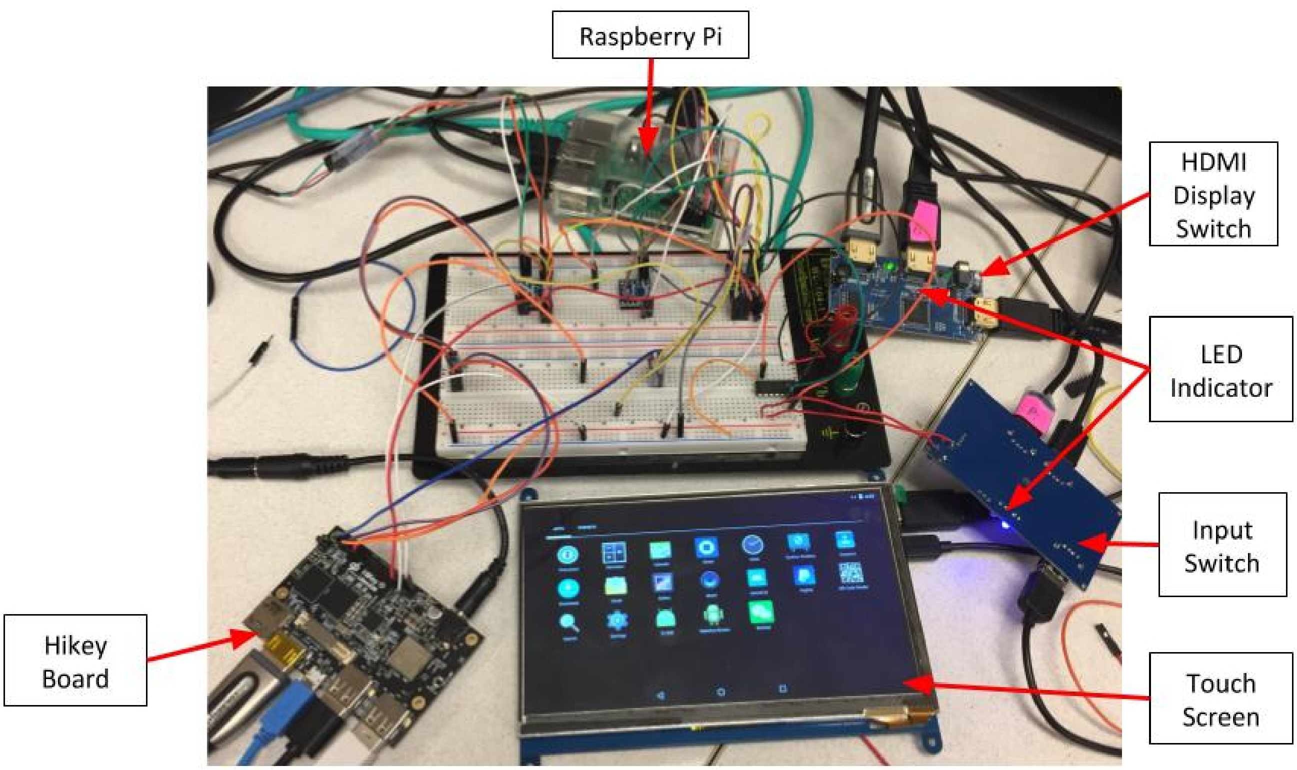

Fig. 3.13.: Hardware Test Setup for Truz-UI

Keyboard Input and Confirmation TA. In order to allow the TA to interact with the Pi board, the TA needs to be able to access the UART driver. The OP-TEE OS version 2.1.0 comes with the PL011 UART driver. To provide access, the prototype modified the userspace library libutee and the OP-TEE OS kernel (adding 150 LOC) to add new system call so that the TA could utilize the driver.

Screen Transition. Since the normal world and secure world share the same screen, when secure text input or confirmation is needed, the secure world takes control of the screen and shows the secure UI. An example of this is shown in Figure 3.14 for the case of secure text input in a banking app. An additional secure LED was added in the wiring of Figure 3.13 to take the picture.

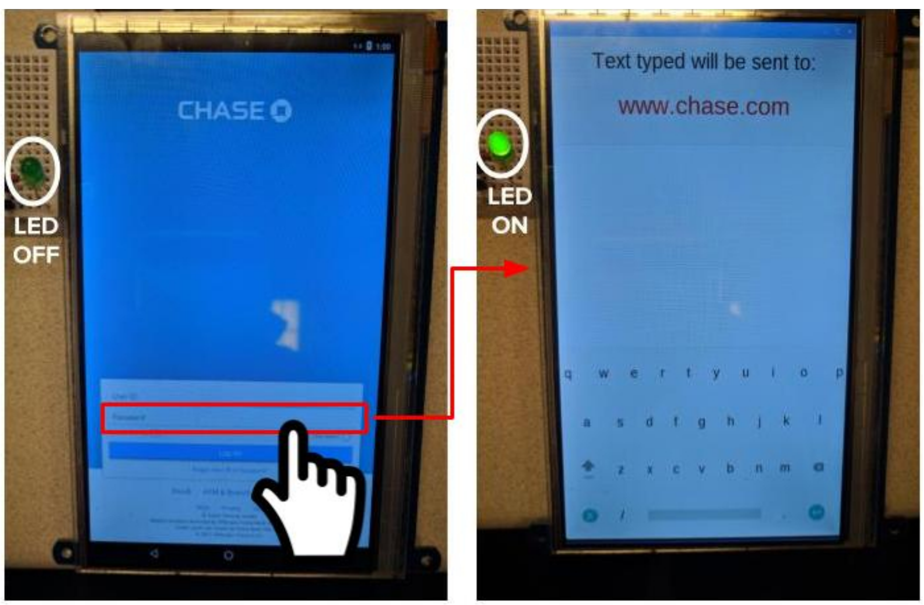

Fig. 3.14.: Screen Transition for Truz-UI

3.10 Security Analysis

This section presents the security analysis of `TruZ-UI`. The design can enforce user's intentions in the presence of either a malicious app or a malicious OS. The analysis uses the stronger attack model and considers the malicious OS as the attacker. The analysis assumes that the TrustZone hardware platform is trusted and the secure boot process has initialized the integrity-verified OP-TEE OS. Hardware attacks, crypto attacks, side channel attacks, and DOS attacks are considered out of scope.

TruZ-UI Secure Text Input Analysis. As discussed in Section 3.5, normal-world apps can leverage the `TruZ-UI` to capture user's secrets (text input) in the secure world. The adversary's goals include monitoring the secret typed, accessing the content displayed, and reading the secret saved in the secure world.

As mentioned in the hardware setup in Section 3.9, the secure world shows the secure UI and gets the screen input through the multiplexer/demultiplexer. The switch controls the USB demultiplexer and HDMI multiplexer. The switch is only controlled by the secure-world I/O ports. The TrustZone Protection Controller (`TZPC`) was configured to allow the secure world to have exclusive control over the switch and secure-world I/O ports. The security analysis of `TruZ-UI` secure text input involves three properties. The first security property is that the secret typed in the secure world cannot be monitored by the normal-world OS. Since the normal world can neither switch the screen USB input nor read the screen input via the secure-world I/O port, the normal world cannot monitor the user's input in the secure world. This prevents keylogging attacks. The second property is that the content displayed from the secure world is not accessible to the normal-world OS.

The normal world can neither switch HDMI output of the screen nor observe the screen content over the secure-world I/O port, preventing it from observing content displayed in the secure world. This helps prevent screen capture attacks. The third security property is that the secret typed in the secure world is never disclosed to the normal world. When a normal-world app uses a secure `EditText`, the secret typed in the secure world is saved in the secure-world memory. Only the reference of the secret is returned to the normal world.

TruZ-UI Attestation Analysis. As discussed in Section 3.6, normal-world apps can request a secure confirmation UI that provides an attestation for user's approved message. The adversary's goals include forging the approval of the message on behalf of the user and forging or replaying the attestation sent to the server.

The security analysis involves three properties. The first security property is that the attestation generated is always tied to the message displayed in the secure world. The attestation is computed based on the message that the user approves in the secure world when the content matches with the user's intention. The second security property is that the normal world cannot forge user's approval of the message that is displayed in the secure world by performing any type of key injection. This is because the normal world cannot access the touch input when the device is in the secure mode (explained in section 3.10). The message is attested in the secure world only when the user approves it. The third security property is that the attestation generated in the secure world cannot be forged by the normal-world OS. The attestation key is generated inside the secure world and only saved in the secure-world memory. The normal world cannot forge an attestation without the keys. Furthermore, a nonce is appended when computing the attestation to avoid replayability.

3.11 Evaluation

In this section, Truz-UI's design is evaluated from three aspects, namely, effectiveness, ease of adoption, and performance. The design was tested on a variety of use cases using real-world applications. Ease of adoption was measured for the developers. To evaluate complete use cases, the evaluation utilized the existing work Truz-HTTP and Split-SSL [180] when data (corresponding to reference) stored in the secure world by Truz-UI needed to be sent to a server.

3.11.1 Effectiveness

To demonstrate the effectiveness, new security features were added to open-source applications by making changes on the client side and server side (if needed). Seven open-source applications were modified, including Elgg [46] and Drupal [44]. To measure the effectiveness in the case of closed-source apps, the OS was modified only for evaluation purpose.

Sensitive file upload. This case study demonstrated how normal-world apps can be enabled 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 [163] requires the app-specific code in the secure world. The open-source app called `Seafile` was used 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. The `Seafile` app was modified to allow the user to enter the secret file content using a secure `EditText`. The user types the file content using the `TruZ-UI` 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). `TruZ-HTTP` and `Split-SSL` are utilized to allow the file to be uploaded successfully to the `Seafile` server.

TrustZone-enabled Android authenticator. To demonstrate that the design can support the Account Manager framework (used to manage Android passwords), an authenticator app for Elgg was written. When a third-party app needs to login to the `Elgg` server, it will ask the Account Manager, which invokes the authenticator app's login `Activity`. This `Activity` uses a secure `EditText`to trigger the `TruZ-UI` 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 using `TruZ-HTTP` and `Split-SSL`. The password reference is saved by the Account Manager, which is not even aware that what it stores is not the actual password. This allows Account Manager to manage the authentication requests for third-party apps without storing the actual passwords in the normal world. The design requires no change to the Account Manager framework.

Attested post. `Drupal` was installed on an Ubuntu server and the handling of the post content type was modified to verify attestation. The `Drupal` Editor app [45] was used as a client. The app was modified 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. The proxy Activity (refer Section 3.6.2) was utilized 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. `TruZ-UI` allows developers to protect any text-based secret that can be typed in apps. This was evaluated by using seven different open-source apps, including Friendica, Elgg, Drupal, MustardMod (with GNUSocial), Kandroid (with Kanboard), Redmine and Seafile. Minimal changes were made 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.

3.11.2 Ease of Adoption

The ease of adoption was evaluated by measuring how much effort developers need to make to add TrustZone support to their apps. The evaluation was conducted using both open and closed-source apps. For open-source, both the client and server code was downloaded from public Github repositories [47]. For closed-source, apps were downloaded from Google Play. To ensure their diversity, apps were downloaded from different categories, including shopping, traveling, productivity, finance, medical, business, food, etc.

Seven open-source apps were modified, by either adding new features to them (e.g., attestation) or leveraging TrustZone to protect their existing features (e.g., login). The time spent on the modification and the number of lines of code (LOC) modified for each app was recorded. Table 3.1 shows the result. 1 LOC for `TruZ-HTTP`, 2 LOC for secure `EditText`, 4 LOC for secure confirmation. As shown in Table 3.1, for apps to protect their login credentials, only 3 lines of code are modified on the client side and the time spent on making the changes was within an hour. For server-side changes, 4 lines of code were needed to extract the secret data from the `HTTP` request. In case of attestation, the attestation logic varied depending on what to attest. The overall change on the server side was less than 20 lines of code.

Table 3.1: 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 apps from the market, closed-source apps were enabled to leverage TrustZone. To protect users' secret in the secure world, the apps were modified to protect user's sensitive data, including passwords, credit card numbers, and files containing a secret. The closed-source apps were repackaged by configuring some selected EditTextin their layout files, so when sensitive data needs to be provided by users, the TruZ-UI keyboard is invoked and the data are typed inside the secure world. To protect users' confirmation in the secure world, the confirmation UI name (Activity or Activity containing AlertDialog) and the corresponding message was hardcoded in a configuration file. The system used 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, a proxy server was setup to verify the attestation. The secure world shares the SSL keys with the proxy server (using existing work [180]), so it can intercept all the SSL traffic. Configruration files were created to inform HTTP and SSL layers (based on [180]) whether the data to be sent to the server contains the TEE-protected secret, attestation message or attestation keys. All configuration files and the proxy server are only for demonstration purpose. If the apps could be modified, such files are not needed.

31 apps were collected, including Chase, Github, Southwest Airline, Piazza, Priceline, Box, Poshmark, Listonic, Dropbox, MediaFire, Applebee's, Discover, Secure Cloud Storage, etc. 15 apps were used for TEE-protected login, 5 for TEE-protected payment, 2 for TEE-protected file upload, and 9 for attestation. The results are shown in Table 3.2. 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 the source code was available for these failed cases, they could be made to work.

Table 3.2: Evaluation Result for Closed-Source Apps

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

3.11.3 Performance

Experiments were designed to measure the round-trip time for code to secure UI invocation and back. The overhead (average over 20 trials) of the implementation adds over the normal case by not counting the drawing time or the user's input time. The `TruZ-UI` keyboard integration adds 123 ms overhead. The confirmation UI integration adds 53 ms overhead. In `TruZ-UI` keyboard integration, the overhead is caused by the interaction between the proxy `IME` app and the keyboard input `TA`. In the confirmation UI integration, the overhead is caused by the interaction between the `TEE` bridge service and the confirmation `TA`. Overall, the delay caused by the overhead for the `TruZ-UI` is barely noticeable when users interact with `TruZ-UI`.

3.12 Publication

The `Truz-UI` design has been published in 2018 as part of a joint work in the paper titled *TruZ-Droid: Integrating TrustZone with Mobile Operating System* [181]. The dissertation author was the second author in this paper publication.

4. TRUZ-CALL: SECURE VOICE INTERACTION FOR VOIP CALLING

4.1 Problem Overview

Mobile phones are one of the most common devices used by people today, with the basic function of calling another person. In recent years, VoIP apps such as Signal [25] and Whatsapp [14] have become popular ways for making a call. Unfortunately the mobile OS platforms (like Android) on which these apps run have made the use of VoIP apps more risky in terms of user privacy. The problem is also compounded by the fact that various actors are trying to compromise Android OS including hacking groups [55] and nation states [82]. The ever present risk of mobile OS compromise can limit one of the important rights in human society i.e. freedom of speech. In context of mobile phones, this translates to being able to call anyone and talk on any subject without fear of someone else listening on the call. Today different types of users need to have a secure means of calling, including activists, journalists, government employees etc.

A high level view of how a VoIP call works in shown in Figure 4.1. Once the VoIP call is established, a caller/callee provides audio input and receives audio output via the device's audio peripherals. A compromised OS can listen to user's conversation during a VoIP call. TrustZone can be leveraged to protect user's voice interaction because of the hardware level isolation it offers.

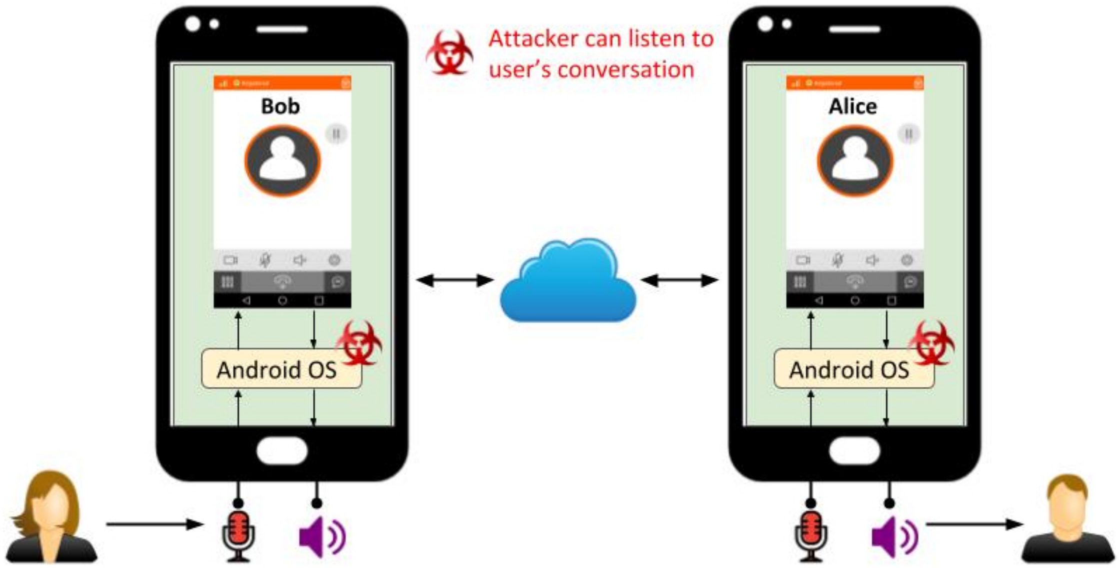

Fig. 4.1.: VoIP Call Overview

In order to protect user's voice interaction during a call, a VoIP app show be able to leverage TrustZone to establish a end-to-end encrypted VoIP call. This dissertation states the following problem: *How can we allow a VoIP app to transparently leverage ARM TrustZone to protect users conversation from an untrusted OS during a VoIP call ?* The design should be transparent to VoIP apps. It should allow the VoIP app to use existing OS APIs used and VoIP protocols. The design should require no change to the VoIP infrastructure.

VoIP apps contain several stages that work in parallel as a pipeline, each stage feeding data to the next (Figure 4.2 (left)). The app uses OS APIs to fetch audio, processes the audio, and sends out packets over the network (reverse flow for incoming packets). The app uses a VoIP protocol like `SRTP` to encrypt and calculate `HMAC` for the audio payload (in `RTP` packets), and send the encrypted payload to the callee device. A transparent design involves preserving the relative structure of the VoIP app software stack, as it affects the way developers write VoIP apps. This dissertation focuses on the essential VoIP app stages of audio I/O, RTP packet construction / parsing, SRTP, and network I/O.

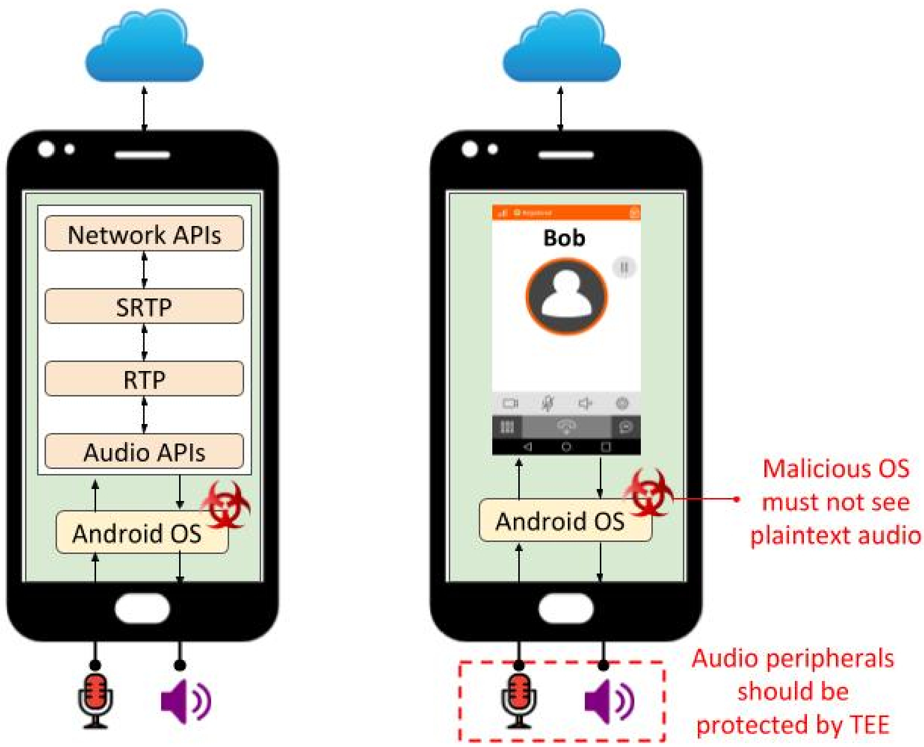

Fig. 4.2.: VoIP App Stages and Secure VoIP Requirement

Since the design would leverage TrustZone, it should minimize the TCB in the secure world. This includes providing generic TA support so that app-specific TA code is not required in the secure world. During the TEE protected VoIP call, the audio peripherals should be controlled by the secure world and the user's conversation audio should be protected from the normal-world OS (Figure 4.2 (right)).

A challenge that is encountered in designing a system like Truz-Call is latency. Since VoIP is a real time system, if the normal world stack invokes TEE at one or more points, it will add computation time to the VoIP call. Any additional time will add latency and will thus affect voice quality. The design should reduce end-to-end latency overhead.

Another challenge is the hardware setup to do prototype evaluation. In TEE research, interfacing hardware peripherals like mic and speaker with the TEE OS on a development board can be challenging for non-hardware experts with limited resources. In order to evaluate the design, a hardware setup needs to be used that allows easier prototyping.

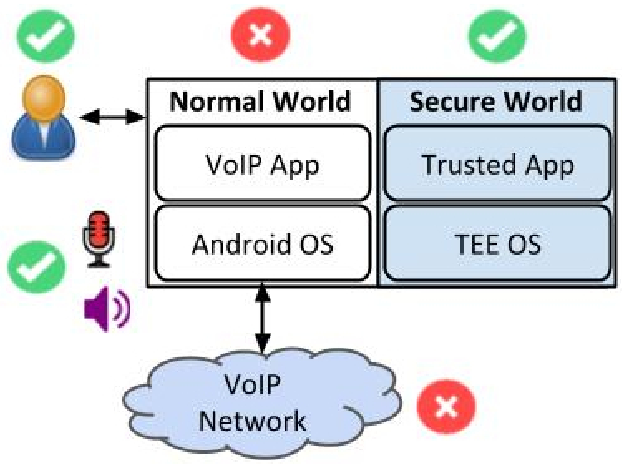

Fig. 4.3.: Voice Interaction Threat Model

Threat Model and Assumptions. The normal world (including Android OS and the VoIP app) is not trusted. The secure world, including the TEE OS and trusted applications (TA), is trusted. The user using the device is trusted. The device hardware, including the audio peripherals (mic and speaker), is trusted. The VoIP network is not trusted, although Truz-Call does not try to protect against network based attacks. Truz-Call is targeted for users who want to securely call friends, family or someone they know personally or have met before. It does not cover key exchange done by VoIP apps (at the beginning of the call), which is why it cannot be used to call an unknown person. To use the design discussed in this dissertation, two users need to exchange a secret phrase using a secure side channel (this will be used to derive the key). Truz-Call can be extended to add key exchange using the TEE by splitting protocols like DTLS [19]. It is also assumed that the user wants to use Truz-Call for a one-to-one call, and not for conference calling.

4.2 Factors Influencing TEE Integration Design

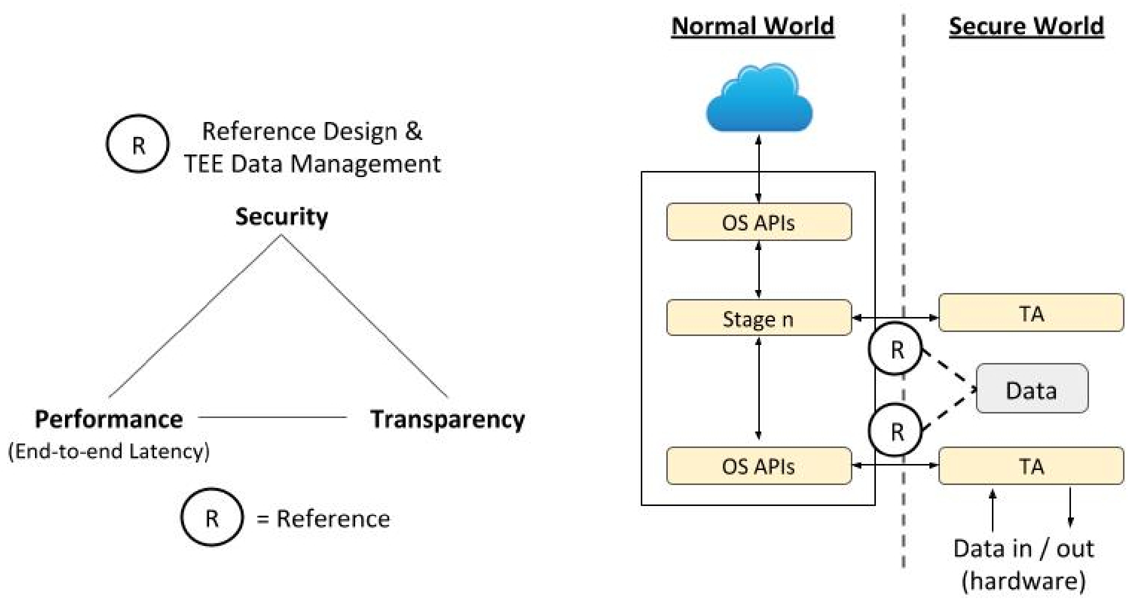

Fig. 4.4.: TEE Integration Design Factors

When designing a secure solution on mobile platform by integrating TEE into a normal world stack (Figure 4.4 (right)), three factors need to be balanced (Figure 4.4 (left)). The solution needs to provide security, i.e. preventing the compromised normal-world OS from accessing sensitive data. This is achieved using references. Alongside references, there will data associated with the reference in secure-world memory. The design (structure) of the reference impacts transparency in the normal world stack, as stages of the normal-world stack operate on the reference data. TEE is integrated and invoked at multiple stages in the normal-world stack. The number of normal-world `TEE` integration points and the way data is managed in the `TEE` impacts overall latency.

4.3 Related Work

The idea of having a secure VoIP call on an untrusted OS has been discussed before in the work *"A Hardware-Assisted Proof-of-Concept for Secure VoIP Clients on Untrusted Operating Systems"* [156]. This existing work has been done on a Xilinx board, which includes a PS section and PL section (FPGA). The PS and PL sections are analogous to normal and secure world respectively. The work is intended for devices like VoIP phones (handset). They used the Linphone app [5] for testing and modified it such that for incoming `SRTP` packet, the header information and payload is forwarded to secure hardware, and for outgoing packet the `SRTP` header and encrypted payload are sent from secure hardware to the normal world. There are several differences between this existing work and `Truz-Call`: (1) Commercial mobile phones don't rely on FPGA; instead they ship with ARM boards that have TrustZone. The existing work does not address any challenges related to leveraging TrustZone for secure VoIP. (2) Xillinux OS does not reflect mainstream mobile OS like Android. The existing work does not address leveraging TrustZone in mobile OS audio stacks to allow existing Audio APIs to be used. (3) A VoIP app has a flow for handling audio packets. In the existing work the `RTP` layer has been eliminated from the normal-world app flow as the design forwards header/payload with secure hardware at the `SRTP` layer. This breaks the relative structure of the software stack used to implement a VoIP app. The design does not utilize Audio

APIs in the normal world to record/play audio data which changes the way developers write VoIP apps. Moving header generation/parsing functionality into the secure world increases the TCB as only part of the SRTP layer remains in the normal world. TruzCall's goal is to maintain the relative structure of the essential parts of the software stack for a VoIP app and avoid moving unnecessary components into the TEE. In summary, the existing work [156] is not transparent (breaks app stack structure and makes the app no longer use OS audio APIs) and has a large TCB.

DRM can provide secure audio/video playback using TEE, but the reference design and TEE data management used in DRM do not apply to VoIP. TrustCall [143] is a commerical product that leverages TEE for secure calling [40]. Based on the information available online, it is not designed for transparency to Android VoIP apps. It only works for the TrustCall app [144]. It also relies on TrustCall specific TA being present inside the TEE [31, 32]. Truz-Call is designed to be transparent to any VoIP app that wants to use existing OS APIs for a SIP/SRTP based call. Truz-Call provides generic TA support, avoiding having app-specific TA code inside the TEE.

4.4 Secure VoIP Calling Problem Scope

In order to design Truz-Call, the problem scope needs to be narrowed down. The problems scope pertains to the type of protocol support to be provided and whether all VoIP app stages should be supported.

4.4.1 Protocol Support

VoIP apps can be written to conduct a call in plain text (using protocol combination like SIP + RTP) or can choose to use end-to-end encryption to protect the user's conversation. Common protocols used by VoIP software for secure calling using end-to-end encryption can be found at [174]. From the data available for protocols used by apps, a common protocol for VoIP with open source implementation is SRTP [10] using SIP [6] for call initiation. Popular apps like WhatsApp rely on SRTP [149]. Instead of providing TEE support to protect calls for all VoIP apps, this dissertation focuses on the problem scope of providing TEE support for apps that already provide end-to-end encryption (using SIP and SRTP), but face a privacy risk due to a compromised OS. Section 2.5 provides more information on how VoIP apps setup a call using SIP and SRTP.

4.4.2 VoIP App Computation Stages

For a VoIP app using SIP and SRTP, audio processing is conducted in several stages as shown in Figure 4.5. The stages comprise audio I/O, audio computation (like resampling, compression), RTP packet construction / parsing, SRTP, and network I/O.

One of the stages in Figure 4.5 is marked as *computation*. To improve audio quality and reduce bandwidth requirements, a VoIP app applies several types of additional computation on the audio data. For audio data read from the mic, computations applied can include read resampling (downsampling), volume adjustment, equalization and

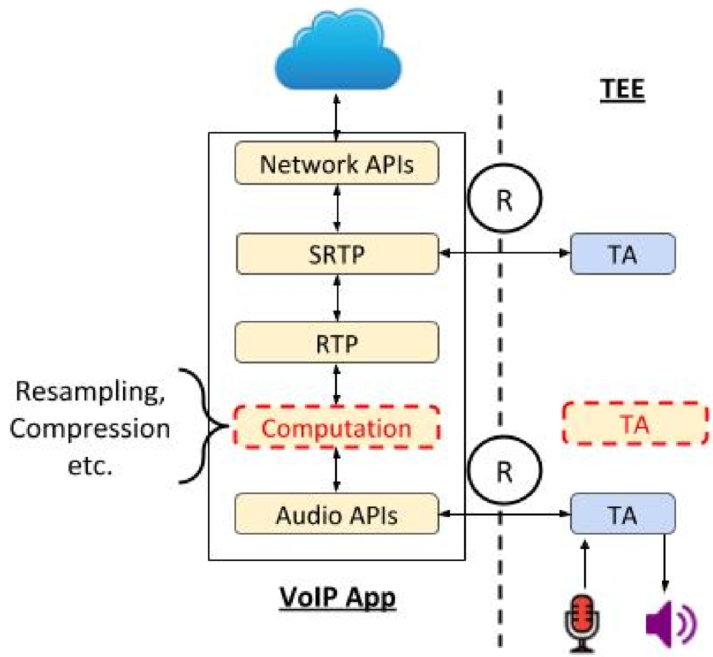

Fig. 4.5.: VoIP App Stages

compression. Before playing received audio, applied computations can include decompression, volume adjustment, equalization, and upsampling.

There is a performance penalty involved in supporting the additional computation stages. End-to-end latency increases with every stage that uses `TEE` (due to invocation time). Supporting the additional computation stages will add performance overhead. It will also adds to the TCB in the secure world. For a design using references for audio data, the additional audio computations can tamper with the reference data. For the problem scope of `Truz-Call`, the additional computation stages are disabled. The design focuses on the essential stages of audio I/O, `RTP` packet construction / parsing, `SRTP`, and network I/O. The design sacrifices audio quality for security.

4.5 Main Idea

Figure 4.6 shows the main idea of `Truz-Call`. The various stages in a VoIP stack work in parallel as a pipeline, each stage feeding data to the next. The audio pipeline in the VoIP app consists of some essential stages. To allow the VoIP pipeline to maintain its existing flow while keeping user's conversation audio in the `TEE`, the design invokes `TEE` at the stages for audio API usage and `SRTP`. This allows the use of the existing relative structure of the software stack.

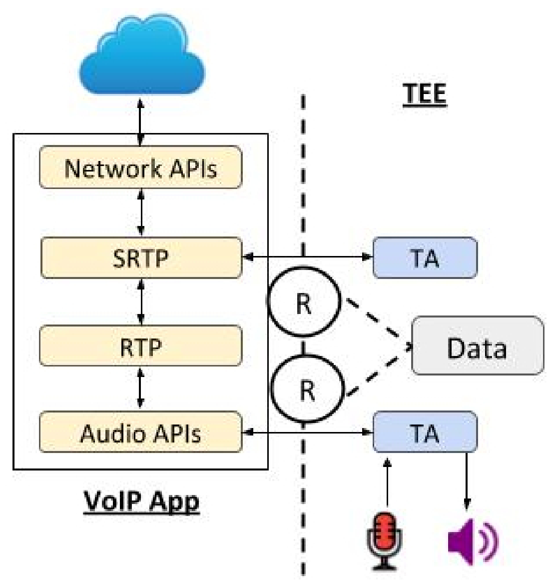

Fig. 4.6.: Truz-Call Design Overview

At the beginning of the call, `TEE` takes control of the audio peripherals. This can be done using TrustZone hardware features and has been done in other works like SeCloak [161]. In order for the `TEE` invocation at several stages of the VoIP stack to work together, the design uses a reference design pattern. When the VoIP app asks for audio using existing APIs, the `TEE` invocation provides it a reference to the real audio data (saved in `TEE`) via the existing normal-world OS audio APIs. The app then proceeds with preparing the RTP packet. When the flow reaches the SRTP layer and it needs to encrypt the data in the RTP payload (which is a reference). The design invokes the TEE and passes the audio data reference. The TEE encrypts the data corresponding to the reference and returns the encrypted payload and HMAC to the SRTP layer to allow the VoIP app flow to continue. This way only essential cryptography operations for SRTP are moved into the TEE. The reverse flow happens for packets received by the device for playback.

The Truz-Call design has been tested on the open-source VoIP app Linphone [5]. It should be emphasized that the changes made to the Linphone app are within the various libraries used by Linphone. The app is composed of several modules, including libraries for SRTP, RTP [88], SIP [68] and audio I/O [77]. A different VoIP app using the same libraries should be able to use Truz-Call 's design. The changes made in the audio framework would be applicable to any VoIP app.

Reference Design Constraints. The OS Audio API expects an audio payload. The follow up stages of RTP and SRTP also operate on audio. Given the design returns references from the TEE, the normal-world OS should not able to deduce the plain text audio from the reference. The normal-world OS can be allowed to know length of the audio.

TEE Data Management Constraints. A VoIP call is a two-way call, i.e. it involves record and playback. The two way flow of audio must happen in parallel. Any dependency between record and playback will add latency. The design should reduce the number of operations in the TEE to reduce latency. Also, the design should reduce the data size passed into the TEE to reduce latency.

Record and Playback Behavior. RTP protocol in the normal-world app uses packetization feature to send data. The TEE data management should be able to accommodate this behavior. Also the RTP protocol uses jitter handling feature to playback data. The TEE data management should be able to accommodate this behavior.

4.6 TEE Invocation and Data Encoding

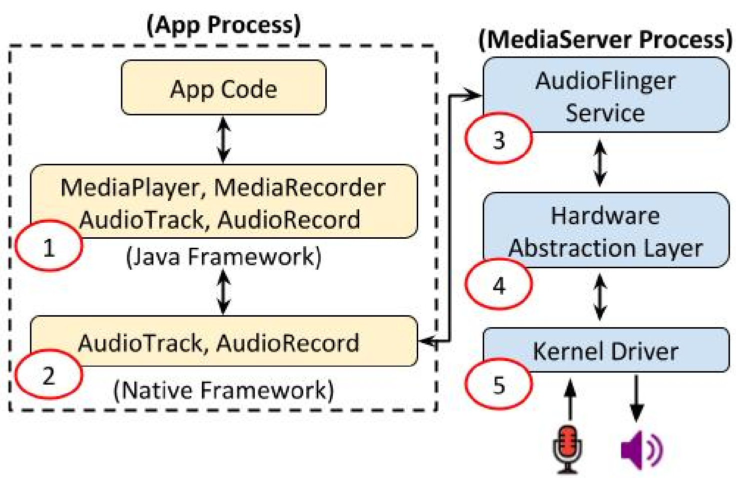

Fig. 4.7.: Android Audio Architecture

This section discusses how TEE is leveraged by various stages of the normal-world VoIP audio pipeline. It also discusses what encoding is used by the TEE to convey the audio data to the normal-world pipeline. Figure 4.7 shows the architecture of Android's audio stack [66]. An Android app can use various Java APIs for Audio I/O, all of which use the same underlying native framework. This communicates with the underlying

AudioFlinger service (Android's sound server [67]). In order to protect the user's conversation during a VoIP call, TEE needs to be leveraged to provide the VoIP app the user's audio without ever releasing the plain text audio from the secure world. The user's audio can only enter the normal world in an encoded form. The question becomes at which layer in the normal-world stack should TEE be invoked for audio. In Figure 4.7, Audioflinger (3) is responsible for resampling [90] and mixing audio streams [67], as well as applying effects. If TEE is used at this layer, we would have to make sure that there is a path that doesn't alter the data obtained by or to be given to the TEE, in order not to break the audio encoding. Using TEE at (4) or (5) will incur the same issue as data will pass through the AudioFlinger. Layer (1) provides the app with several APIs to read/write audio. To allow the VoIP app developer to use any API for Audio I/O, the design decision was to use TEE at layer (2).

4.6.1 Audio Data Encoding

Once the TEE invocation point for the audio framework has been identified, we have to decide an encoding to provide audio data to the normal world. The data provided to the native audio framework can be encrypted by the TEE. In this case, the cryptographic operations done in the app's SRTP layer will become redundant; the audio data will be encrypted twice. It will also add latency to the VoIP flow because of the additional time spent encrypting the audio data again. One way to handle this design option would be to disable the operations done in the normal world SRTP layer, but this would disable an essential stage of the app flow. The goal of Truz-Call is to preserve the relative structure of the essential layers in the VoIP app, including the SRTP layer.

In order to allow the app to still use the SRTP library for encryption and HMAC, the design does not provide encrypted data to the native audio framework. When the app requests audio data, the native audio framework gets a reference for the audio. The reference is a string with the same length as the requested audio data. The RTP layer prepares a packet containing audio reference(s) as the payload. When the SRTP layer needs to encrypt the packet, it invokes the TEE which encrypts the audio data corresponding to the audio reference(s) in the RTP payload and calculates the HMAC for the RTP packet. Once the TEE returns the result, the SRTP flow can continue to send the packet out. On the receiving device the reverse will happen. The SRTP library will invoke the TEE to get an audio reference corresponding to the RTP encrypted payload, with the decrypted audio staying in the TEE. When the native audio framework needs to play the audio, the reference is given to the TEE which plays the corresponding audio. Figure 4.8 shows the TEE invocation points (the RTP layer is omitted).

4.6.2 Independent Audio Pipeline Stages

Given two types of TEE invocations (by the native audio framework and by the SRTP library), Truz-Call needs to make sure that the TA logic and corresponding data for these invocations is handled in a way such that there is no bottleneck created in the normal-world audio pipeline. To handle the two types of TEE invocations, the design needs to allow sharing of data via a common memory space between the corresponding

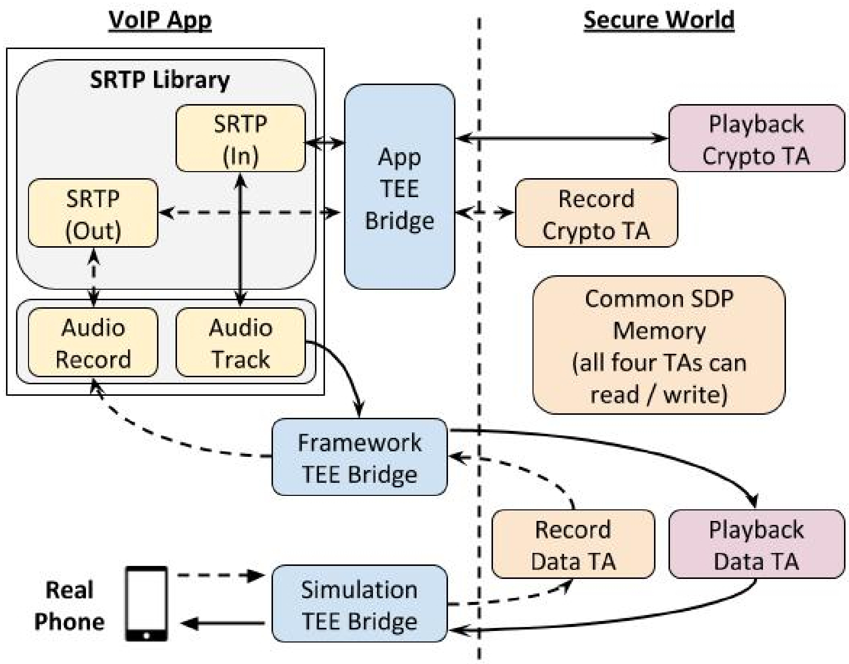

Fig. 4.8.: TEE Invocation by Audio Framework and SRTP

TA logic. The plain text audio in TEE must be accessible to the cryptographic logic when SRTP library provides it a reference and conversely the audio data decrypted must be accessible to the TEE audio playback logic when it is provided with a reference by the native audio framework. When a TA is invoked, it can access three types of memory including stack, heap and shared memory. Only data in heap and shared memory can retain its value across multiple TEE invocations. TEE provides two types of shared memory, namely unsecure shared memory (used by normal world to pass arguments) and secure shared memory (not visible to normal world, but visible to TEE components). The two candidates to keep plain text audio in common memory are heap and secure shared memory. Heap cannot be used for this design because our design constraint demands reduced latency. In order to use heap as a common memory, the TEE logic corresponding to different normal-world stages will need to belong to the same TA because the TEE OS provides isolated heaps for different TAs. This would require multiple normal-world pipeline stages to invoke the same TA, which would require the TA to be configured with TA_FLAG_MULTI_SESSION [52]. This would make the TA invocations serialized i.e. different normal-world stages won't be able to call the TA simultaneously (the call from one stage will have to wait for the call from the other stage to finish). This would create a performance bottleneck and add latency. Therefore the design uses secure shared memory to provide common memory for plain text audio in the TEE. OP-TEE provides this feature via secure data path (SDP) [51]. It allows a secure pool of memory to be allocated in the TEE with normal world having a reference to this memory. The SDP reference is made available to the TEE bridges in the normal world. The normal-world bridges pass the reference when invoking corresponding TAs so that the common memory containing the plain text audio is accessible in the TA logic.

4.6.3 TEE Bridges and TAs

Figure 4.8 shows three TEE bridges and four TAs inside the TEE. The TEE bridges are native daemons (running with root privilege) that allow normal world components to invoke the TAs. The App TEE Bridge allows the SRTP layer (Java code) to invoke the Record Crypto & Playback Crypto TAs responsible for cryptographic operations (encryption and HMAC) in the TEE. The Framework TEE Bridge allows the native audio framework to invoke the Record Data TA responsible for collecting audio data and providing reference for audio data, and Playback Data TA responsible for playing out audio data corresponding to the provided references. The Simulation

`TEE Bridge` allows the design to record & play audio using a simulation environment by using a real phone to provide the audio hardware (discussed in Section 4.12).

`Truz-Call` sends and receives audio references to/from the `TEE`, which means each time the normal world needs audio or wants to play audio a `TEE` invocation will be needed. Each invocation from the normal world involves opening a session with the `TEE` OS. Each `TEE` invocation session consumes some memory in the `TEE` OS due to saved state. At the same time the `TEE` environment is only assigned a limited amount of memory [56]. If the normal world keeps opening sessions based on the requirements of an on-going VoIP call, the `TEE` OS will exhaust its memory and deny any more `TA` invocations which will stop the secure call. Closing a session and opening it again for each `TEE` invocation will contribute to latency. To solve this issue we make our `TEE` bridges *persistent* by reusing `TEE` sessions. A bridge only initiates one `TA` session (with each `TA` that needs to be used) at the beginning of the call. All other `TEE` invocations via the bridge reuse the persistent session. This way the VoIP call can use `TEE` without exhausting its memory and can go on for any duration.

4.7 VoIP Call Initiation

This section discusses how `Truz-Call` handles the VoIP call setup. As mentioned in Section 4.1, the design assumes that the user wants to call a known person as key exchange is not handled using the `TEE`. Before a secure call is setup, the caller and callee need to exchange a secret phrase using a text entry that will be input using a secure UI. This has been addressed in other works [162, 163, 181, 182]. When the user types in this secret phrase, the user also enters the SIP address of the callee. The secret phrase and the associated SIP address are saved in the TEE trusted storage [34].

The user will initiate the VoIP call using the app's UI in the normal world. The call will need to first establish a connection using SIP using a SIP INVITE packet to the Linphone server. Before sending this packet, Truz-Call invokes a TA and passes the callee's SIP address. The user will be shown a confirmation UI asking whether a secure call should be initiated. Once the user approves, the TA will lookup the secret phrase associated with the SIP address. Both the SRTP and SRTCP protocols need two sets of master key and salt (for send and receive directions). The TA concatenates the secret phrase with a random string generated using the TEE random device. The TA calculates the master keys and salts by concatenating this new string with four fixed values and generating SHA-256 hashes. Each master key needs to be 16 byte and master salt needs to be 14 byte, so each key + salt pair is 30 bytes (first 240 bits of the hash is used). The TA keeps the master keys and salts in memory. Next the TEE would take control of the audio peripherals on the device so that normal world cannot access the user's conversation audio during the VoIP call (in Truz-Call 's testing a simulation based environment is used, but in an actual product TEE will need to control the audio hardware). A secure LED light (only accessible to the TEE) will be turned on which allows the user to know whether the audio hardware is under TEE's control. The TA returns control to the normal world and returns the random string that was concatenated to the secret phrase. The SIP flow continues and uses this random string as its CALL-ID [6]. The CALL-ID will be conveyed to the receiving device when it receives the SIP INVITE so that it can generate the corresponding master keys and salts. Once SIP has established a connection, the app will use the RTP protocol to communicate with the other device on the call. RTP RFC [8] dictates that the initial value of the sequence number should be random. After SIP has established a connection, a TA is invoked which generates a random number using TEE random device. This number is returned to the normal world and is used as the initial sequence number. Section 4.9 discusses how the TEE checks whether the normal world has obeyed to use the sequence number given by the TEE.

As shown in Figure 2.10, after an RTP channel is setup, a key exchange needs to take place to obtain master keys and salts to secure RTP and RTCP. Instead of using protocols like DTLS [19] and ZRTP [18], the app invokes the TA which has the master keys and salts in memory. Instead of returning the master keys and salts, the TA returns references (random strings with same length as key/salt and mapped to these data in the TA memory) to the normal world. For secure RTP / RTCP channel to be setup the app uses a key derivation function (KDF). This derives a session encryption key, session HMAC key and a session salt based on a master key and salt. Truz-Call uses the TA to generate the session keys and salts, by passing it the references for master keys and salts. Truz-Call uses the same approach to generate the keys in the TA as the normal world does in the non-secure case. The keys are generated using AES-CTR. The counter and plain text are fixed in the app for individual cases of key calculation; only variable involved is the master key and salt. The KDF passes the counters and plain texts to the TA. The TA returns references for session keys and salts to secure RTP. The TA returns the sessions keys & salts to secure RTCP in plain text, because RTCP is not handled in the TEE for the Truz-Call design as RTCP does not carry audio payload. It should be noted that the TEE invocation by KDF is only utilized once (at the beginning of call). It does not add any latency to user's conversation once the secure call is setup. Once the session keys and salts are setup, RTP and RTCP can be secured using SRTP and SRTCP.

So far, this section has covered the call setup flow on the caller's device. The flow on the callee device will be similar. When the SIP INVITE is received, before handling it, a TA is invoked and is passed the caller's SIP address and the CALL-ID. The control of the audio hardware will be taken over by the TEE. The TA looks up the secret phrase corresponding to the SIP address. The TA will calculate the master keys and salts. The KDF in normal world will invoke the TA in a similar manner to generate session keys and salts to secure RTP and RTCP.

4.8 TEE Invocation by Audio Framework

Android native framework consists of AudioRecord and AudioTrack, which contain the functions obtainBuffer() and releaseBuffer(). All Audio I/O utilizes these functions. Truz-Call invokes TEE in these native framework functions. This section discusses how these invocations work. In Android's implementation (AOSP), these native functions interact with the Audio Flinger, which provides the app process a buffer to either read data from or write data to. In Truz-Call, the native functions interact with the TAs to either get audio reference from or send audio reference to the TEE. The native framework allows reading and writing audio in different modes [42, 43], including a callback mode using which the audio data is fetched from or provided to a callback function. Linphone's native mediastreamer library [77] uses the callback mechanism for audio I/O. The native framework runs native threads

(AudioRecordThread and AudioTrack Thread) which use obtainBuffer(), the callback and releaseBuffer() in a while loop (Figure 4.9). The threadloop() function containing this while loop is executed periodically based native Thread class [28, 50].

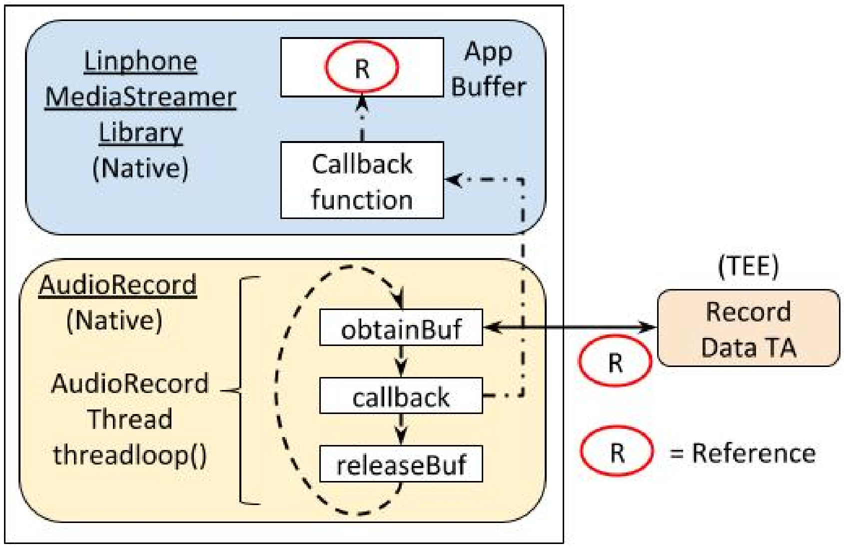

Fig. 4.9.: Use of TEE in Native AudioRecord

4.8.1 TEE Invocation by AudioRecord

VoIP apps using RTP buffer audio data before sending it out (packetization [98]). In case of Linphone, 640 bytes is buffered. In AOSP's implementation, to construct 640 bytes of audio data, at the call initiation the app instructs the audio framework that it should be notified each time 640 bytes of audio data is available. As the call progresses, the AudioRecordThread attempts to get the requested amount of audio from the AudioFlinger via obtainBuffer(). If enough audio data is not available, the framework notifies the app with the available amount via the callback and makes up for the remainder by continuing the loop. `Truz-Call` emulates this behavior as the `AudioRecordThread` uses `obtainBuffer()` to allocate a buffer and ask the `Record Data TA` for a reference based on the size requested by the app. If the requested amount of audio data is not available, the `Record Data TA` returns a reference of the same length as the available amount. The `AudioRecordThread` sends the reference to the `mediastreamer` library via a callback. The `releaseBuffer()` call frees the buffer. The `AudioRecordThread` makes up for the remainder by continuing the loop.

4.8.2 TEE Invocation by AudioTrack

VoIP apps using `RTP` use a jitter buffer. The `RTP` library [88] uses this buffer to hold packets as they arrive because of the possible variable delay involved. This allows the packets to be played in sequence. When the call is in progress, the amount of audio played by the app varies based on how much data the app wants to make available. When using Android's AOSP implementation, at call initiation the app instructs the native audio framework to request a certain number of bytes from the app during the call. The `AudioTrackThread` is constrained by the amount of audio data the `AudioFlinger` can take based on the `obtainBuffer()` call. The `AudioTrackThread` requests the app based on the buffer size available from `AudioFlinger`. The app responds with a size equal to the minimum of size asked and size available. The `AudioTrackThread` sends the audio data to `AudioFlinger` using `releaseBuffer()`. The `AudioTrackThread` handles the remainder by continuing the loop. `Truz-Call` 's design emulates this behavior. Initially AudioTrackThread requests the app based on the configured size via the callback. The callback gets the audio reference from mediastreamer. The reference received from the app is sent to the Playback Data TA in releaseBuffer(). The TA responds with the available size in TEE. If there is a remainder from the configured size (set at call initiation), then the loop is continued, and the AudioTrackThread requests a size from the app based on the buffer size available in the TEE.

4.9 TEE Invocation by SRTP

This section discusses how SRTP leverages the TEE for encryption and HMAC. The SRTP library does replay detection [10], which is not moved into the TEE in Truz-Call. The SRTP library in Linphone uses AES-CTR for encryption using 128 bit keys and uses SHA-128 when calculating HMAC. For AES-CTR, the SRTP library calculates the counter from four values: packet index, SSRC, salt and a block counter [29]. Packet index is a combination of the sequence number and a rollover counter (counts sequence number rollover of 65535). Packet index is distinct for each packet. The salt is calculated at the beginning of the call and is kept in the TEE. SSRC is an identifier for a source of RTP packets involved in a VoIP call and is given to TEE at the beginning of the call. The block counter increments from zero for each packet. As mentioned in Section 4.8, the native audio framework provides audio references to the app based on the size of available audio. This results in the RTP packet eventually constructed in the app consisting of a set of references in the payload. For each RTP packet, the SRTP layer sends the entire packet and session encryption & HMAC key references to the Record Crypto TA. The TA calculates the counter for AES-CTR using the sequence number in the RTP header. For the first packet the TA compares the sequence number against the initial sequence number to ensure that the normal world is using the sequence number specified by the TEE. For subsequent packets the sequence number is expected to increment by one each time and the TA verifies this (in case of rollover the TA verifies that the packet index is increasing). The TA encrypts the audio data corresponding to the set of references in the RTP payload (further discussed in Section 4.10). Once the encrypted payload is in place in the packet, the TA computes the HMAC and returns the result to the normal world. The SRTP library can then continue with sending the packet out. On the receiver device, the reverse steps happen. The Playback Crypto TA is given the received packet. The TA verifies the HMAC. If the verification fails, the TA informs the normal world. Otherwise, the TA calculates the counter from the sequence number and SSRC in the packet, the salt (from call setup) and the block counter. The TA decrypts the payload, replaces it with a reference and returns the result to the normal world. The SRTP layer forwards the packet containing the reference to the RTP handling layer to continue playback.

4.10 Reference Data Management

This section explains how Truz-Call manages the plain text audio data in the TEE memory, and how it translates references to audio data or generates references for audio data. To manage audio data in the TEE, ring buffers are utilized similar to the normal world. Android follows the standard practice of using FIFO buffers to manage audio data. This is done in the `AudioFlinger` [41] and in Linux's ALSA driver [35]. `Truz-Call` uses two ring buffers inside `TEE`'s `SDP` memory, one for record data and other for playback data.

4.10.1 Data Management for Record

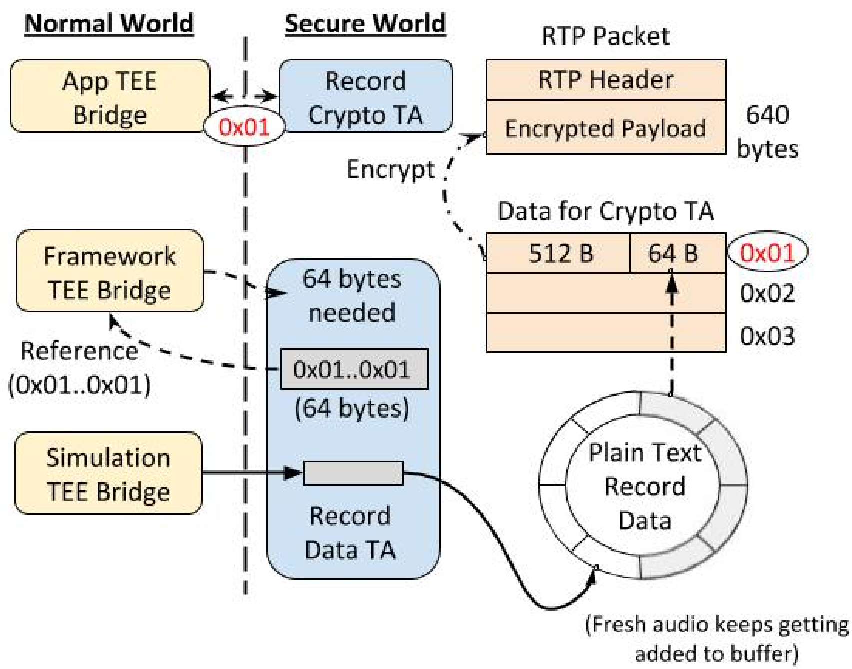

Fig. 4.10.: Reference Data Management for Record

`RTP` in normal world uses packetization. The VoIP app buffers a certain number of bytes before constructing an `RTP` packet. The native audio framework may send multiple requests to the `TEE` to provide the required number of bytes to the app. `Truz-Call` matches VoIP packetization behavior in the `TEE`. As shown in Figure 4.10, each time the native audio framework requests a certain number of bytes, the `Record Data TA` moves the requested (or available) number of bytes from the ring buffer to a separate cache in the SDP memory. The data in the ring buffer is provided by the `Simulation TEE Bridge` which gets it from the simulation hardware setup. The cache is necessary because by the time the `SRTP` layer invokes `TEE`, the data corresponding to the reference(s) may have been overwritten in the ring buffer (the overwriting behavior is similar to how audio drivers in Linux buffer data [9]). The `TEE` needs to give the audio framework a reference corresponding to the audio data moved into the cache. As discussed in Section 4.9, when the `SRTP` library invokes the `Record Crypto TA`, it needs to encrypt the `RTP` payload, for which it needs a buffer containing all the audio data corresponding to the set of references.

One of the design constraints of TruzCall is to reduce latency. A simple implementation would be to lookup the audio data corresponding to each reference, assemble the buffer and then proceed to encryption and `HMAC`. This would add latency because of the time spent in the `TEE` to assemble the buffer before actually starting the encryption (data corresponding to each reference would require two `memcpy()` operations). In order to reduce latency an approach is needed that uses less time in the `TEE` to prepare the buffer to be encrypted. When the `SRTP` library invokes the `TEE`, the buffer corresponding to the `RTP` payload should already be setup ready to be used. To achieve this, the cache in the `SDP` memory is organized holding plain text audio in multiples of packetization buffer size (configurable at call initiation). Whenever the native audio framework asks the `TEE` for audio data, before returning a reference the corresponding (or available) bytes of audio are copied into the cache. The cache is always preparing the next buffer for `RTP`. Since the reference to be returned by `TEE` is supposed to be the same length as requested (or available) number of bytes, the `TA` returns a string which is generated by using memset() and repeating the index in the cache (e.g. in Figure 4.10, string returned is 0x01..0x01). This string is the reference for the normal world. When the SRTP library invokes TEE, the first byte in the RTP payload is the index in the cache for the next buffer to be encrypted (reduction in data sent to the TEE from 640 bytes to 1 byte reduces latency). This approach results in one memcpy() needed for data per reference. The difference between two vs one memcpy() may appear insignificant, but it should be noted that TEE invocation happens several times per second during a call, and all that latency adds up to affect voice quality.

4.10.2 Data Management for Playback

RTP in the normal world does jitter handling using a jitter buffer [76]. Out-of-order delivery and/or delay variation in RTP causes jitter [23]. As RTP data is decrypted in the TEE, a cache is required to hold decrypted data until the app plays it via audio API. Truz-Call matches VoIP jitter buffer behavior in the TEE. It allows playing of received audio after being reordered by the normal world (in case of out-of-order RTP packets). Individual tracking is done for amount of played audio in the TEE for each decrypted RTP payload.

Similar to how a cache is maintained to prepare RTP payload for encryption, a separate cache is used in the SDP memory to keep the playback RTP payload decrypted in the TEE. As shown in Figure 4.11, when the SRTP library receives a packet from the network, it forwards it to the Playback Crypto TA for HMAC verification and decryption. Once decrypted the buffer is added to the next index in the cache. The

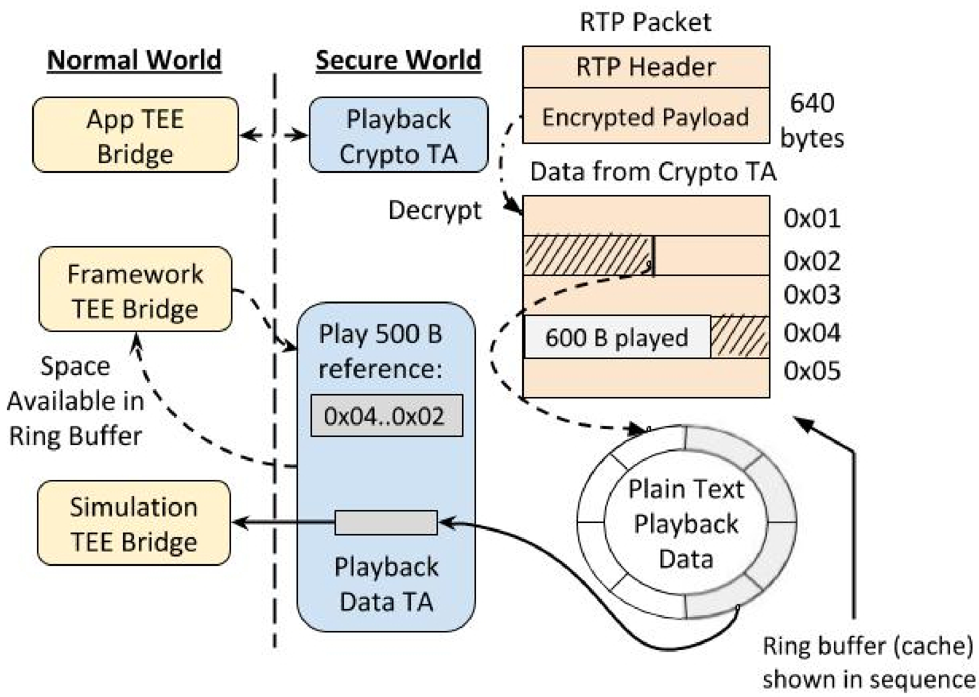

Fig. 4.11.: Reference Data Management for Playback

reference returned to the `SRTP` library is of the same length as the `RTP` payload, and is assigned the cache index value (using `memset()`). When the native audio framework requests playback data from the app, the size can vary (discussed in Section 4.8). As the `Playback Data TA` gets requests to play audio, it copies data from the cache index into the playback ring buffer and keeps track of how much data has been played from the index. Cache index used to play audio is specified by the passed reference. Figure 4.11 shows a case when 5 `RTP` packets are received in the normal world, but they are out of order, with correct order requiring audio for packet 4 to be played first, followed by packet 2. The figure shows the state when 600 bytes have been played from packet 4, and the playback request spans audio data from two indexes `0x04` and `0x02` (the passed reference string had 0x04 40 times and 0x02 460 times). The data in the playback ring buffer is played out by the Simulation TEE Bridge.

A question that can be asked is why can't one just make the ring buffers large enough so that enough data is always available for record or enough space is available for playback ? TEE environments operate with limited amount of memory. In a production environment, severals TAs can be present in the TEE for various use cases, which can reduce the amount of memory available. In addition, the amount of audio data available in TEE at any time depends on the type of audio hardware and the type of interface used. Also, 640 bytes is used to organize the cache based on the packet size used by Linphone. A different VoIP app may ask more or less bytes per packet. The goal of Truz-Call 's design is to be generic such that it can help reduce latency in different scenarios for VoIP.

4.11 Security Analysis

This section discusses the security analysis of the Truz-Call design. It is assumed that side channel attacks, covert channel attacks, hardware related attacks and attacks related to VoIP network are out of scope. The analysis assumes that the TrustZone hardware platform is trusted and the secure boot process has initialized the integrity-verified OP-TEE OS. The goal of the malicious normal-world OS is to obtain the plain text audio for a VoIP call. The OS can attempt to do this at various phases of the VoIP call. In each phase, the described scenarios won't work because of the various properties of the design. The OS may try to obtain the secret phrase typed by the user. During the secret phrase entry, TEE controls the UI and input, and user is informed of this using a secure LED. This has been discussed in existing work [162, 163, 181, 182]. The OS may try to fool the user that the secure call is initiated, but not give control to the TEE and mimic the secure UI for call initiation as shown by the TEE. The OS will not be able to access the secure LED, which is used to inform the user whether the audio peripherals are indeed in control of the TEE. Due to this, the OS cannot fool the user regarding secure call initiation. The OS may try to obtain the master key. The OS won't know the master key calculated during call initiation as the secret phrase used for its calculation is protected and the TEE gives the normal world only a reference to the master key. The encryption and decryption for SRTP in the TEE uses AES-CTR, which is a stream cipher and can be subjected to various attacks [173], including keystream reuse, bit-flipping and chosen-IV attacks. The normal-world OS can influence the counter because the sequence number is sent by the normal world. If the same key and counter are used, the XOR of cipher text can give XOR of plain text. In Truz-Call, the counter is not allowed to be repeated. As mentioned in Section 4.9, the counter calculated in TEE is derived from packet index, which is derived from sequence number and rollover counter. The TA verifies that the packet index is increasing each time. Bit-flipping requires knowledge of part of the plain text. The normal-world OS does not have access to the plain text audio. Chosen-IV attack relies on choosing certain IVs and analyzing the generated keystreams. The normal-world OS cannot observe the keystream as it resides in TEE memory.

As mentioned in Section 4.9, the SRTP library does replay detection. It does this based on packet index and uses a replay list & window to detect replay attacks. This functionality is not moved into the TEE. The normal-world OS may attempt to replay received packets. This is countered as the TA checks to ensure that the packet index handled is always increasing. The normal-world OS can attempt to replay voice payload for outgoing packets by holding onto references seen before. The size of the audio cache in the SDP memory provides a brief time gap before same index is used again due to index roll over. The TA zeros out the memory once the data at a certain index has been used. Reuse of an older index won't result in re-sending of data.

4.12 Simulation Test Environment

This section discusses the simulation based approach used for building the hardware environment for testing Truz-Call. This is the first time a simulation based approach has been applied to the area of TEE research. Similar approach is used in other areas like embedded system testing where it is referred to as hardware-in-the-loop simulation [12]. In TEE research one often needs to interface hardware peripherals with the TEE OS. This task can be challenging for non-hardware experts, depending on the available support from the TEE OS vendor. In the Truz-Call prototype, the Hikey 620 development board [72] is used. The OP-TEE OS provides different driver support [86] for different boards, and for the Hikey it provides UART and SPI drivers. Common audio hardware [60] used in prototyping rely on I2S for which no driver is provided by OP-TEE. Given the lack of support from the vendor and the community, with limited resources it would not be efficient to develop a board specific driver stack to make I2S work on Hikey. The board has USB interface available, but using it with TEE would require introducing the USB stack in the TEE OS. UART could be used to get audio into TEE, but it would require audio compression techniques like DPCM [171] and

ADPCM [170] with sample rate limited by the UART bandwidth. SPI could be used for audio, but it presents its own challenges including data buffering, full bus utilization, unnecessary conversion/overhead, and fine-grained clock speed control [58]. To build a hardware test environment to demonstrate Truz-Call, an approach needs to be used that does not depend on the available support from the vendor, and can best retain the quality of data needed for the experiment. To meet this requirement, a simulation based testing environment is introduced, in which a real phone is used to provide the audio hardware. The audio data from the phone is streamed to the TA in the TEE OS via the Simulation TEE bridge. The bridge is considered part of the secure world.

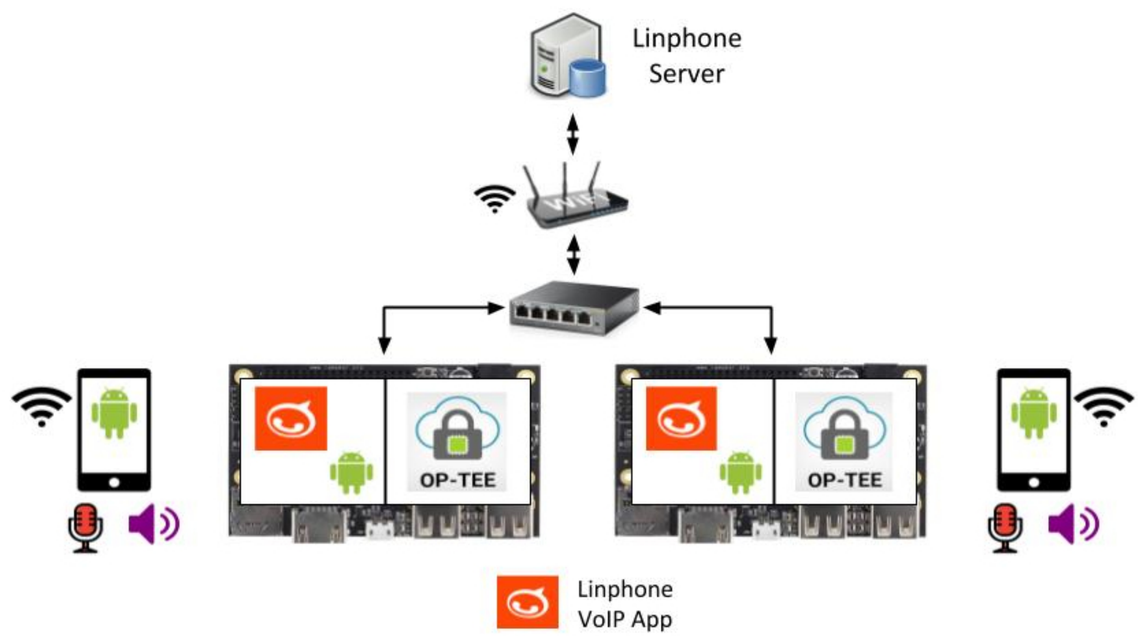

Fig. 4.12.: Simulation Setup

To setup the environment (Figure 4.12), a Nexus 5X phone is used with each of two Hikey 620 development boards (two ends of VoIP call during evaluation). Both Hikeys run Android OS version 7.1.2 in the normal world and OP-TEE OS version 2.5 in the

secure world. The Hikeys use USB ethernet adapters for internet access. Both Hikeys are connected to the same switch and can reach the internet via a connected router. The internet access is needed because the VoIP app needs to connect to its server for call initiation. The open source Linphone app [5] is used for testing (version v3.3.2).
Figures 4.10 and 4.11 showed how the `Simulation TEE bridge` provides data for record and gets data for playback. The bridge communicates with an Android app on the Nexus phone over TCP to send / receive audio data. The combination of the bridge and the external phone replaces the need for drivers inside the `TEE` OS for audio hardware access by the `TAs`. The simulation bridge does send/receive plain text audio between the external phone and the `TEE` Data `TAs`, but this component is used for easier prototyping. If a vendor adopted `Truz-Call`, the simulation bridge would no longer be needed as `TAs` would directly use audio drivers provided by the vendor in the `TEE`. In that case user's conversation plain text audio would never be returned to the normal world. The app on the Nexus phone records and plays audio in 16-bit PCM format (mono) at a sample rate of 16 KHz. The app continuously sends recorded audio to the bridge which makes it available to the ring buffer for record data in the `TEE`. The bridge periodically gets available audio in the `TEE` playback ring buffer and sends it to the app for playback on the phone. Although the simulation environment provides the benefit of making hardware setup easier for prototyping, it does add latency because of the time taken to send/receive audio data to/from the external phone. A video demo of the simulation setup can be found at [141].

4.13 Evaluation

This section discusses the evaluation done for `Truz-Call` using the Linphone app and the simulation test environment. From the point a call is established `Truz-Call` uses existing VoIP protocols. Any additional delay added is on the end device. The design doesn't change the delay on the network. The evaluation focuses on measuring modifications for secure VoIP on the end device. Network delay can vary as it does in everyday usage of VoIP. Since both Hikey boards act as sender and receiver during a VoIP call, metrics reported were collected on one of the devices. The reported metrics are based on three VoIP app configurations: (1) C-Off, (2) C-On and (3) Secure. In the first two cases, the VoIP app does not use `Truz-Call`, but the additional audio computation stages are turned off vs on respectively. In the third case, the VoIP app uses `Truz-Call` and the additional stages are turned off. Comparing the non-secure cases with `USB` audio (hardware attached to normal world) against secure case with simulation setup would be unfair because the simulation would add some latency. In all cases, the simulation environment was used for audio data. In the non-secure cases, audio data obtained by the `Simulation Bridge` is passed directly to the native audio framework.

For the test cases C-off and Secure, the additional audio computations in the Linphone app are disabled. In case of the computations resampling and compression, simply disabling them breaks the flow of the app because how it is engineered. So the code of these two stages was modified so that the reference audio data is not modified. Linphone downsamples 48KHz to 16KHz (reverse on receiving), so the configuration of the app was changed s.t. it directly asks for 16 KHz, in which case downsampling is not needed. In case of compression, the data is directly copied over to the target buffer instead of actually compressing. With these changes, the app needs 16-bit PCM audio data, and the simulation test environment is configured to read / write 16-bit audio.

4.13.1 Performance

This section compares the impact of Truz-Call on the time taken during a VoIP call. Truz-Call impacts the amount of time the app uses between getting audio data and sending out a packet (and vice versa for received audio). The time taken in the SRTP layer is reported as that involves the use of TEE in the secure case. Once a call is established, the time taken for a spoken word to be heard at the other end of the call will change when Truz-Call is used (end-to-end time). The time it takes the app to get audio data for record or send audio data for playback using our simulation setup is also reported. The evaluation focuses here on the time taken between native audio framework and the Simulation Bridge (the time taken by the daemon to send/receive audio data to/from the external phone over the network is excluded). The reported results are the average from 20 measurements. The overhead added in SRTP is 0.48 ms for outgoing packets and 0.54 ms for incoming packets. This has little impact on overall performance as Truz-Call adds a quarter second average overhead compared to C-off for end-to-end time during a call. The end-to-end time for C-on is higher because it uses additional computation stages in the VoIP pipeline, which are not used by the secure case.

Table 4.1: Truz-Call Performance Evaluation

	Non-Secure	Secure
SRTP Time per Outgoing packet (ms)	0.16	0.64
SRTP Time per Incoming packet (ms)	0.12	0.66
End-to-End Time (seconds)	C-off: 4.27 C-on: 5.6	4.51
Audio Input Time (ms / KB)	16.95	18.45
Audio Output Time (ms / KB)	14.31	32.96

4.13.2 VoIP Quality

VoIP call quality can be affected by several factors [23, 79, 98], including packet loss, voice quality, delay and delay variation (jitter). For VoIP, 1-2.5% of packet loss is considered acceptable [172]. The evaluation includes measurements for 2% packet loss in the test for voice quality. To test packet loss, the evaluation uses the Linux `iptables` tool. Mean opinion score (MOS) is a well-known measure of voice quality [80]. It is a subjective test wherein participants judge the quality of a voice transmission system by rating the voice quality on a scale of 1 to 5. The evaluation used Amazon Mechanical Turk [81] to gather the data from 60 participants (US-based). The audio recordings from calls using non-secure (C-on) and secure cases were provided. The recordings were audio data received on one of the Nexus phones in the simulation setup. The participants were also asked to answer a question based on each recording to check if they understand the content and to ensure survey quality. The survey and the recordings can be found at [83, 84, 92, 93, 97]. The MOS scores and percentage of participants that answered the questions correctly are reported. The MOS scores were expected to be low because of the additional latency from the simulation setup. MOS scores provide user perceived quality difference between the non-secure and secure cases. The participants were able to comprehend the contents of the secure call at least 81% of the time. This result would be better if an audio driver was available in the TEE, as simulation makes prototyping easier but adds latency during testing.

Table 4.2: Truz-Call VoIP Quality Evaluation

		C-on	Secure
MOS (no packet loss)		2.1	1.3
MOS (2% packet loss)		2.0	1.2
Correct Answer (no loss)		95%	95%
Correct Answer (2% loss)		98%	81%
	C-off	C-on	Secure
JBM (ms)	55	211	207
IAJ (average)	26.41	27.38	26.12
IAJ (median)	26.5	27.3	26.6
JB (ms)	67.5	89.06	79.26

There are several types of delay [11, 98] involved in VoIP. In Truz-Call 's evaluation, the relevant delays include processing delay and packetization delay. Processing delay relates to the audio codec algorithm which is used for compression. Since the additional audio computation stages were disabled in the secure case, the delay incurred for this stage was not measured. The packetization delay relates to the buffering of audio by the RTP library before sending out a packet. Truz-Call does not change the amount of audio buffered for each packet. The evaluation measures the time taken to prepare each RTP packet before it is handed off to the SRTP layer. The average time taken for each case was as follows: (1) C-On: 19.98 ms, (2) C-off: 18.08 ms, (3) Secure: 21.23 ms. During a VoIP call, RTP packets may arrive out of sequence and/or at varying intervals [23, 57, 73]. VoIP apps like Linphone use a jitter buffer [76] to hold incoming packets before the corresponding audio is played out, which adds some delay. Since

Truz-Call uses TEE at different layers of the VoIP stack, TEE invocations can add timing irregularity and contribute to jitter. Three metrics related to jitter are reported: (1) JBM: maximum jitter buffer delay obtained from RTCP XR [7], (2) IAJ: inter-arrival jitter obtained from RTCP SR [8], (3) JB: jitter buffer size. Metric (1) is the maximum delay applied to received packets by the jitter buffer. Metric (2) is mean deviation of the difference in packet spacing at the receiver compared to the sender for a pair of packets (the average and median are reported). For metric (3), the average value is reported. The values correspond to a 15 minute call. The secure case adds average 1.25 ms overhead in RTP packet construction, but adds less jitter compared to C-on, due to less number of stages in the VoIP pipeline. When compared to equal number of pipeline stages in C-off, secure case does add jitter overhead, but still results in a quarter second average end-to-end time overhead.

4.14 Publication

The Truz-Call design has been published in 2020 in the paper titled *TruzCall: Secure VoIP Calling on Android using ARM TrustZone* [150]. The dissertation author was the first author in this paper publication.

5. TRUZ-SIM: HARDWARE SIMULATION TO ASSIST TRUSTZONE RESEARCH

5.1 Problem Overview

In TEE research one often needs to interface different types of hardware peripherals with the TEE OS. This task can be challenging for non-hardware experts, depending on the available driver support from the TEE OS vendor. In this chapter, the TEE OS in focus will be OP-TEE, given its wide adoption [154, 157, 159, 160, 167, 184, 185] in research. The TEE board in focus will be Hikey (as discussed in Section 2.2).

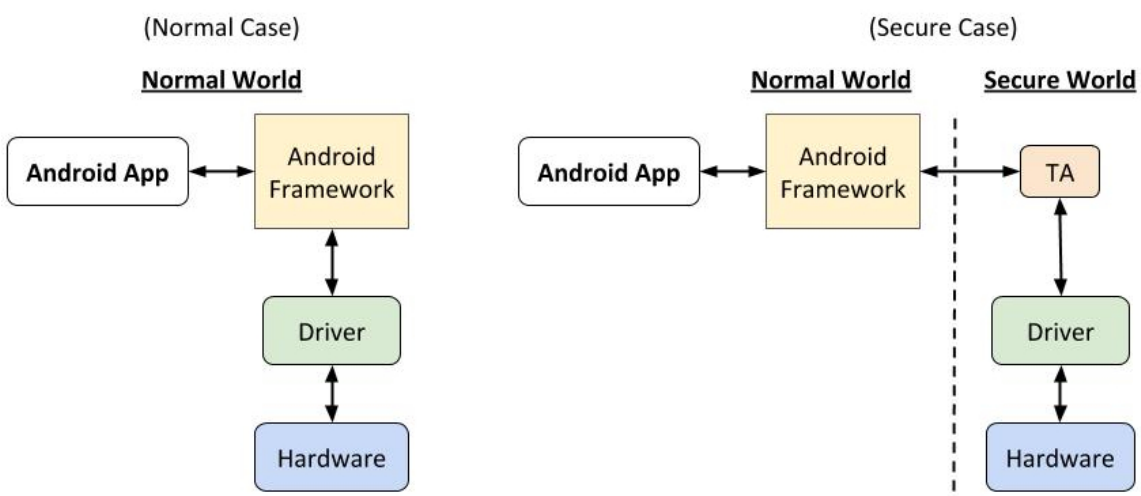

Fig. 5.1.: Access Hardware in Normal vs Secure Case

Referring Figure 5.1, in normal case an Android app requests hardware related data (e.g. GPS location) from the Android framework, which fetches it from hardware attached to the normal world via drivers in the Android kernel. To maintain transparency, in secure case an Android app will still use existing APIs, but will require data to be fetched from the TEE via the existing Android framework. The data obtained will vary based on the use case. It could be attested raw data obtained via the TEE. It can also be reference to raw data saved in the TEE memory. Currently there is no usable driver in the TEE kernel to allow a TA to read data from different types of hardware like GPS and camera.

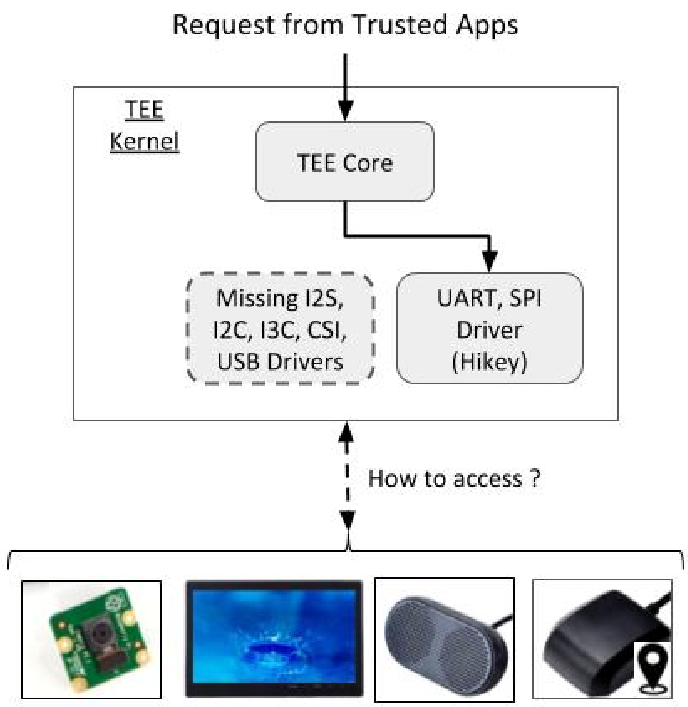

Fig. 5.2.: Existing Driver Support in TEE Kernel

The OP-TEE OS vendor provides UART and SPI drivers for Hikey. There are several issues a researcher can face when trying to interface hardware with the TEE OS.

1. Interfacing hardware with UART / SPI interface: If the researcher wants to use existing drivers (like UART and SPI), he/she must write a driver stack on top of those existing drivers. The researcher will have to spend considerable amount of

time to investigate the protocol of the specific hardware to be interfaced. Example for this can be the JPEG camera [74] (manual [75]).

2. Interfacing hardware with `I2C / CSI / I2S` interface: Since the `TEE` OS does not provide a driver for these interfaces, the researcher cannot attach the corresponding hardware and interact using the mentioned protocols. Example of such hardware can include CSI based camera [113], audio peripherals [60] and sensors (e.g. accelerometer [127]). For interfaces like `I2C`, one can use a bridge (example [146]) to connect with `UART`. In order to use a bridge hardware setup, some hardware experience would be needed and could be challenging for non-hardware experts.

3. Limited pins for multiple attachments: The Hikey board provides two `UART` ports and one `SPI` port on its low speed expansion header. One `UART` port is usually used for console. In the case where the researcher wants to attach multiple devices to `UART`, techniques like multiplexing [13] will need to be used, which would require hardware experience.

The researcher cannot write a custom driver stack for each hardware vendor, and cannot write interface specific driver layer into the `TEE`. Due to the previously listed challenges, there is a need for a `TEE` prototyping environment that can allow researchers to interface different category of hardware from different vendors with the `TEE` OS irrespective of the available driver support. The design should provide trusted applications with quality of data that matches a real phone. The design should also require reduced setup time and no hardware experience. The goal is to provide a design that can encourage independent researchers to prototype their ideas based on TEE.

5.2 Related Work

There are several works that provide hardware access across the same OS. Most of these works have been done on the Android OS, allowing apps on one Android device to access hardware on a second Android device. Rio [152] provides I/O sharing between mobile devices by splitting the I/O stack at the device file boundary. Semantics-Aware Design for Mounting Remote Sensors on Mobile Systems [158] builds a remote sensor I/O stack that is efficient in terms of communication energy and time costs. Interconnecting Heterogeneous Devices in the Personal Mobile Cloud [164] builds a resource sharing framework as a middleware in the mobile OS. Mobile Plus [166] allows Android applications to utilize system functionalities across devices by extending Android's binder inter-process communication (IPC) mechanism. Heterogeneous Multi-Mobile Computing [151] allows mobile apps to share and combine multiple devices by redirecting and transforming heterogeneous device input and output across mobile devices. It uses a data-centric approach by importing and exporting data to and from each mobile system using common cross-platform device data formats.

TrustUI [162] uses a split device driver architecture to allow a TA in the TEE to use hardware using normal-world drivers. The work is funded by a vendor and likely has vendor driver support in the TEE. For example, the TA in this work can operate on a framebuffer for display, which would require a framebuffer driver [21, 175]. Also the approach is not viable for peripherals like GPS sensor and camera as the normal world will be able to tamper with raw data being given to the secure world. There is no existing work that solves the problem for TEE to provide TAs transparent hardware access for different category of hardware, under the constraint that the researcher does not have the relevant drivers for hardware access in the TEE kernel.

5.3 Main Idea

Since there isn't sufficient driver support in the TEE OS, the design would need to leverage drivers outside the TEE to interface with the hardware. The idea involves creating a driver in the TEE OS that uses a *cross-OS binding* with a driver in a different OS to allow the trusted application in the TEE to transparently access hardware attached to the second OS.

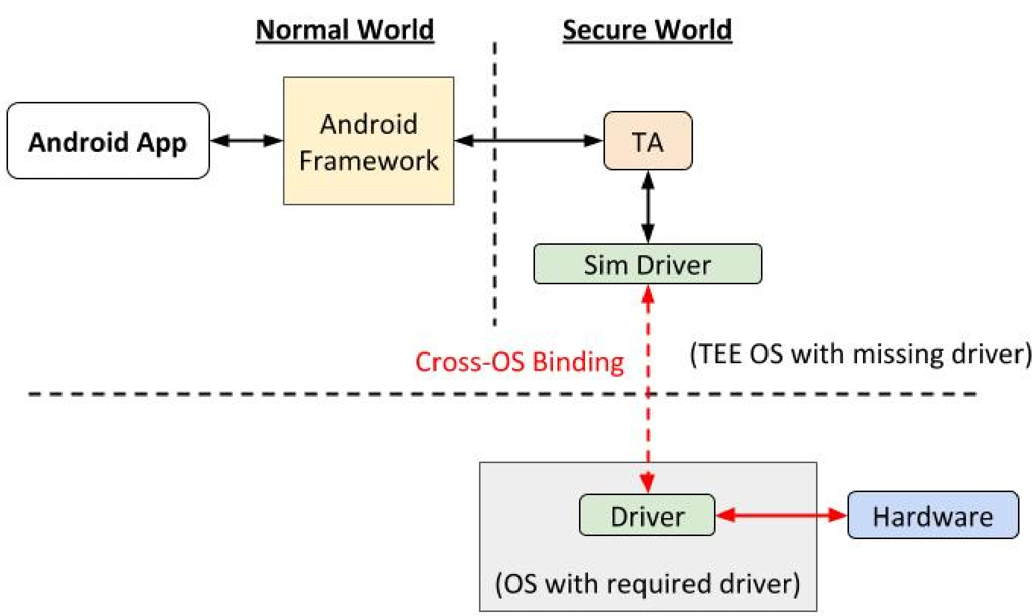

Fig. 5.3.: Cross-OS Binding for Hardware Access

To maintain transparency, in secure case an Android app will still use existing APIs of the Android framework. The TEE integration with the Android framework will invoke a TA inside the secure world. The TA needs to be fetch hardware data. Depending on the use case, TA could attest it to return to the normal world, or save it in TEE memory and return a reference to the normal world. The TA will utilize a thin driver layer added in the TEE kernel to access hardware. The driver leveraged by the TA inside the TEE will be referred to as *simulation driver*. A simulation in general is a system that exhibits the behavior of and performs functions of a real-world entity. In Truz-Sim, the design is trying to provide behavior / function of hardware attached to the TEE by leveraging hardware attached to the second OS and using corresponding drivers.

There are two aspects that need to be addressed: (1) How to interface the TEE with the second OS via cross-OS binding ? (2) How to transparently use devices from another OS ? Section 5.4 discusses interfacing between the TEE and the second OS. Transparency is discussed in Sections 5.4 and 5.5.1.

5.4 Design

As mentioned in Section 5.3, the design needs to utilize drivers outside the TEE to interface with hardware. In Truz-Sim, Raspberry Pi is used to provide the second OS, as it has rich community support to attach a variety of hardware. Figure 5.4 shows three options for cross-OS binding:

1. Bind with USB driver in the normal-world OS to interact with hardware attached to the normal world.

2. Bind with driver in Raspberry Pi OS via `UART` / `SPI` driver in the `TEE` OS.

3. Bind with driver in Raspberry Pi OS via network driver in normal-world OS.

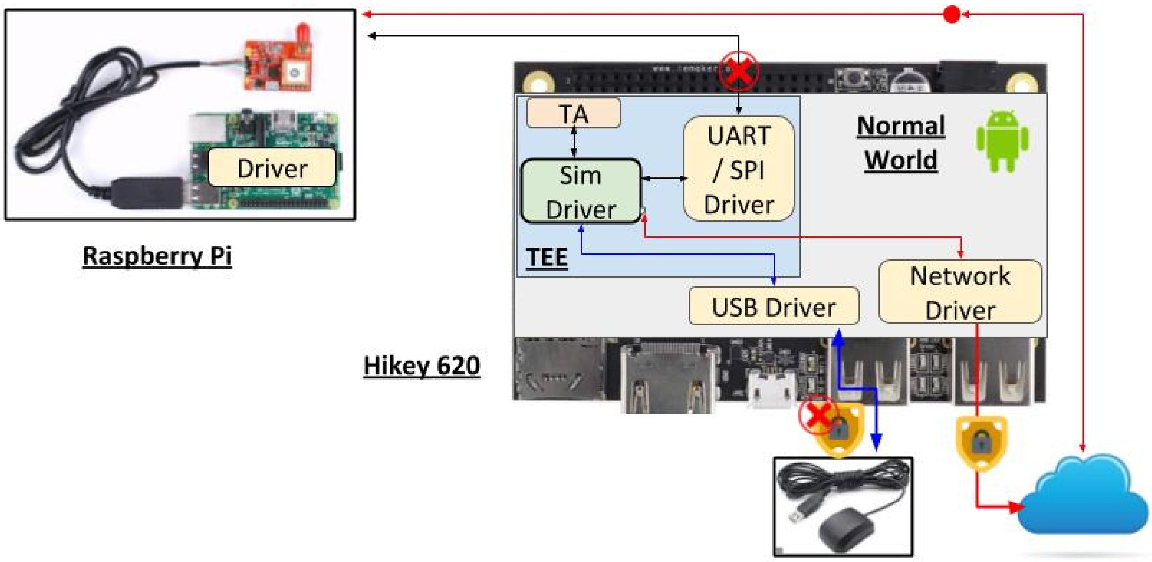

Fig. 5.4.: Cross-OS Binding Options

UART and SPI for Interfacing with Hikey. In Figure 5.4, the use of `UART` and `SPI` has been crossed out. The Hikey board has two `UART` ports on its low speed expansion connector. One `UART` is used for console. The other `UART` was observed as disabled in the Android + OP-TEE branch used for testing (this may change in the future). In order to use `SPI`, devices would need to communicate in a master/slave relationship [39], using the Pi board as a slave and the Hikey board as a master. There is no working demo to make `SPI` slave work on the Raspberry Pi in the community [33, 94, 95]. In addition to these, using `UART` / `SPI` also comes with the challenge discussed in Section 5.1.

Selecting Cross-OS Binding Option. Given the options shown in Figure 5.4, one or more options need to be selected for the design. In addition to the constraints mentioned in Section 5.1, there is also a security constraint which requires the cross-OS binding option to provide the same security guarantee as the real TEE driver path. Under the option selected, the normal-world OS should not be able to read data from the hardware. For hardware attached to the normal-world, the security constraint cannot be guaranteed. The option selected is to access hardware attached to Raspberry Pi via normal-world network driver, as the path can provide required security properties. For prototyping purposes, researchers can encrypt the data passed betweent the TEE and the Pi board.

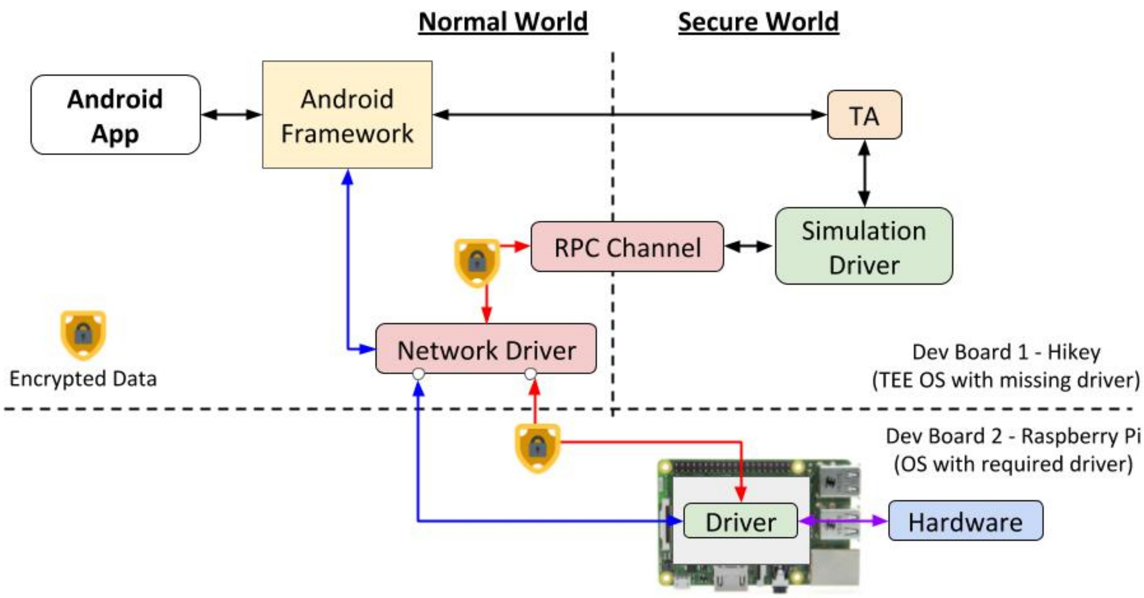

Fig. 5.5.: High Level Design

5.4.1 High Level Design

Based on selected cross-OS binding option, the high level design is shown in Figure 5.5. The TA will use a simulation API to invoke a thin simulation driver layer added in the TEE kernel to access hardware. The simulation driver will provide the TA a cross-OS binding with the required driver in the Raspberry Pi OS, allowing the TA to read data from hardware attached to the Raspberry Pi board. The data will be transparently returned to the Android app via `TEE` integration. The simulation driver will provide a cross-OS binding with drivers in the Raspberry Pi OS using the `RPC` channel via the normal-world network driver. In the existing OP-TEE OS, `RPC` is used in situations when a `TEE` thread needs to call some service from the normal world. In such case, the `TEE` saves the `TEE` execution state in its executing thread and invokes the normal world. When the normal world returns to the `TEE`, it resumes its thread execution. There are two main `RPC` services invoked by the `TEE`: (1) forwarding of a non-secure interrupt, and (2) invocation of a normal-world service (allocate shared memory, access normal-world filesystem, etc.).

The simulation design allows both normal world and secure world to get data from the Raspberry Pi OS. The normal-world Android framework can get data from the Raspberry Pi OS via the normal-world network driver and provide it transparently to the Android app. This data will be received in plain text. To simulate the real scenario, the researcher can hard code a symmetric key in the `TEE` and the Pi board to encrypt the data passed between the `TEE` and the Pi board to prevent the normal world from reading the data.

Hardware Support. `Truz-Sim` will be designed to allow testing of `TEE` based research ideas on mobile devices. A reference diagram of a modern mobile architecture can be found at [130]. There are various peripherals used with recent mobile devices including UI, sensors, audio, camera, bluetooth etc. In the project `Truz-Call`, an early version of simulation has been used to get audio data using external drivers. In the `Truz-Sim` project, the hardware covered includes GPS sensor, camera and UI.

Use Cases. There are several possible use cases for `Truz-Sim`. It can be used in scenarios where the `TA` needs to gets raw data from hardware, save it in `TEE` memory and return a reference to the normal world. This can be useful to address use cases as those discussed in chapters 3 and 4. It can also be useful in cases where `TA` needs to return attested raw data to the normal world. The use case of attestation will be used to explain the design for camera and GPS access in the next section. It should be noted that the attestation use cases as problems have been identified by a different PhD student [1]. `Truz-Sim` is providing the design to facilitate the testing of these use cases.

5.4.2 Camera Access Design

The camera access use case (Figure 5.6) involves both normal world and secure world needing data from the Pi board. The user takes a picture via a camera app in two steps. First the user needs to request a camera preview to allow the user to position the object in front of the camera. During this step, the camera needs to get live images from the camera hardware. The camera app requests preview via the camera library. In order to maintain transparency, the app will use existing APIs in secure case. The camera library would use the normal-world network driver to get the camera feed (series of camera pictures) from the driver in the Raspberry Pi OS. Once the object is in the right position, the user will click the camera button to take the picture. The camera app will request the picture via the camera library. The camera library will utilize `TEE` integration to send the request to the `TA` in the secure world. The `TA` will utilize the simulation API to request a camera picture. The simulation driver in the `TEE` kernel will use the cross-OS binding to forward the

[1] Ammar Salman (assalman@syr.edu)

request to the driver in the Raspberry Pi OS to get the camera picture. In order to show the entire flow works, in `Truz-Sim` the `TA` returns the obtained camera picture data to the normal world where it is transparently returned to the app via `TEE` integration with the camera framework. A complete test by a researcher would involve the `TA` also returning an attestation for the camera picture to the normal-world app, which could eventually be sent to the server.

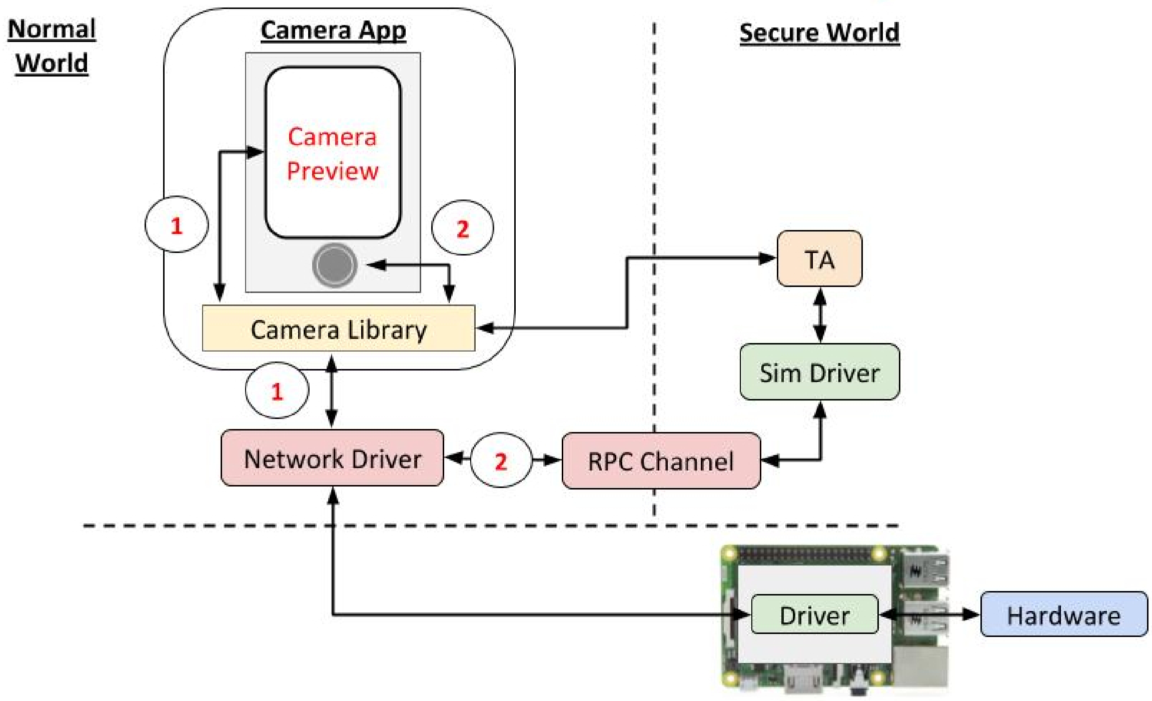

Fig. 5.6.: Camera Access Design

5.4.3 GPS Access Design

The GPS access use case (Figure 5.7) involves only the secure world needing data from the Pi board. An Android app gets a GPS location (latitude and longitude) via the Android framework's `LocationManagerService`. In the secure case, for transparency reasons the app will use the existing API. The location service will get the location via TEE integration by invoking a TA, which will use the simulation API to request a GPS location. The simulation driver in the TEE kernel will use the cross-OS binding to forward the request to the driver in the Raspberry Pi OS to get the next GPS sentence. If the received GPS sentence does not contain latitude / longitude information, then the TA will try again until a GPS sentence with latitude / longitude information is obtained. In order to show the entire flow works, in Truz-Sim testing the TA returns the GPS data to the normal world where it is transparently returned to the app via TEE integration with the Android location framework. A complete test by a researcher would involve the TA also returning an attestation for the GPS data to the normal-world app, which could eventually be sent to the server.

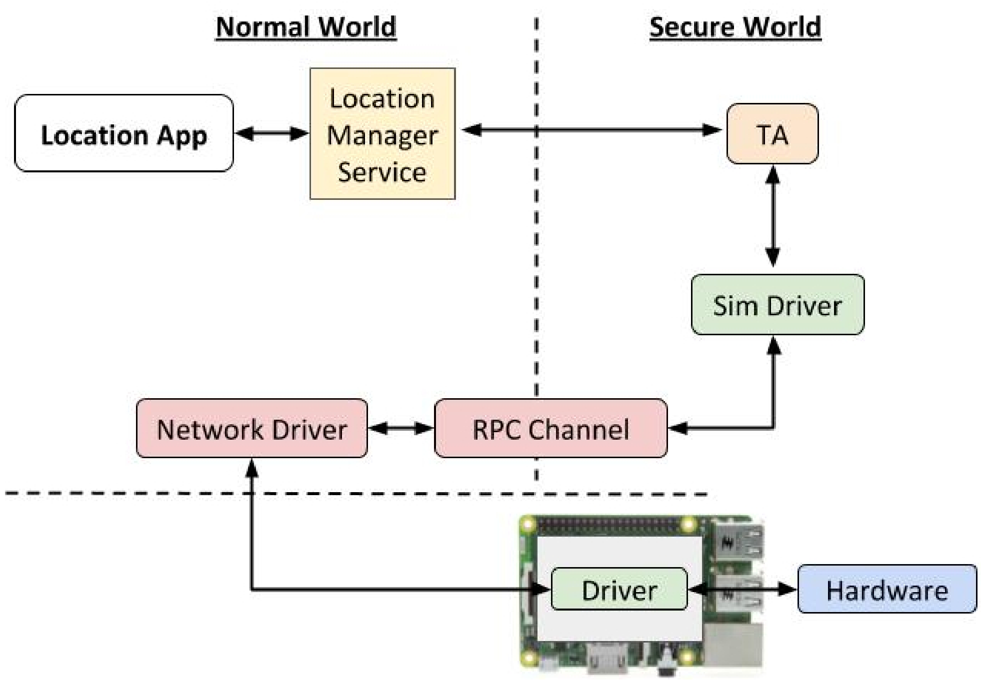

Fig. 5.7.: GPS Access Design

5.5 Implementation

This section provides implementation details for `TEE` library support for simulation, simulation driver, and details of camera and GPS access design.

5.5.1 Trusted App APIs for Hardware Access

The `TA` accesses hardware via APIs provided by a library in the `TEE`. In a real world scenario, the `TA` will use APIs specified by Global Platform [119] (abbreviated as GP). The `TA` API categories set by GP include Peripheral API and Event API [59]. In order to be transparent in case of simulation, it is important that the `TA` uses either the same API or an API with compatible behavior as GP APIs. OP-TEE does not provide GP APIs in the `TEE` library. In order to demonstrate that `Truz-Sim` can be compliant with GP, the design customizes the existing `TEE` library (`libutee`) to provide a simulation API that has compatible behavior with GP APIs. The GP APIs are not ported into OP-TEE.

It is also important that when a `TA` accesses hardware via the simulation API, it is not aware of where the data is coming from or the type of interface being used. The simulation is currently accessing data via the normal world using an external Pi board, but in the future researchers may extend it to use local interfaces inside the `TEE` (to access Pi or other external board) depending on the interface support at the time. When using simulation, the `TA` should only worry about the category of device being used (e.g. GPS, camera etc.).

GlobalPlatform Peripheral and Event APIs. Reading using peripheral API [59, Section 9.7.8] allows a TA to implement polled communication with a peripheral. The TA does not wait on any hardware signal and can use the API to retrieve the data available at the time of calling. The TA allocates a buffer of bufSize bytes before reading using peripheral API. On return, this will contain as much data as is available from the peripheral, up to the limit of bufSize. The bufSize parameter will be updated with the actual number of bytes placed into the buffer. The TA can use the peripheral API to write a buffer of certain size to the peripheral [59, Section 9.7.10].

The event API [59, Section 9.8] supports an event loop that enables a TA to process messages from peripherals. The event loop is useful in scenarios where peripheral interaction occurs asynchronously. This API is based on use of an event queue. A TA can call the event API to check if there are any events available. A TA can get multiple events at a time. The TA can specify the maximum number of events to be returned. The TA can also specify a timeout, so that a TA with multiple responsibilities can address them periodically without needing to use multi-threading. Events submitted to the event queue for a given peripheral are submitted in the order in which they occur. As Truz-Sim does not port the GP APIs into OP-TEE, this behavior is demonstrated by having a simulation API that can allow a TA to interact with peripherals using both polling and event queues. The hardware scope of Truz-Sim testing includes GPS, camera and UI (touch input). Peripheral API behavior is demonstrated in case of the GPS sensor, and the event API behavior is demonstrated in case of Camera and UI (touch input).

5.5.2 Simulation Driver

In order for the `TA` to use the simulation driver in `TEE` to interact with drivers on the Pi board (to leverage hardware attached to Pi), a *cross-OS binding* is needed. This binding is shown in Figure 5.8. The `TA` invokes the simulation driver via the modified userspace library `libutee` and a new system call added to the OP-TEE OS kernel. The simulation driver sends the request via the normal world using a `RPC` call [87]. This allows the request to reach the normal world daemon called *tee-supplicant* [136]. The daemon forwards the request using a `TCP` connection (via normal world network driver) to a Python program running on the Raspberry Pi board. The Python program can use available libraries to interact with hardware via drivers in the kernel.

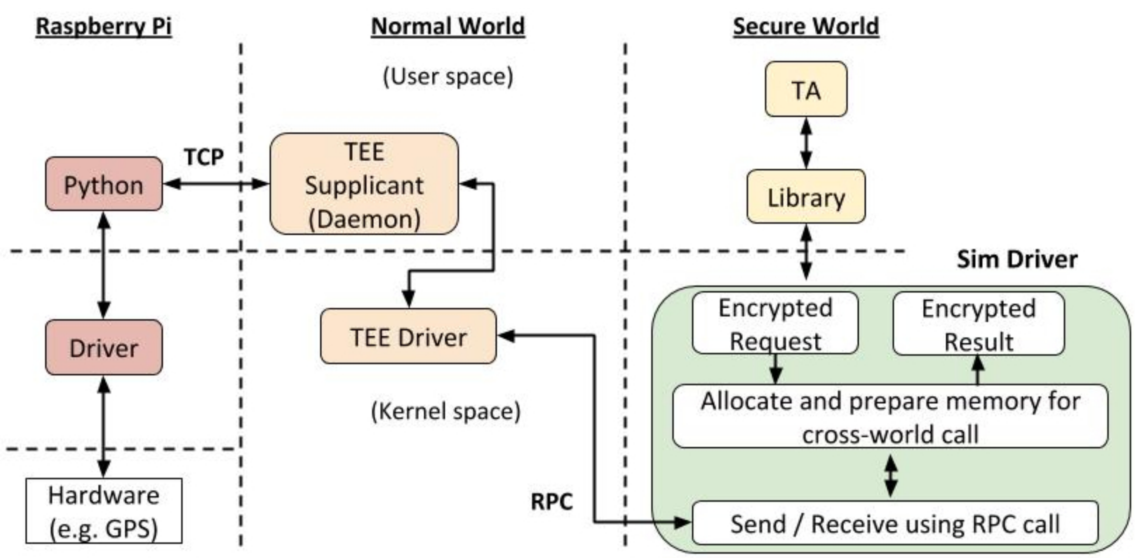

Fig. 5.8.: Use of RPC by Simulation Driver

Using RPC Call. A contiguous buffer will be utilized when sending arguments using the `RPC` channel across to the normal world. The simulation driver will receive serialized and encrypted simulation request payload from user space. The TEE library in user space will transparently serialize, encrypt and decrypt data for the TA. Since the RPC is a cross-world call, memory is required for data sent to normal world, and data expected from normal world. The simulation driver will allocate shared memory for input and output. In the OP-TEE kernel, a RPC can be invoked [135] by allocating input / output parameters using `thread_rpc_alloc_payload()`, preparing parameters of type `struct thread_param` and invoking the RPC using `thread_rpc_cmd()`. OP-TEE uses pre-defined commands to inform `tee-supplicant` in the normal world about the type of RPC request, for example `OPTEE_MSG_RPC_CMD_LOAD_TA` for loading a TA. For `Truz-Sim`, a new RPC command called `OPTEE_MSG_RPC_CMD_SIM` was added to `tee-supplicant` RPC handling in `process_one_request()` [137] to interact with the Raspberry Pi board using TCP.

5.5.3 Normal World App Testing

In order for the `Truz-Sim` design to be useful for testing TEE based ideas, the `Truz-Sim` project must evaluate normal-world TEE integration for various device types. As shown in Figure 5.9, in normal cases an Android app uses libraries provided by the Android to interact with devices via normal-world device drivers (black arrows in figure). It is important to maintain transparency for the Android app for testing secure cases, i.e. the app should be able to use the same APIs with minor configuration change to indicate use of TEE. To evaluate testing of secure cases, the existing Android library and Android framework are modified s.t. the app can use the same APIs to leverage TEE. In such cases,

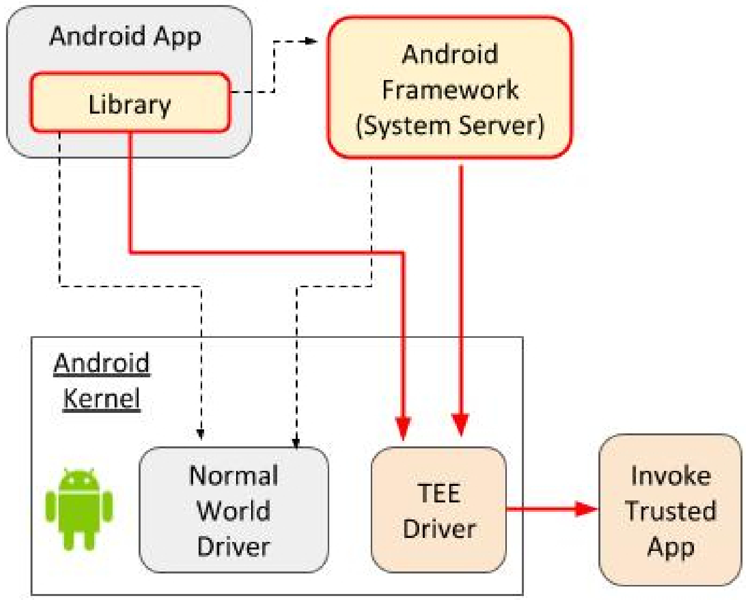

Fig. 5.9.: Normal World App Leveraging TEE for Secure Cases

the TEE driver in the normal world is used so that a TA is invoked and data for corresponding hardware is obtained via the TEE.

5.5.4 Camera Access Implementation

This section discusses how an Android app obtains a picture via the Android camera API and how the Truz-Sim design is used to transparently provide the Android app a picture via the TEE. An Android app can control the camera and get a picture using the architecture [112] shown in Figure 5.10. The app will use the camera API via a library provided by Android. The API allows the app to interact with a camera service in the mediaserver process. For Truz-Sim design evaluation for the camera use case, the camera library was modified such that the app can transparently use the existing camera API and request a picture via the TEE (e.g. attested) for the secure case.

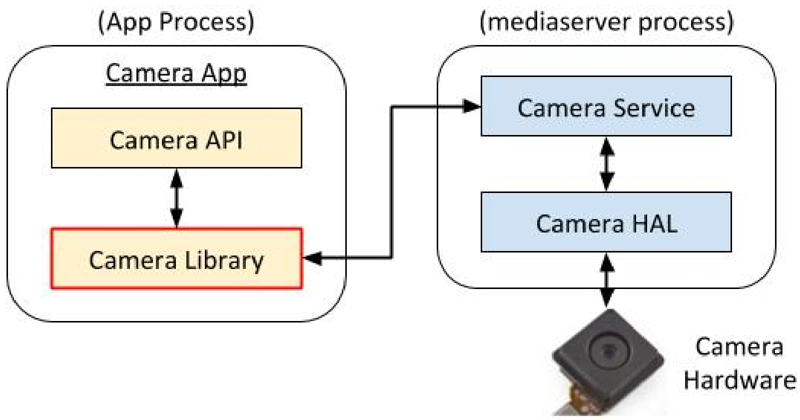

Fig. 5.10.: Default Control Flow for Getting Picture from Camera

Camera API Versions 1 and 2. Android provides two versions of the camera API (v1 [70, 110] and v2 [111]). v1 was deprecated in Android API 21. However when evaluating `Truz-Sim` for various apps, it was observed that many of the apps tested still used the v1 API. This may due to the fact that the v1 API is simpler and more consistent [69]. To evaluate the camera test case, integration for both camera v1 and v2 APIs was done to test the `Truz-Sim` design. Four major steps are involved when an Android app uses the Camera API: (1) the app accesses the camera to get a camera instance, (2) the app creates a camera preview, which involves using a view in the app's `Activity` to display what is observed by the camera as this allows the user to position the camera to take the picture, (3) capture is initiated to get a picture from the camera, (4) the picture is displayed or saved to a file. The comparison of the steps for v1 and v2 APIs is shown in table 5.1.

Camera Integration Changes. The `Truz-Sim` design needs to be tested for cases where the Android app needs to get a camera picture (e.g. attested) from the `TEE`. To

Table 5.1: Camera Access Steps in V1 and V2 APIs

Step	**V1 API**	**V2 API**
Access Camera	Using Camera API `open()`	Using CameraManager API `openCamera()`
Camera Preview	Using a class derived from `SurfaceView` and using Camera API `startPreview()`	Creating a `CameraCaptureSession` and using the API `setRepeatingRequest()`
Initiate Capture	Using API `takePicture()` and providing callback using Picture-Callback's `onPictureTaken()` to obtain picture data	Creating a `CaptureRequest`, with output `Surface` (e.g. `TextureView`) added using `addTarget()`; using `CameraCaptureSession` API `capture()`; callback `onCaptureCompleted()` invoked once picture is taken
Display Picture	Example: convert picture data to `Bitmap` and display in a view	The camera device sends a frame of the picture data into the output `Surface` included in the request

maintain transparency, the Android app needs to use the existing API to take a picture. Figure 5.11 shows the control flow for getting a picture via Truz-Sim. Before initiating the capture, the app will need a camera preview for the user. During this step, the app will get the pictures from the Pi board via the modified camera library (step ①). Once the camera has been positioned to take the picture, the user will click the button in the app. This will send a request to the TEE via a bridge (native daemon), causing the invocation of a TA (step ②). The TA will use simulation API to request a camera picture, which will result in a request sent to the external Pi board via RPC (step ③). The picture is returned to the TA, which returns it to the camera library. The library replies to the Android app via the corresponding callback based on the camera API version.

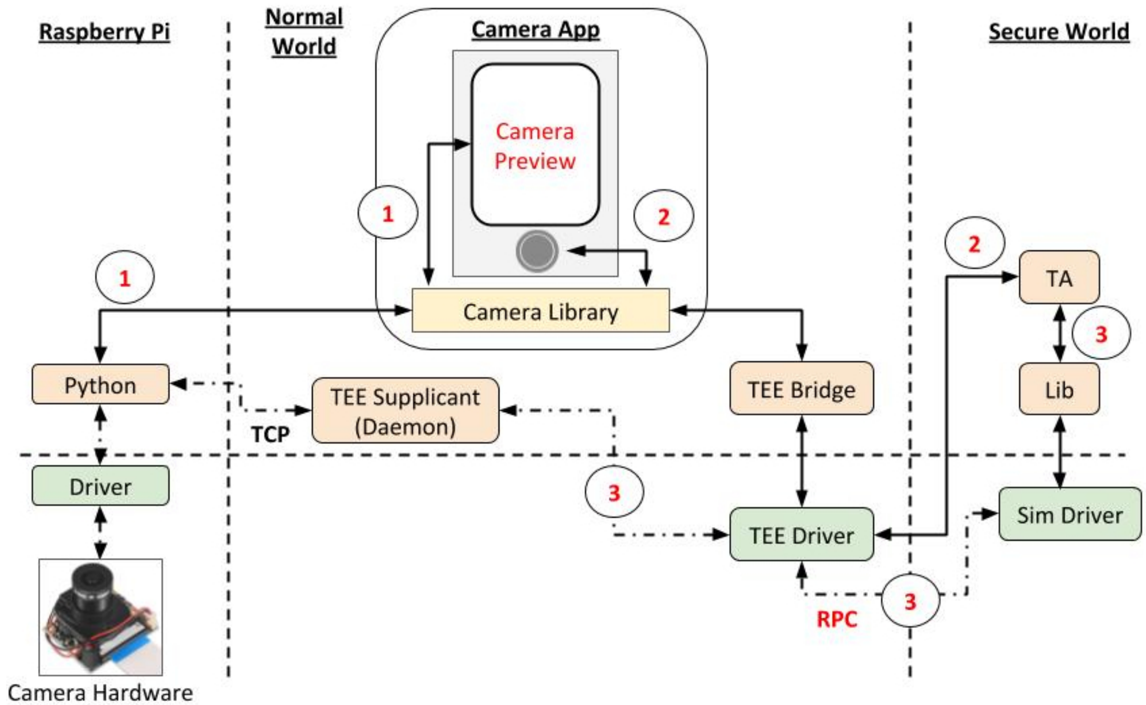

Fig. 5.11.: Truz-Sim Flow for Getting Picture from Camera

Camera Library Changes. Since the Android app will use the existing API for transparency, the `Truz-Sim` integration needs to ensure that the behavior matches the original case when getting the picture via the `TEE`. Figure 5.12 shows how the camera library was modified to evaluate the `Truz-Sim` design. To ensure that the simulation handling for camera behavior does not block the app's UI thread, the handling is done on separate threads via a new defined type `CameraThread` inside the camera library.

Using `Truz-Sim`, when an app uses the v1 API to open the camera, it will get a `Camera` instance and can proceed to use API `startPreview()` for the camera preview. When using the v2 API, the app invokes `openCamera()` to get a `CameraDevice`, and gets a special derived type called `SimulationCamera`. The app uses the `CameraDevice` reference to create a `CameraCaptureSession`, which

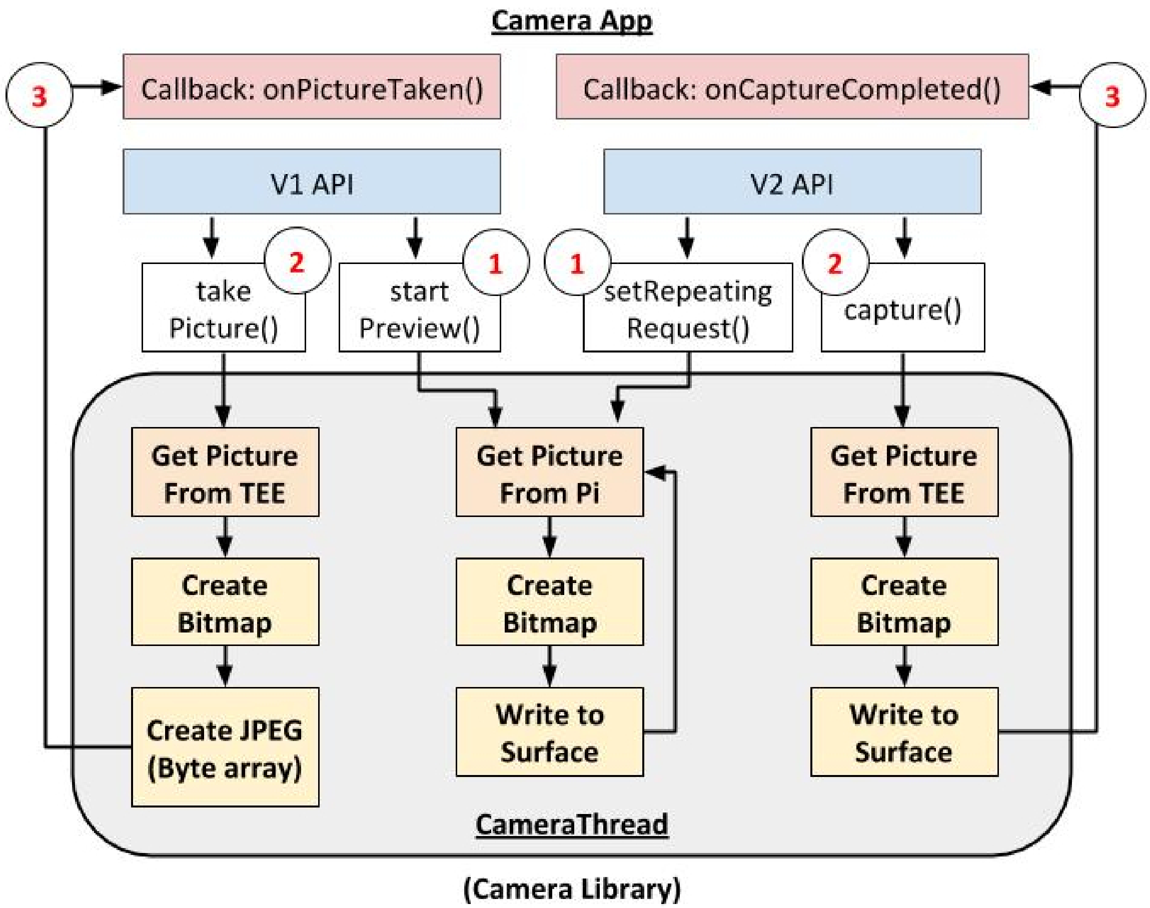

Fig. 5.12.: Truz-Sim Camera Library Modifications

creates a special type called `SimulationCameraSession`. The app can use the API `setRepeatingRequest()` to start the camera preview.

When an app wants to provide a camera preview (for both v1 and v2 APIs), the `CameraThread` follows a loop involving getting a picture from the Pi board, creating a `Bitmap` using the received data, and writing the bitmap to a `Surface` [107]. This involves using a `Canvas` [17, 99], with the steps involving use of the `Surface` APIs `lockCanvas()`, and `unlockCanvasAndPost()`, and the `Canvas` API `drawBitmap()`. When the app wants to take a picture using the v1 API via `takePicture()`, the `CameraThread` is used to get a picture from the `TEE` and converting the received data to a JPEG byte array to be returned via the callback onPictureTaken(). When the app uses the v2 API capture(), the picture retrieved from TEE is written to the output Surface.

TA API Usage for Camera. As mentioned in Section 5.5.1, in the Truz-Sim project the event API behavior is demonstrated for the use case of camera. The test case involves an Android app requesting one picture (e.g. attested) from the camera via the TEE. Unlike a GPS sensor which is always streaming data, the camera picture will not be immediately available at the time of request at the Pi board. The TA uses the simulation API and waits for next complete camera picture event by using parameters for maximum number of number of events as 1 and timeout as 5 seconds. Once the TA receives the camera picture, it can further attest the picture. In the evaluation, to demonstrate that the path works, the picture is simply returned to the camera library in the normal world, which returns it transparently to the Android application.

Accessing Camera Picture on Raspberry Pi. The Python program on the Pi board uses the PiCamera [37] library to access a camera image from a camera module [113]. The Truz-Sim evaluation focuses on taking the picture of a QR code, so the Python code uses a resolution of 224 X 208 when requesting the picture. PiCamera provides a 3D RGB array [3] via PiRGBArray. The python code returns the raw 3D array data to the TA in the TEE.

5.5.5 GPS Access Implementation

This section discusses how an Android app obtains GPS location via the Android framework and how the Truz-Sim design is used to transparently provide the Android app a GPS location via the TEE. An Android app gets GPS location from an Android service called Location ManagerService. As shown in Listing 5.1, the app creates a LocationListener with a callback called onLocationChanged(). The app requests the location service for GPS location using the API requestLocationUpdates() [129]. Once the location service has a GPS location (from a provider like GNSS), it invokes the onLocationChanged() callback providing a Location object, which can be used to obtain latitude and longitude information. Figure 5.13 shows the control flow for an app obtaining location.

Listing 5.1: Android App Getting GPS Location

```
public class MainActivity extends AppCompatActivity {
  private LocationManager locationManager;
  private LocationListener listener;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    locationManager = (LocationManager)
        getSystemService(LOCATION_SERVICE);
    listener = new LocationListener() {
     @Override
```

```
        public void onLocationChanged(Location location) {
            // location.getLatitude()
            // location.getLongitude()
        }
    };
    locationManager
     .requestLocationUpdates("gps", 5000, 0, listener);
}
```

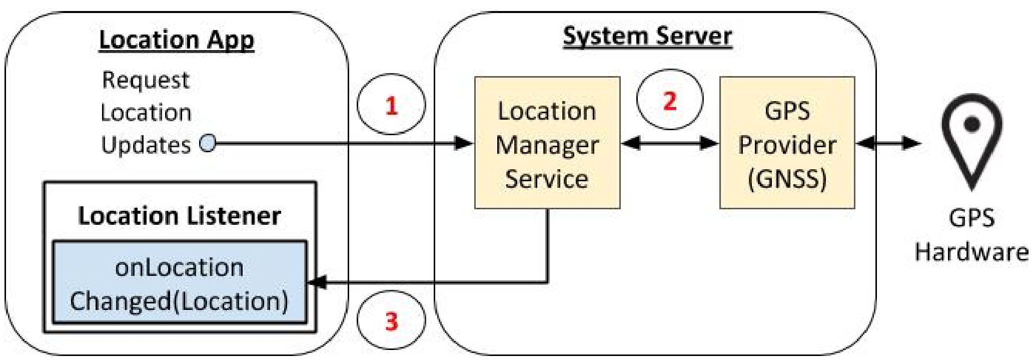

Fig. 5.13.: Default Control Flow for Getting GPS Location

Location Integration Changes. The `Truz-Sim` design needs to be tested for cases where the Android app needs to get GPS location (e.g. attested) from the `TEE`. To maintain transparency, the Android app needs to use the same API to get the location. Figure 5.14 shows the control flow for getting the GPS location via `Truz-Sim`. The app uses the API `requestLocation Updates()` to ask for GPS location from `LocationManagerService` (LMS) (step ①). The LMS forwards the request to the

TEE via a bridge (native daemon), leading to the invocation of a TA (step ②). This is blocking call done on a separate thread, so that LMS is not blocked. The TA will use the simulation API to request GPS location, which will result in a request sent to the external Pi board via RPC (step ③). The python code on the Pi board uses the pySerial library [140] to will retrive the location from GPS hardware [121]. The location is returned to the TA, which returns it to LMS. The location service replies to the Android app via the callback onLocationChanged() to provide the GPS location (step ④).

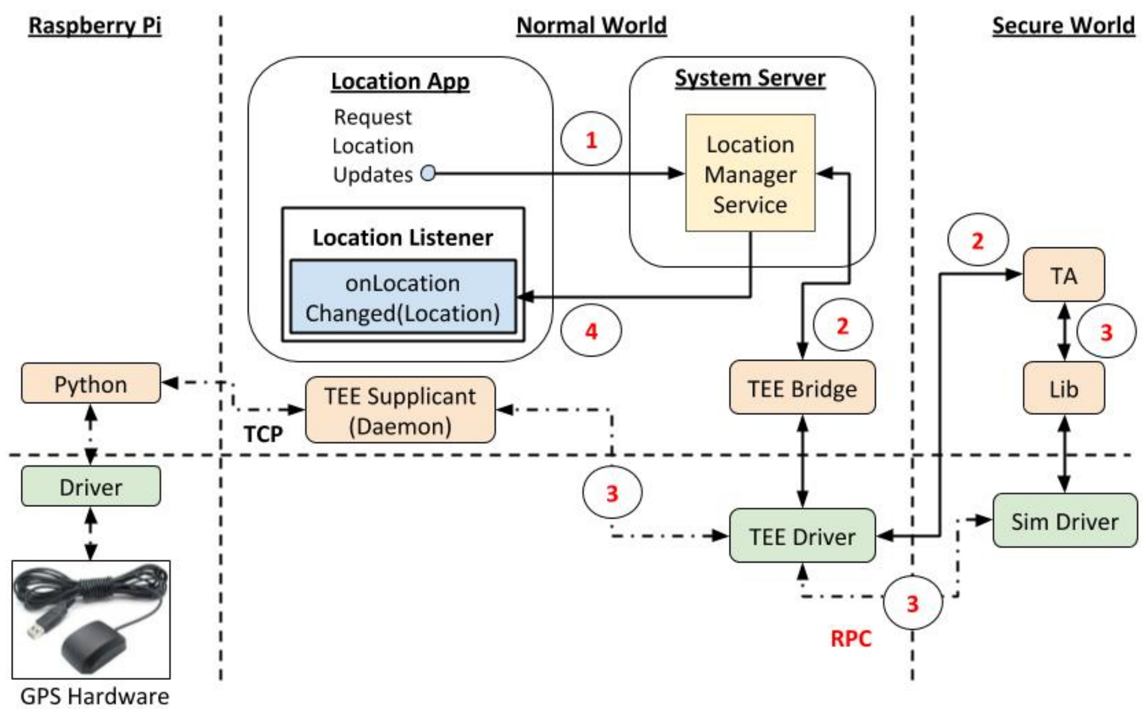

Fig. 5.14.: Truz-Sim Flow for Getting GPS Location

GPS NMEA Sentence. A GPS receiver module uses a protocol called NMEA, with each block of data received referred to as a NMEA sentence or a just "sentence". There are different types of GPS sentences [4, 20]. When GPS hardware [121] is connected to the Pi board, different types of GPS sentences are observed as shown in Figure 5.15. In the

Truz-Sim project, the focus is on three sentences, namely GPGLL, GPRMC and GPGGA. These sentences can provide latitude and longitude information. From the example of GPGLL sentence shown in the Figure 5.15, 4302.29963 (N) and 07607.84018 (W) are latitude and longitude respectively. During testing, values except latitude and longitude were hard coded in the LMS as several apps need more information that just latitude and longitude. Researchers can further expand the scope and analyze other sentences for more information.

```
$GPGLL,4302.29963,N,07607.84018,W,183416.00,A,D*76
$GPRMC,183417.00,A,4302.29953,N,07607.84030,W,0.305,,010919,,,D*61
$GPVTG,,T,,M,0.305,N,0.565,K,D*26
$GPGGA,183417.00,4302.29953,N,07607.84030,W,2,05,3.26,116.7,M,-34.4,M,,0000*60
$GPRMC,183419.00,A,4302.29930,N,07607.84067,W,0.159,,010919,,,D*63
$GPVTG,,T,,M,0.159,N,0.294,K,D*24
$GPGGA,183419.00,4302.29930,N,07607.84067,W,2,05,3.26,118.0,M,-34.4,M,,0000*60
$GPGSA,A,3,51,25,29,15,05,,,,,,,,5.59,3.26,4.54*00
$GPGSV,3,1,10,02,55,098,19,05,78,027,35,15,17,193,43,21,04,295,25*78
```

Fig. 5.15.: GPS Sentences Observed On Raspberry Pi

TA API Usage for GPS. As mentioned in Section 5.5.1, in the Truz-Sim project the peripheral API behavior is demonstrated for the use case of the GPS sensor. The TA will use the simulation API and specifies a maximum size for GPS sentence length. The python code on the Pi board retrieves the next GPS sentence seen. The TA requests the next GPS sentence (one at a time) until a GPGLL sentence is found. The TA gets the raw GPGLL data from the Pi board, and returns GPGLL sentence to the LMS. The LMS parses latitude and longitude from GPGLL and converts to decimal coordinates before returning to the app. In Truz-Sim evaluation, the GPS sentence is simply returned to the normal world, but in a real test case, the TA can also return an attestation of the GPS location to the normal world.

Record and Replay. Depending on the type of building the researcher is testing in, there may be issues observed when using GPS indoors [122]. It will depend on the building's construction material and potential interference sources. The researcher can choose to use a GPS signal amplifier. The researcher can also use a record and replay approach, wherein the researcher records a raw GPS trace when outside the building and save it to a file. When using the simulation setup, the researcher can use the saved file in the Python program to provide the next GPS sentence upon request from the TEE.

5.5.6 UI Touch Input

As discussed in Sections 2.3 and 2.4, Android apps can use `EditText` and `AlertDialog` to get text input and action confirmation respectively. Text input is used as a test case to evaluate whether `Truz-Sim`'s design can be used for UI touch input. Chapter 3 discusses how seamless keyboard binding can be used to allow an Android app to get secure text input via the TEE (shown in Figure 3.4). The interaction between the normal-world input method framework and the TEE will be assumed to be the same as Figure 3.4. Under the scope of `Truz-Sim`, the evaluation needs to establish that the TA can use the simulation based approach to reliably get the text input. The final flow for a researcher using `Truz-Sim` to test `Truz-UI` will look like Figure 5.16.

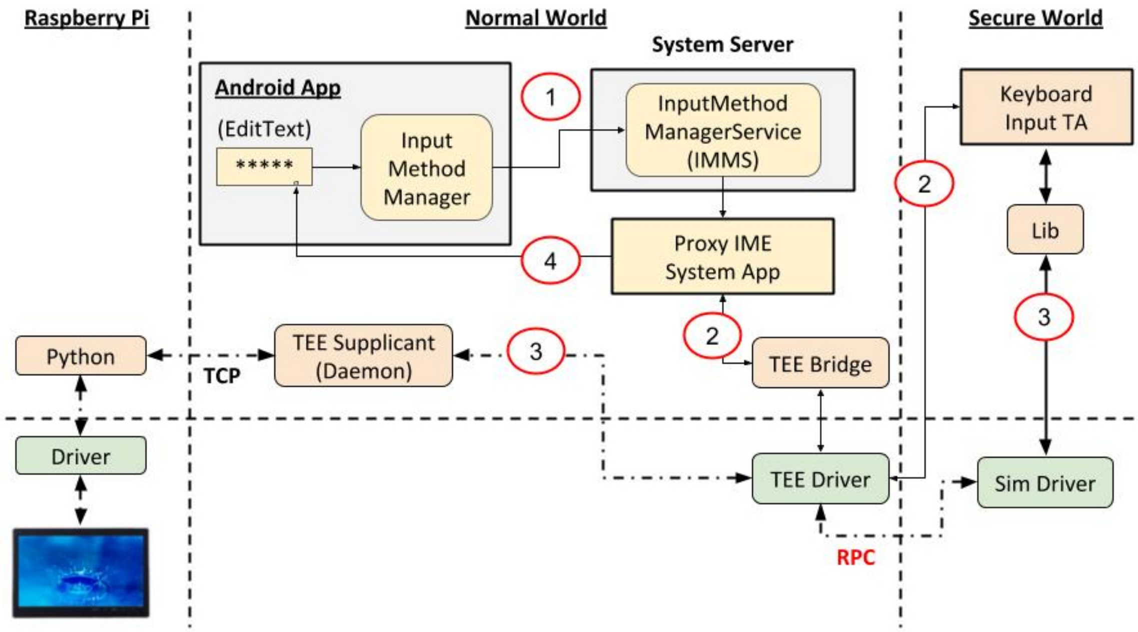

Fig. 5.16.: Flow for Truz-UI Test Using Truz-Sim

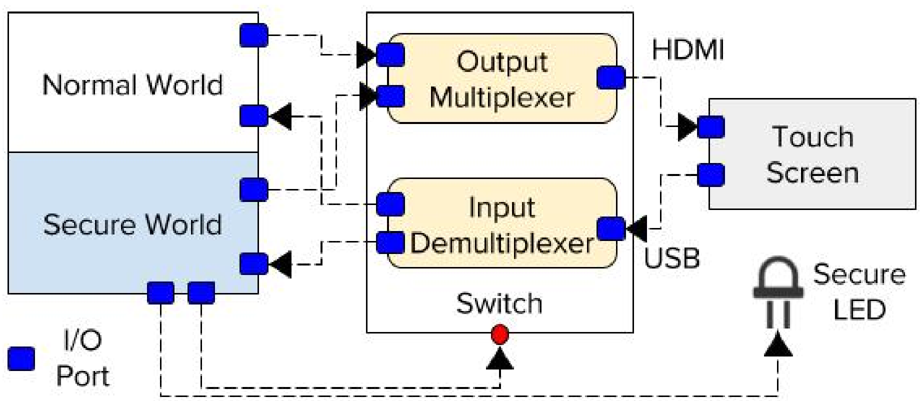

Fig. 5.17.: Hardware Setup Overview for Truz-UI (From Chapter 3)

Hardware Setup. Section 3.9 presents a overview picture of the hardware setup for testing the use case of secure text input (under `Truz-UI`). The picture is duplicated in Figure 5.17 for reference. The same picture can apply when a researcher is testing based on the `Truz-Sim`'s design. The difference with Section 3.9 is the use of the `RPC` channel instead of the `UART` channel from the `TEE`. It was the intention of the disseration author to use the most recent AOSP + OP-TEE build available at the time to evaluate Truz-Sim and UART was not an option.

When the TA is invoked by the normal-world method framework, it can use the RPC channel (similar to UART) to inform the Pi board of the request, which is also lead the Python code to change the switch setting for the mulitplexer and demultiplexer. The Pi board will have control of display and touch input. Figure 5.16 shows a snapshot of this state where normal-world is not controlling the display or touch input.

In previous sections (5.5.5 and 5.5.4) and corresponding evaluation sections (5.6.1 and 5.6.2), the use of switches was not discussed. A researcher can use a hardware setup similar to Figure 5.17 for the use cases of camera and GPS to further extend the setup shown in Figure 5.22. The researcher can use a USB based camera / GPS, and use a USB switch (under Pi's control) to decide which world can control the peripheral. When the peripheral is in normal world's control, its workability will depend on whether the AOSP build at the time has necessary support for the USB peripheral.

TA API Usage for Touch Input. As mentioned in Section 5.5.1, in addition to camera, the event API behavior is also demonstrated for the use case of UI (touch input). Unlike the evaluation for camera, where only one event is read at a time (picture event), the UI touch input evaluation covers reading multiple events at a time. Taking the example of a TA needing password input from the user, the TA will use the simulation API to send a RPC request to the Python code on the Pi board to request text input. During the test, the TA specified maximum number of events as 5 and timeout as 3 seconds. The Python program on the Pi board uses the Tkinter library [142] to draw the UI, similar to the one

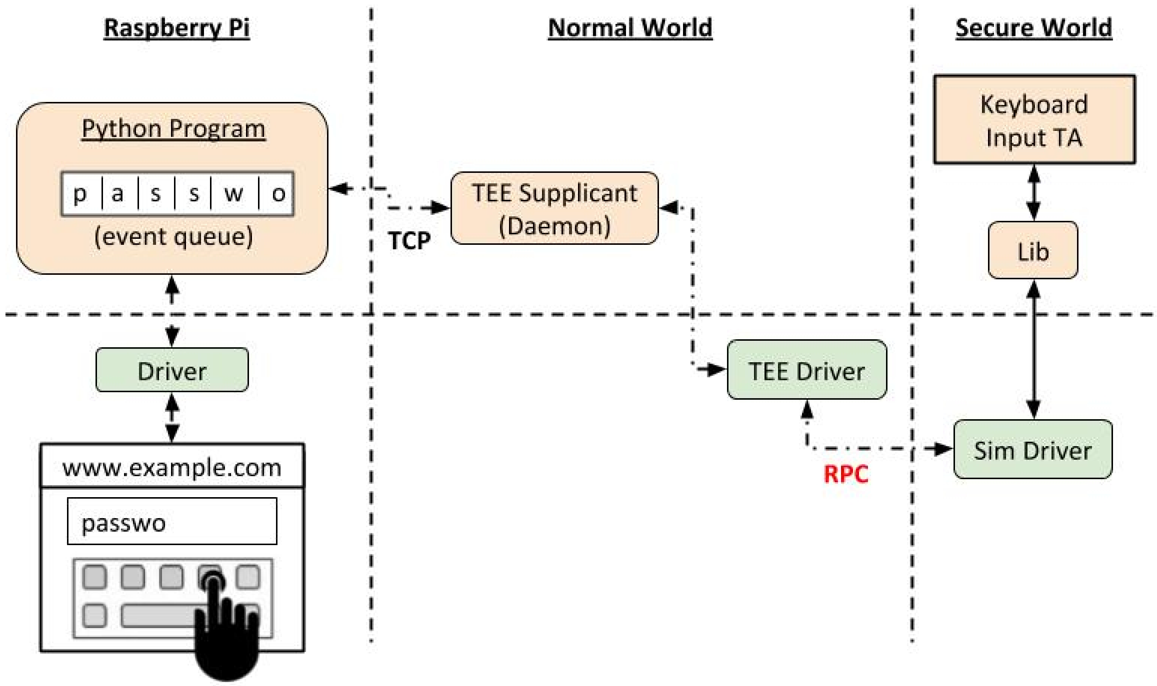

Fig. 5.18.: Event Queue Used By Python Program on Pi Board

shown in Figure 3.14 (right). The Python code will maintain an event queue and accept user input. Each event will correspond to one character typed by the user. The Python code uses the `perf_counter` from the `time` package to keep track of the elapsed time to decide when to return the result. The python code will respond with the character set obtained within the timeout period. If the input has not been terminated, the `TA` will request the next set of typed characters. Once the `TA` verifies the user input has been terminated, the `TA` will use the provided characters as the password input.

5.6 Evaluation

5.6.1 GPS Testing

The setup for doing GPS testing for Truz-Sim involved using a Hikey board with AOSP (version 9) and OP-TEE (version 3.6.0) installed. Three APKs for Google Play store, firebase and Google Play services were also installed as apps need their support in order to run. The packages com.android.vending, com.google.android.gms and com.google.android.gsf were obtained from the website apk mirror [109]. The testing was done only on closed source apps downloaded from the Google Play store. The hardware setup is similar to the picture shown in Figure 5.22, except instead of a camera, a USB GPS dongle [121] is attached to the Pi board. The HDMI display in the figure shows the normal world (Android) from the Hikey board. The Pi board has a Wifi dongle that allows the Hikey to reach it via the network. An example result of testing with a closed source app is shown in Figure 5.19 (results corresponds to the app “My GPS Location”) with latitude and longitude obtained from the TEE. Table 5.2 shows the list of apps successfully tested for latitude and longitude information using Truz-Sim.

Attestation Issue. In addition to using APIs for LMS to obtain location, Android apps can also get location from GMS [118]. During testing it was observed that on the Hikey build, GMS is using LMS to get the location (see Figure 5.20). For the ten apps tested, it is observed that the location provided by simulation can be used by closed source apps (three test apps used LMS and seven used GMS + LMS). LMS and GMS return a

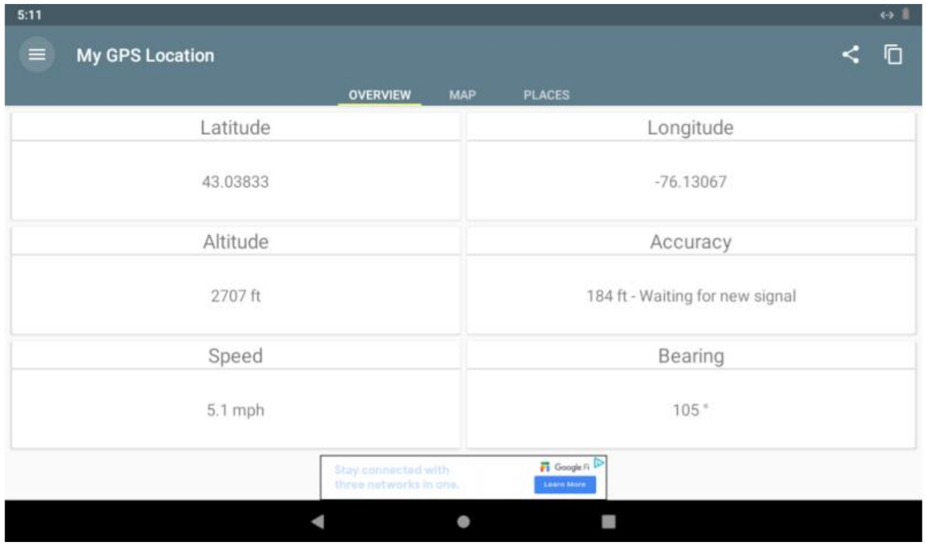

Fig. 5.19.: GPS Test Case

Table 5.2: List of Closed-Source GPS Apps Tested

App Name	**Google Play Link**
My GPS Location	`https://play.google.com/store/apps/details?id=com.digrasoft.mygpslocation`
MapQuest	`https://play.google.com/store/apps/details?id=com.mapquest.android.ace`
Latitude Longitude	`https://play.google.com/store/apps/details?id=com.mylocation.latitudelongitude`
Driving Route Finder	`https://play.google.com/store/apps/details?id=com.virtualmaze.drivingroutefinder`
Foursquare City Guide	`https://play.google.com/store/apps/details?id=com.joelapenna.foursquared`
Accuweather	`https://play.google.com/store/apps/details?id=com.accuweather.android`
Lyft	`https://play.google.com/store/apps/details?id=me.lyft.android`
EventBrite	`https://play.google.com/store/apps/details?id=com.eventbrite.attendee`
Meetup	`https://play.google.com/store/apps/details?id=com.meetup`
HotPads Apartments & Home Rentals	`https://play.google.com/store/apps/details?id=com.hotpads.mobile`

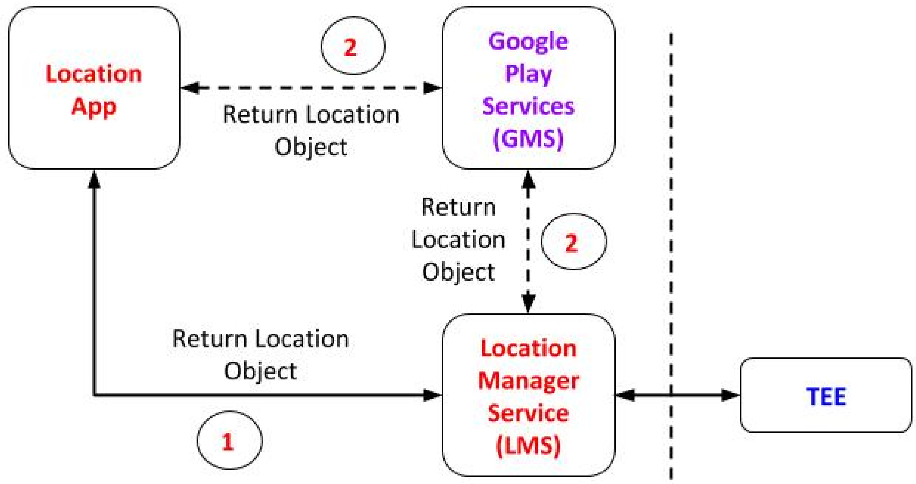

Fig. 5.20.: Paths Used to Obtain Location

Location object to the app. In the current `Truz-Sim` testing, LMS returns a Location object for transparency, and `TEE` simply returns the location information to demonstrate the entire flow works. In case a researcher uses simulation to get the location attestation from the `TEE`, the LMS path will guarantee to provide the attestation to the app, as the attestation can be attached to the Location object. The GMS path does not guarantee this, as the GMS may alter the Location object, reconstruct it or may forward it as it received it.

Performance. To evaluate performance, the time taken to get GPS location using the `Truz-Sim` setup was measured. The timing reported does not include time taken in the app logic. Timing was measured starting when the app requests GPS data, and ending when the GPS data is handed over to the app. The testing was done using the app "MapQuest". The test was repeated 20 times. Figure 5.21 shows the round trip times for individual steps.

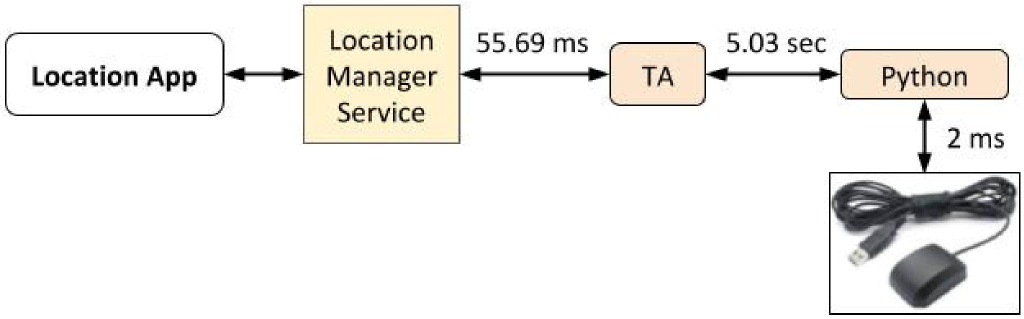

Fig. 5.21.: GPS Simulation Access Performance Breakdown

The most amount of time is taken between the TA and the Python code on the Pi board. The setup used a Pi 3B board with a USB Wifi adaptor and the Hikey using a USB ethernet adaptor. Tests were also done using an ethernet cable attached to the Pi board (Hikey and Pi connected via a switch) and the timing result was similar. The time between the TEE and the Pi board is influenced by several factors. Different Raspberry Pi boards have different networking performance [138]. The Hikey board used in the experiment supports USB 2.0 [123]. More recent version of the board provides USB 3.0 support [124]. If the researcher chooses to use a LAN setup, then the category of the ethernet cable used [117] will affect the transmission speed. In future if SPI can work for the Raspberry Pi board in slave mode, then different performance would be observed compared to using RPC.

The GPS receiver was used at an update rate [120] of 1 Hz and a baud rate of 115200. The GPS receiver was connected via USB 2 which provides a bit rate of upto 480 Mbps (the four USB ports on the Pi board are connected to a common bus operating at max rate of 480 Mbps [139]). GPS receivers used in industry can provide 20 Hz update rate [16] and are connected via [130] the I2C interface (with bit rate of upto 5 Mbps [176]) or I3C interface (using bit rate of 10 to 11 Mbps [132]).

5.6.2 Camera Testing

Similar to the discussion in the previous section, the setup for the testing of the camera use case for `Truz-Sim` involved a Hikey board (AOSP and OP-TEE installed) with necessary Google packages installed. The testing was done using the camera app in the AOSP Hikey build and several closed source apps downloaded from the Google Play store. The hardware setup is shown in Figure 5.22, where a camera module [113] is attached to the Raspberry Pi board via the `CSI` interface. An example result of testing with a closed source app (FastScanner) can be found in the demo video [114]. The test shows the app getting a picture of a QR code via the `TEE`. In the video, the recording camera is put down at time 0:30 for 4 seconds, in order to press the button in the app to request camera capture. Table 5.3 shows the list of apps tested to get a camera picture using `Truz-Sim`. In the table, only the AOSP Camera test is for an app using the v2 API; the rest of the apps in the table used the v1 API.

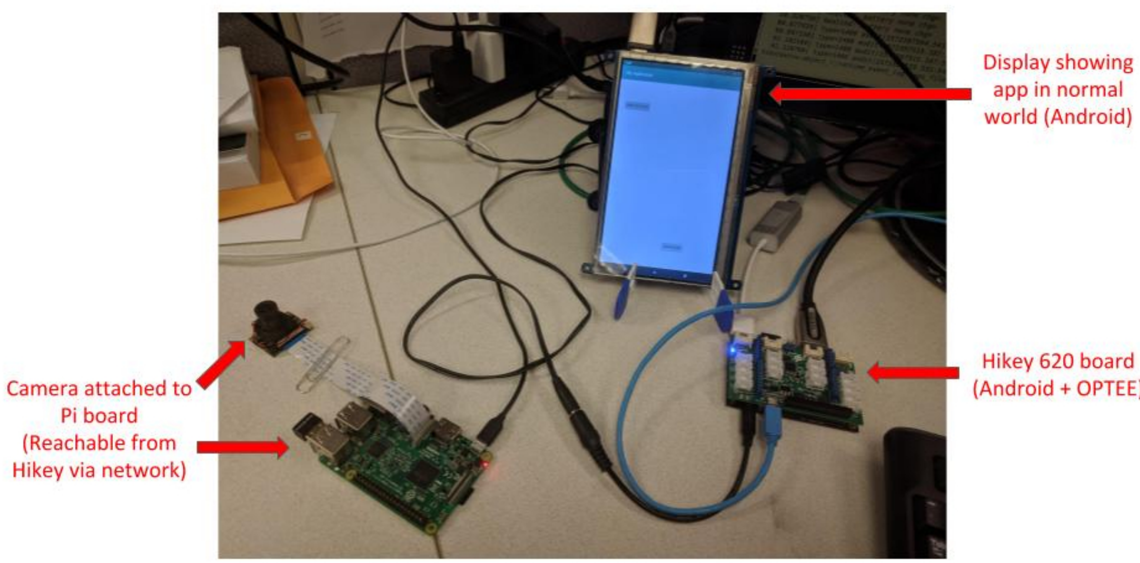

Fig. 5.22.: Camera Test Setup

Table 5.3: List of Camera Apps Tested

App Name	**Google Play Link**
AOSP Camera	Part of Hikey Build
FastScanner	https://play.google.com/store/apps/details?id=com.coolmobilesolution.fastscannerfree
Cam Scanner	https://play.google.com/store/apps/details?id=com.bcaapps.scanner
Clear Scan	https://play.google.com/store/apps/details?id=com.indymobileapp.document.scanner
Document Scanner	https://play.google.com/store/apps/details?id=com.cv.docscanner
ScanBizCards Lite	https://play.google.com/store/apps/details?id=com.scanbizcards
Smart Doc Scanner	https://play.google.com/store/apps/details?id=com.mobilicy.docscanner
Jet Scanner Lite	https://play.google.com/store/apps/details?id=com.stoik.jetscanlite
Receipts by Wave	https://play.google.com/store/apps/details?id=com.waveaccounting.receipts

There were several issues observed while testing camera apps from the Google play store on the Hikey board. The issues corresponded to support not provided by `Truz-Sim` and to various errors observed during runtime. Errors included multi-dex support and `ImageView` / `TextView` inflation errors. Some apps didn't work because they relied on Google's CameraX library [115] or the Mobile Vision API [49]. Other apps didn't work because they didn't follow Google's recommended steps for using the camera API. Some apps applied additional rotation which sometimes results in mirror picture. This can be solved by researchers by applying an additional orientation change (in the python code) based on the test case the researcher is pursuing. In case of the camera app in AOSP's Hikey build, the app can take a picture without any issue, but when a second app requests the camera app for a picture using an `Intent`, the camera app doesn't send the result back via an `Intent`. This issue was not further investigated as it was presumed that the issue would be fixed in the future as the build matures.

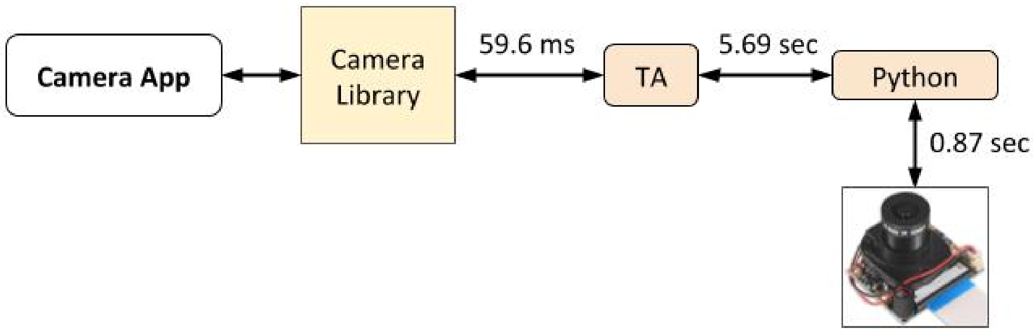

Fig. 5.23.: Camera Simulation Access Performance Breakdown

Performance. To evaluate performance, the time taken to get the camera picture using the `Truz-Sim` setup was measured. The timing reported does not include time taken in the app logic. Timing was measured starting when the app requests the camera to take picture (after user presses the button), and ending when the picture data is handed over to the app. The testing was done using the app "FastScanner". The test was repeated 10 times. Figure 5.23 shows the round trip times for individual steps. The factors influencing the timing between `TEE` and the Pi board are similar to the discussion in Section 5.6.1. The delay in getting the picture from the camera on the Pi board will be influenced by shutter delay and bandwidth of the CSI interface. Before a picture is taken, 100 ms are used in Python code to wait for the camera to warm up. Cameras used with Raspberry Pi use an image capture approach called rolling shutter [38, 177]. This is similar to mobile phone digital cameras. The Raspberry Pi uses MIPI CSI-2 interface with bandwidth of upto 2 Gbps [36]. Cameras used in industry are also connected via CSI [130], with MIPI CSI-3 interface supporting a bit rate of upto 14.88 Gbps [131].

5.6.3 UI Touch Input Testing

Testing UI touch input with `Truz-Sim` involves testing interaction between the normal-world input method framework and the `TA` , and the path used by the `TA` when it uses simulation to get user input. The first part has already been evaluated in chapter 3. This section shows an example for the flow shown in Figure 5.18, involving the `TA` getting user touch input from the Pi board. For this independent test, the `TA` was directly invoked from command line in the normal-world, with the expected returned result to be the password typed by the user.

```
D/TA: TA_CreateEntryPoint:52 has been called
D/TA: TA_OpenSessionEntryPoint:81 has been called
D/TA: ui_test:368 TA - getting user input from Pi
D/TA: sim_test:14 tee_sim.c - sim_test()
D/TA: ui_test:384 TA: Return value: abcd
D/TA: sim_test:14 tee_sim.c - sim_test()
D/TA: ui_test:384 TA: Return value: pass
D/TA: sim_test:14 tee_sim.c - sim_test()
D/TA: ui_test:384 TA: Return value: word
D/TA: ui_test:405 TA - Returning string: abcdpassword
D/TA: TA_DestroyEntryPoint:63 has been called
```

Fig. 5.24.: TA Log When Accessing UI Touch Input

Figure 5.24 shows an example log for the `TA` using simulation to get user input. The log corresponds to a `TA` using maximum number of events as 5 and a timeout of 3 seconds. The occurence of `sim_test()` in the log corresponds to the system call added in OP-TEE. The string ‘abcdpassword’ was typed slowly on the UI displayed by the

Python code to check timeout behavior. In this test the password is simply returned to the normal world. In a real test for `Truz-UI`, the researcher will use the reference concept discussed in chapter 3, save the password in the `TEE` memory, and return a reference corresponding to the password to the normal-world input method framework.

5.7 Discussion

`Truz-Sim` achieves the goal of reducing setup time and reducing hardware experience required on behalf of the researcher in order to setup a hardware test environment to do `TEE` research, given the rich community support available for interfacing peripherals with the Raspberry Pi. For a typical research project where researcher wants to use a peripheral (like sensors, camera etc.) in the design, using `Truz-Sim` will suffice in order to evaluate the feasibility of the researcher's design. Given the current iteration of `Truz-Sim` is based on Hikey and Raspberry Pi, the interface support the researcher will get via the Pi board matches that on Hikey's low / high speed header including `I2C`, `SPI`, `CSI`, `USB`, `I2S`. The researcher can use these interfaces without vendor support in the `TEE`.

`Truz-Sim` has a limitation when it comes to latency. In the performance evaluation for GPS and camera, there is a delay observed between the `TA` and the Pi board when using the `RPC` channel. The time spent on `RPC` and in the Python code on the Pi board is additional overhead, compared to a real phone case [130] where a direct `CSI` or `I3C` bus would be used for communication between the application processor and the peripherals.

The additional latency will impact researchers who want to test a system with real time requirements.

Peripheral bit rate used in simulation depends on the interface support available. Taking camera and GPS as example, the latest iteration of Raspberry Pi at the time of writing provides `CSI` support for camera, but only provides `I2C` support and does not provide `I3C` support for connecting GPS. GPS can also connected via `USB`. Existing developer board hardware support will impact the interface researcher can use in the simulation experiment. This will impact researchers who want to test a given peripheral with the latest interface specification.

6. CONCLUSION AND FUTURE WORK

In summary, this dissertation provides solutions transparent to applications to protect user interaction channels on a mobile platform using ARM TrustZone. The dissertation focuses on the user interaction channels of UI input and audio I/O. First, this dissertation has proposed `Truz-UI`, a transparent design that allows normal-world apps to leverage TrustZone via existing OS APIs to protect user interaction via UI input. The design utilizes a *cross-OS binding* between the UI interaction in the secure world and the code in the normal-world app, allowing the app developer to request a secure version of the UI and provide the code to be bound to this UI. Second, this dissertation has proposed `Truz-Call`, a transparent design to protect users audio I/O during a VoIP call by integrating `TEE` at essential stages in a VoIP apps audio pipeline. The design allows VoIP apps to leverage TrustZone while using existing OS APIs and VoIP protocol, and provides generic TA support so that no app-specific TA code is needed. Lastly this dissertation proposed `Truz-Sim`, a design for a simulation based `TEE` prototyping environment that can allow researchers to interface different category of hardware with the `TEE` OS irrespective of the available support from the vendor. The design utilizes a *cross-OS binding* between the trusted application in the `TEE` and hardware attached to a different OS on a different board like Raspberry Pi. All solutions have been implemented and tested on the TrustZone-enabled Hikey development board.

6.1 Secure Input Interaction for Hybrid Applications

A hybrid mobile application is developed using web technologies like HTML, CSS and JavaScript, and then wrapped in a native application [27, 126]. This is facilitated by an embedded browser component in the native application. In Android, this feature is provided by the `WebView` component [147]. `WebView` allows an app developer to display web content as part of the `Activity` layout. A recent survey [48] shows an increase in preference on part of app developers to adopt hybrid app development. Given the adoption of hybrid apps, there is a need to provide secure user input interaction for cases involving `WebView`. `Truz-UI` can be further extended to cover hybrid apps.

Since Android 4.4 (KitKat), the `WebView` component has been based on the Chromium open source project [148]. The Chromium architecture [153] involves two major components, the browser kernel and the rendering engine. To access operating system functionality such as user interaction, the rendering engine relies on the browser kernel API. In case of text input, Chromium under its content module uses `ImeAdapterImpl` [116] which uses Android framework's `InputMethodManager` to request display of a keyboard. This pattern matches the keyboard request covered in chapter 3. This indicates that the proxy `IME` app can be used to request invocation of a secure keyboard in the `TEE`. Further investigation would be required on how to allow marking of UI elements inside the `WebView` as secure and how to setup the *cross-OS binding* to allow a reference result from the `TEE` to be returned transparently to the web application code inside the `WebView`.

6.2 VoIP Computation Stages in TEE

As shown in Figure 4.5, one of the stages in a VoIP app is marked as *computation*. For input audio, this includes computation like read resampling (downsampling), volume adjustment, equalization and compression. For output audio, this can include decompression, volume adjustment, equalization, and upsampling. `Truz-Call` disables the additional computation stages as end-to-end latency increases with every stage that uses `TEE` (due to invocation time), and the stages will also add to the TCB in the secure world. The design also needs to ensure that no computation stage tampers with the reference data in the normal world.

Instead of supporting these stages by integrating each stage with the `TEE`, an alternate solution can be to move the additional computation stages entirely into the `TEE`. Simply moving the computation stages in their existing form will increase the TCB. Therefore there is a need for a lightweight audio computation pipeline in the `TEE` that can achieve sufficient audio quality improvement. The design for this would need to address the tradeoff of acceptable TCB in the `TEE` vs acceptable audio quality for the VoIP call.

6.3 Expanding Hardware Simulation Support

`Truz-Sim` has so far been tested for camera, GPS and UI touch input. There is further expansion and testing that can be done to demonstrate support for broader variety of peripherals. Referring a mobile system diagram [130], testing can be expanded to include peripherals like fingerprint, baseband, sensors (including accelerometer, gyroscope etc.) and NFC. This can facilitate further tests like fingerprint login where fingerprint data is only accessible by the secure world, and secure SMS with SMS text only visible to the secure world. Testing can also be expanded to ensure simultaneous peripheral access. Current testing involves only one type of peripheral at a time. Further testing can be done to ensure simulation is stable enough to support cases where multiple peripherals need to be accessed, e.g. in facial authentication [183] where camera and accelerometer data is needed. Given that there is no existing work to use `SPI` in slave mode on Raspberry Pi, further investigation can be done as that would provide an alternative channel between the `TEE` and the Pi board.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Enhance Device Security With T6.
https://www.trustkernel.com/en/products/tee/t6.html.
[Accessed: March 27, 2019].

[2] SDP: Session Description Protocol.
https://tools.ietf.org/html/rfc2327, 1998.

[3] 3D RGB Matrix.
https://web.archive.org/web/20200704235426/https://www.researchgate.net/figure/A-three-dimensional-RGB-matrix-Each-layer-of-the-matrix-is-a-two-dimensional-matrix_fig6_267210444, 2001.

[4] GPS - NMEA sentence information.
https://web.archive.org/web/20200606033839/http://aprs.gids.nl/nmea/, 2001.

[5] Linphone - Open Source VoIP project. https://www.linphone.org/, 2001.
[Accessed: Feb 2019].

[6] SIP: Session Initiation Protocol.
https://tools.ietf.org/html/rfc3261, 2002.

[7] RTP Control Protocol Extended Reports (RTCP XR).
https://tools.ietf.org/html/rfc3611, 2003.

[8] RTP: A Transport Protocol for Real-Time Applications.
https://tools.ietf.org/html/rfc3550, 2003.

[9] Introduction to Sound Programming with ALSA.
https://web.archive.org/web/20190421034814/https://www.linuxjournal.com/article/6735, 2004.

[10] The Secure Real-time Transport Protocol (SRTP).
https://tools.ietf.org/html/rfc3711, 2004.

[11] Understanding Delay in Packet Voice Networks.
https://web.archive.org/web/20200529042111/https://www.cisco.com/c/en/us/support/docs/voice/voice-quality/5125-delay-details.html, 2006.

[12] What Is Hardware-in-the-Loop?
http://www.ni.com/en-us/innovations/white-papers/17/what-is-hardware-in-the-loop-.html, 2006.

[13] Talk To Multiple Devices With One UART. https://web.archive.org/web/20190330012545/https://www.electronicdesign.com/4g/talk-multiple-devices-one-uart, 2007.

[14] Whatsapp - Simple, Secure, Reliable messaging. https://www.whatsapp.com/, 2009.

[15] Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). https://tools.ietf.org/html/rfc5764, 2010.

[16] GPS Receiver Features 20Hz Update Rate. https://www.eetimes.com/gps-receiver-features-20hz-update-rate, 2010.

[17] Understanding Canvas and Surface concepts. https://web.archive.org/web/20200709011746/https://stackoverflow.com/questions/4576909/understanding-canvas-and-surface-concepts/38496500, 2011.

[18] ZRTP: Media Path Key Agreement for Unicast Secure RTP. https://tools.ietf.org/html/rfc6189, 2011.

[19] Datagram Transport Layer Security. https://tools.ietf.org/html/rfc6347, 2012.

[20] GPS Sentences. https://web.archive.org/web/20200628203050/https://www.rfwireless-world.com/Terminology/GPS-sentences-or-NMEA-sentences.html, 2012.

[21] Linux Framebuffer Drivers. https://web.archive.org/web/20200619212653/https://moi.vonos.net/linux/framebuffer-drivers/, 2012.

[22] From SIP to RTP - Overview. https://www.informaticapressapochista.com/asterisk/from-sip-to-rtp-part-1/, 2012. [Accessed: Feb 28, 2019].

[23] RTP, Jitter and Audio Quality in VoIP. https://web.archive.org/web/20200528213941/https://kb.smartvox.co.uk/voip-sip/rtp-jitter-audio-quality-voip/, 2012.

[24] Samsung KNOX Whitepaper, 2013.

[25] Signal Messenger. https://www.signal.org/, 2013.

[26] ARM's Reach: 50 Billion Chip Milestone. https://www.broadcom.com/blog/arms-reach-50-billion-chip-milestone-video, 2014.

[27] Native vs Hybrid App Development. https://www.sitepoint.com/native-vs-hybrid-app-development/, 2014.

[28] How to Write a Native Thread and How to Use It. http://shooting.logdown.com/posts/247468-android-native-thread, 2014.

[29] SRTP Crypto - AES ICM. https://github.com/Linphone-sync/srtp/blob/master/crypto/cipher/aes_icm.c, 2014.

[30] The Android Input Architecture. https://web.archive.org/web/20200412022343/http://newandroidbook.com/files/AndroidInput.pdf, 2015.

[31] Electronics Weekly - KoolSpan Encrypting Voice Comms for Secure Channel. https://web.archive.org/web/20200526002658/https://www.electronicsweekly.com/blogs/eyes-on-android/security/koolspan-encrypting-voice-comms-secure-channel-2015-02/, 2015.

[32] Intercede's MyTAM Enabled Enhanced Trust for Android apps to Protect Against Hackers and Malware. https://web.archive.org/web/20200526002924/https://www.intercede.com/investor-news/intercedes-mytam-enables-enhanced-trust-for-android-apps-to-protect-against-hackers-and-malware/, 2015.

[33] Raspberry Pi StackExchange - Can Raspberry Pi function as SPI slave ? https://web.archive.org/web/20200628202509/https://raspberrypi.stackexchange.com/questions/36169/can-raspberry-pi-function-as-spi-slave, 2015.

[34] Secure Storage in OP-TEE. https://www.slideshare.net/linaroorg/sfo15503-secure-storage-in-optee, 2015.

[35] ALSA Driver–HW Buffer. http://www.echojb.com/hardware/2016/12/21/283392.html, 2016.

[36] Raspberry Pi CSI bandwidth. https://raspberrypi.stackexchange.com/questions/51715/what-is-the-speed-of-the-camera-serial-interface-csi-and-its-cable, 2016.

[37] PiCamera. https://picamera.readthedocs.io/en/release-1.13/index.html, 2016.

[38] Picamera - Camera Hardware. https://picamera.readthedocs.io/en/release-1.13/fov.html, 2016.

[39] Basics of the SPI Communication Protocol. https://web.archive.org/web/20200616211306/https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/, 2016.

[40] Trustonic - A Day in the Life of the TEE. https://www.trustonic.com/news/blog/day-life-tee/, 2016.

[41] In-Depth Understanding of the Android Audio Framework. https://blog.csdn.net/ch97ckd/article/details/78641457, 2017. [Accessed: Feb 28, 2019].

[42] AndroidXref Android 7.1.2 - AudioRecord.h. http://androidxref.com/7.1.2_r36/xref/frameworks/av/include/media/AudioRecord.h, 2017.

[43] AndroidXref Android 7.1.2 - AudioTrack.h. http://androidxref.com/7.1.2_r36/xref/frameworks/av/include/media/AudioTrack.h, 2017.

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

[45] Drupal Editor - App by Dissem on Github. https://github.com/Dissem/Drupal-Editor, 2017.

[46] Elgg: A Powerful Open Source Social Networking Engine. https://elgg.org/, 2017.

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

[48] Ionic Developer Survey: App Trends. https://ionicframework.com/survey/2017#trends, 2017.

[49] Mobile Vision Barcode API. https://developers.google.com/vision/android/barcodes-overview, 2017.

[50] AndroidXref Android 7.1.2 - Thread.h. http://androidxref.com/7.1.2_r36/xref/system/core/include/utils/Thread.h, 2017.

[51] Secure Data Path with OPTEE. https://www.slideshare.net/linaroorg/bud17400-secure-data-path-with-optee, 2017.

[52] Meanings of TA FLAGS. https://web.archive.org/web/20190312204816/https://github.com/OP-TEE/optee_os/issues/1590, 2017.

[53] Android Reference Boards. https://web.archive.org/web/20180521054301/https://source.android.com/setup/build/devices, 2018.

[54] GlobalPlatform - TEE System Architecture v1.2. https://globalplatform.org/specs-library/tee-system-architecture-v1-2/, 2018.

[55] Hacking Group Spies on Android Users in India Using PoriewSpy. https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-group-spies-android-users-india-using-poriewspy/, 2018. [Accessed: Feb 28, 2019].

[56] OPTEE OS - Hikey Platform Memory Config. https://github.com/OP-TEE/optee_os/blob/master/core/arch/arm/plat-hikey/platform_config.h, 2018.

[57] RTP, RTCP and Jitter Buffer. https://web.archive.org/web/20200628202653/https://blog.wildix.com/rtp-rtcp-jitter-buffer/, 2018.

[58] Why I2S is Better for Transmitting Audio Compared to SPI. https://web.archive.org/web/20200529014701/https://electronics.stackexchange.com/questions/384328/why-i2s-is-better-for-transmitting-audio-compare-to-spi, 2018.

[59] GlobalPlatform Technology - TEE Internal Core API Specification (Version 1.2). https://web.archive.org/web/20200614015034/https://globalplatform.org/wp-content/uploads/2016/11/GPD_TEE_Internal_Core_API_Specification_v1.2_PublicRelease.pdf, 2018.

[60] I2S Amplifier Breakout. https://www.digikey.com/short/pzp5n4, 2019. [Accessed: April 15, 2019].

[61] How to Change the Keyboard on Your Android Phone. https://www.androidcentral.com/how-set-default-keyboard-your-android-phone, 2019.

[62] Android Developers: Android Keystore System. https://developer.android.com/training/articles/keystore.html, 2019.

[63] Hackers Can Compromise Your Android Phone With a Single Image File. https://web.archive.org/web/20200326001718/https://news.yahoo.com/hackers-compromise-android-phone-single-153148548.html, 2019.

[64] Global Messenger Usage, Penetration and Statistics. https://web.archive.org/web/20200326001944/https://www.messengerpeople.com/global-messenger-usage-statistics/, 2019.

[65] ARM - Changing Exception levels. https://web.archive.org/web/20191018025500/https://developer.arm.com/docs/den0024/latest/fundamentals-of-armv8/changing-exception-levels, 2019.

[66] Android Audio Overview. https://source.android.com/devices/audio/, 2019.

[67] Android Audio Terminology. https://source.android.com/devices/audio/terminology, 2019.

[68] Belle-sip in Linphone architecture. http://linphone.org/technical-corner/belle-sip, 2019. [Accessed: April 2, 2019].

[69] Conquering Android Camera APIs. https://web.archive.org/web/20200703193550/https://infinum.com/the-capsized-eight/conquering-android-camera-api, 2019.

[70] Android Camera API - android.hardware.Camera. https://developer.android.com/reference/android/hardware/Camera, 2019.

[71] Android : Vulnerability Statistics. https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224, 2019.

[72] Hikey (LeMaker). https://www.96boards.org/product/hikey/, 2019. [Accessed: April 15, 2019].

[73] VoIP Basics: About Jitter. https://web.archive.org/web/20200628202555/http://toncar.cz/Tutorials/VoIP/VoIP_Basics_Jitter.html, 2019. [Accessed: April 7, 2019].

[74] JPEG Camera Module. https://web.archive.org/web/20190706225446/https://jpegcamera.com/home-featured-product/sc03mpa/, 2019.

[75] JPEG Camera Manual. https://drive.google.com/file/d/1LeTp-novusl9LDquRAPBmsdwOKs1dqiC/view, 2019.

[76] Linphone - Introduction to Our New RTP Adaptive Jitter Buffer Algorithm. https://web.archive.org/web/20200529042853/https://www.linphone.org/news/introduction-our-new-rtp-adaptive-jitter-buffer-algorithm, 2019. [Accessed: April 7, 2019].

[77] Mediastreamer2 in Linphone Architecture. http://linphone.org/technical-corner/mediastreamer2, 2019. [Accessed: April 2, 2019].

[78] Pew Research Center - Mobile Phone Ownership Over Time. https://www.pewresearch.org/internet/fact-sheet/mobile/, 2019.

[79] Mean Opinion Score for VoIP Testing. https://web.archive.org/web/20200528214158/https://www.voipmechanic.com/mos-mean-opinion-score.htm, 2019. [Accessed: March 23, 2019].

[80] Measuring Voice Quality. https://web.archive.org/web/20190419110821/https://route-test.com/mean-opinion-score-mos-measure-voice-quality-voip/, 2019. [Accessed: April 7, 2019].

[81] Amazon Mechanical Turk. https://www.mturk.com/, 2019. [Accessed: April 7, 2019].

[82] An Inside Look at Nation-State Cyber Surveillance Programs. https://blog.lookout.com/shmoocon-2019, 2019. [Accessed: Feb 28, 2019].

[83] TruzCall Voice Recording - Non-Secure 0 Percent Loss. https://drive.google.com/file/d/1Fz8pR-FjLl4ug5jrt0BZ2KU2cJ5JKUeT/view?usp=sharing, 2019.

[84] TruzCall Voice Recording - Non-Secure 2 Percent Loss. https://drive.google.com/file/d/1pKitr_GN19tP46pAEORQfQ9N8E8vyl0x/view?usp=sharing, 2019.

[85] Open Portable Trusted Execution Environment. https://www.op-tee.org/, 2019. [Accessed: March 27, 2019].

[86] OPTEE OS Drivers (Github). https://github.com/OP-TEE/optee_os/tree/master/core/drivers, 2019.

[87] OPTEE Core Documentation. https://web.archive.org/web/20200625005933/https://optee.readthedocs.io/en/latest/architecture/core.html, 2019.

[88] oRTP in Linphone Architecture. https://www.linphone.org/technical-corner/ortp, 2019. [Accessed: April 2, 2019].

[89] GlobalStats Stat Counter - Mobile Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide, 2019.

[90] Sample Rate Conversion. https://source.android.com/devices/audio/src, 2019.

[91] Samsung Trusted Boot and TrustZone Integrity Management Explained. https://web.archive.org/web/20191227100204/https://insights.samsung.com/2019/09/04/samsung-trusted-boot-and-trustzone-integrity-management-explained/, 2019.

[92] TruzCall Voice Recording - Secure 0 Percent Loss. https://drive.google.com/file/d/1kf77uWONtPZzkMzoMf9eIWCCvQf3P0T_/view?usp=sharing, 2019.

[93] TruzCall Voice Recording - Secure 2 Percent Loss. https://drive.google.com/file/d/1yy-ZzMecRMfbW6ho6xDV9xtye5Regy9L/view?usp=sharing, 2019.

[94] Raspberry Pi Forum - Pi 3 B+ as SPI slave. https://web.archive.org/web/20190918175505/https://www.raspberrypi.org/forums/viewtopic.php?p=1413070, 2019.

[95] Raspberry Pi Forum - SPI Slave Interface on RPi 4. https://web.archive.org/web/20200616212705/https://www.raspberrypi.org/forums/viewtopic.php?t=250788, 2019.

[96] Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2019.

[97] TruzCall Voice Quality Survey. https://drive.google.com/file/d/1HdEPQqIUBORvzESjy2zKUcW-6XnUV3Wr/view?usp=sharing, 2019.

[98] How To Measure VoIP Quality And Jitter. https://web.archive.org/web/20190419065704/https://route-test.com/voip-quality-delay-jitter-measurement/, 2019. [Accessed: March 23, 2019].

[99] Android Developers: Canvas. https://developer.android.com/reference/android/graphics/Canvas, 2020.

[100] Android Dialog. https://developer.android.com/guide/topics/ui/dialogs, 2020.

[101] Android EditText. https://developer.android.com/reference/android/widget/EditText, 2020.

[102] Android - Input Method. https://developer.android.com/guide/topics/text/creating-input-method, 2020.

[103] Android - Input Method Framework. https://developer.android.com/reference/android/view/inputmethod/InputMethodManager#ArchitectureOverview, 2020.

[104] Android InputConnection. https://developer.android.com/reference/android/view/inputmethod/InputConnection, 2020.

[105] Android Specify the Input Method Type. https://developer.android.com/training/keyboard-input/style, 2020.

[106] Android Reference Boards. https://web.archive.org/web/20200312161243/https://source.android.com/setup/build/devices, 2020.

[107] Android Developers: Surface. https://developer.android.com/reference/android/view/Surface, 2020.

[108] Android Fingerprint HAL. https://source.android.com/security/authentication/fingerprint-hal, 2020.

[109] APKMirror. https://www.apkmirror.com/, 2020.

[110] Android Camera API v1. https://developer.android.com/guide/topics/media/camera, 2020.

[111] Android Camera API v2 - android.hardware.camera2. https://developer.android.com/reference/android/hardware/camera2/package-summary, 2020.

[112] Android Camera Architecture. https://source.android.com/devices/camera, 2020.

[113] Raspberry Pi Camera Module. https://web.archive.org/web/20200707220428/https://www.amazon.com/gp/product/B07DNSSDGG/ref=ppx_yo_dt_b_search_asin_image?ie=UTF8&psc=1, 2020.

[114] Truz-Sim Camera Test. https://drive.google.com/file/d/174uYHpo7heROq-Ox_bE18oWbugAtbGez/view?usp=sharing, 2020.

[115] CameraX Overview. https://developer.android.com/training/camerax, 2020.

[116] Chromium ImeAdapterImpl. https://web.archive.org/web/20200718211020/https://chromium.googlesource.com/chromium/src/+/refs/heads/master/content/public/android/java/src/org/chromium/content/browser/input/ImeAdapterImpl.java#515, 2020.

[117] How to Choose an Ethernet Cable. https://www.digitaltrends.com/computing/different-types-of-ethernet-cables-explained/, 2020.

[118] Android Developers - Get the Last Known Location. https://developer.android.com/training/location/retrieve-current, 2020.

[119] Global Platform. https://globalplatform.org/, 2020.

[120] The Basics of GPS. https://learn.sparkfun.com/tutorials/gps-basics/all, 2020.

[121] GPS Dongle. https://www.amazon.com/gp/product/B078Y52FGQ/ref=ppx_yo_dt_b_search_asin_image?ie=UTF8&psc=1, 2020.

[122] Why Doesn't GPS Work Inside a Building? https://itstillworks.com/doesnt-gps-work-inside-building-18659.html, 2020.

[123] User Manual for HiKey 620. https://www.96boards.org/documentation/consumer/hikey/hikey620/hardware-docs/hardware-user-manual.md.html, 2020.

[124] User Manual for HiKey 960. https://www.96boards.org/documentation/consumer/hikey/hikey960/hardware-docs/hardware-user-manual.md.html, 2020.

[125] 96Boards Hikey. https://www.96boards.org/product/hikey/, 2020.

[126] What is Hybrid App Development? https://ionicframework.com/resources/articles/what-is-hybrid-app-development, 2020.

[127] Triple-Axis Accelerometer. https://web.archive.org/web/20200617213051/https://www.adafruit.com/product/2019, 2020.

[128] Android Developers - InputMethodService. https://developer.android.com/reference/android/inputmethodservice/InputMethodService, 2020.

[129] Android Developers - LocationManager. https://developer.android.com/reference/android/location/LocationManager, 2020.

[130] MIPI Alliance Overview. https://web.archive.org/web/20200604215742/https://www.mipi.org/about-us, 2020.

[131] MIPI CSI-3. https://mipi.org/specifications/csi-3, 2020.

[132] MIPI I3C. https://www.mipi.org/resources/I3C-frequently-asked-questions, 2020.

[133] Android Developers: MotionEvent. https://developer.android.com/reference/android/view/MotionEvent, 2020.

[134] OP-TEE: Platforms Supported. https://optee.readthedocs.io/en/latest/general/platforms.html, 2020.

[135] OP-TEE OS - Load TA RPC Example. https://web.archive.org/web/20200704205942/https://github.com/ForgeRock/optee-os/blob/master/core/arch/arm/kernel/ree_fs_ta.c, 2020.

[136] OP-TEE: optee client - tee-supplicant. https://github.com/OP-TEE/optee_client/tree/master/tee-supplicant, 2020.

[137] OP-TEE: optee client - tee-supplicant C code. https://web.archive.org/web/20200704212421/https://github.com/OP-TEE/optee_client/blob/master/tee-supplicant/src/tee_supplicant.c, 2020.

[138] Raspberry Pi Networking Benchmarks. https://www.pidramble.com/wiki/benchmarks/networking, 2020.

[139] USB on Raspberry Pi. https://www.raspberrypi.org/documentation/hardware/raspberrypi/usb/README.md, 2020.

[140] Python pySerial. https://pythonhosted.org/pyserial/pyserial.html, 2020.

[141] Simulation Setup Demo. https://drive.google.com/file/d/1_kuxg2lp_NnOGm1D8U7HsZP_-wyRyvwa/view?usp=sharing, 2020.

[142] Python tkinter. https://docs.python.org/3/library/tkinter.html, 2020.

[143] Koolspan TrustCall. https://koolspan.com/trustcall/, 2020.

[144] Koolspan TrustCall Android FAQ. https://koolspan.com/newsroom/android-support-faqs/, 2020.

[145] Trustonic. https://www.trustonic.com/about-us/, 2020.

[146] I2C to UART Bridge Controller. https://web.archive.org/web/20200617215238/https://www.digikey.com/product-detail/en/diodes-incorporated/PI7C9X762BLE/PI7C9X762BLE-ND/4924179, 2020.

[147] Android Developers: WebView. https://developer.android.com/reference/android/webkit/WebView, 2020.

[148] WebView for Android. https://developer.chrome.com/multidevice/webview/overview, 2020.

[149] Analyzing WhatsApp Calls. https://web.archive.org/web/20200209095556/https://medium.com/@schirrmacher/analyzing-whatsapp-calls-176a9e776213, 2020.

[150] A. Ahlawat and W. Du. TruzCall: Secure VoIP Calling on Android using ARM TrustZone. In *2020 Sixth International Conference on Mobile And Secure Services (MobiSecServ)*, pages 1–12, 2020. URL https://ieeexplore.ieee.org/abstract/document/9042945.

[151] N. AlDuaij, A. Vant Hof, and J. Nieh. Heterogeneous Multi-Mobile Computing. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys 19, page 494507, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366618. doi: 10.1145/3307334.3326096. URL https://doi.org/10.1145/3307334.3326096.

[152] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A System Solution for Sharing I/O between Mobile Systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys 14, page 259272, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327930. doi: 10.1145/2594368.2594370. URL https://doi.org/10.1145/2594368.2594370.

[153] A. Barth, C. Jackson, and C. Reis. The Security Architecture of the Chromium Browser. Technical report, 2008. URL http://css.csail.mit.edu/6.858/2018/readings/chromium.pdf.

[154] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Network and Distributed Systems Security (NDSS) Symposium*, February 24-27 2019. URL https://dx.doi.org/10.14722/ndss.2019.23448.

[155] R. Buhren, J. Vetter, and J. Nordholz. The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture. volume 9977, 11 2016. doi: 10.1007/978-3-319-50011-9_29.

[156] M. Ender, G. Duppmann, A. Wild, T. Poppelmann, and T. Guneysu. A Hardware-Assisted Proof-of-Concept for Secure VoIP Clients on Untrusted Operating Systems. In *Proceedings of 2014 International Conference on ReConFigurable Computing and FPGAs*, ReConFig' 14, Cancun, Mexico, Dec 8-10 2014. URL https://doi.org/10.1109/ReConFig.2014.7032489.

[157] C. Göttel, P. Felber, and V. Schiavoni. Developing Secure Services for IoT with OP-TEE: A First Look at Performance and Usability. In J. Pereira and L. Ricci, editors, *Distributed Applications and Interoperable Systems*, pages 170–178, Cham, 2019. Springer International Publishing. ISBN 978-3-030-22496-7.

[158] Y. Jong, P. Hsiu, S. Cheng, and T. Kuo. A Semantics-Aware Design for Mounting Remote Sensors on Mobile Systems. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016. URL `https://ieeexplore.ieee.org/document/7544382`.

[159] F. Kohnhuser, D. Pllen, and S. Katzenbeisser. Ensuring the Safe and Secure Operation of Electronic Control Units in Road Vehicles. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 126–131, 2019.

[160] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek. PrOS: Light-Weight Privatized Secure OSes in ARM TrustZone. *IEEE Transactions on Mobile Computing*, 19(6): 1434–1447, 2020.

[161] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. SeCloak: ARM TrustZone-based Mobile Peripheral Control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 1–13, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5720-3. doi: 10.1145/3210240.3210334. URL `http://doi.acm.org/10.1145/3210240.3210334`.

[162] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 8:1–8:7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3024-4. doi: 10.1145/2637166.2637225. URL `http://doi.acm.org/10.1145/2637166.2637225`.

[163] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena. DroidVault: A Trusted Data Vault for Android Devices. In *Proceedings of the 2014 19th International Conference on Engineering of Complex Computer Systems*, ICECCS' 14, Tianjin, China, Aug 4-7 2014. URL `https://doi.org/10.1109/ICECCS.2014.13`.

[164] Y. Li and W. Gao. Interconnecting Heterogeneous Devices in the Personal Mobile Cloud. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017. URL `https://ieeexplore.ieee.org/document/8057083`.

[165] D. Liu and L. P. Cox. VeriUI: Attested Login for Mobile Devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, HotMobile' 14, Santa Barbara, CA, USA, February 26-27 2014.

[166] S. Oh, H. Yoo, D. R. Jeong, D. H. Bui, and I. Shin. Mobile Plus: Multi-Device Mobile Platform for Cross-Device Functionality Sharing. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys 17, page 332344, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349284. doi: 10.1145/3081333.3081348. URL `https://doi.org/10.1145/3081333.3081348`.

[167] D. J. Sebastian, U. Agrawal, A. Tamimi, and A. Hahn. DER-TEE: Secure Distributed Energy Resource Operations Through Trusted Execution Environments. *IEEE Internet of Things Journal*, 6(4):6476–6486, 2019.

[168] 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 and Networks*, DSN' 15, Rio de Janeiro, Brazil, June 22-25 2015.

[169] Trustonic. Trustonic TEE Trusted User Interface. https://www.trustonic.com/news/blog/benefits-trusted-user-interface/, 2019.

[170] Wikipedia. Adaptive Differential Pulse-Code Modulation — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Adaptive_differential_pulse-code_modulation, 2018.

[171] Wikipedia. Differential Pulse-Code Modulation — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Differential_pulse-code_modulation, 2018.

[172] Wikipedia. Packet Loss — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Packet_loss, 2019.

[173] Wikipedia. Stream Cipher Attacks — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Stream_cipher_attacks, 2019.

[174] Wikipedia. Comparison of VoIP Software — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Comparison_of_VoIP_software, 2019.

[175] Wikipedia. Framebuffer — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Framebuffer, 2020.

[176] Wikipedia. I2C — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/I%C2%B2C, 2020.

[177] Wikipedia. Rolling Shutter — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Rolling_shutter, 2020.

[178] Wikipedia. Voice over ip — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Voice_over_IP", 2020.

[179] S. D. Yalew, G. Q. M. Jr., and M. Correia. Light-SPD : A Platform to Prototype Secure Mobile Applications. In *Proceedings of Workshop on Privacy-Aware Mobile Computing*, PAMCO' 16, Paderborn, Germany, July 05 2016. URL https://dl.acm.org/doi/10.1145/2940343.2940349.

[180] K. Ying. *Integrating TrustZone Protection with Communication Paths for Mobile Operating System*. PhD thesis, Syracuse University, 2019. URL https://surface.syr.edu/cgi/viewcontent.cgi?article=2019&context=etd.

[181] 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 Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 14–27, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5720-3. doi: 10.1145/3210240.3210338. URL http://doi.acm.org/10.1145/3210240.3210338.

[182] 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.

[183] D. Zhang. Trustfa: TrustZone-Assisted Facial Authentication on Smartphone. Technical report, 2014. URL http://www.donglizhang.org/trustfa.pdf.

[184] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. Minimal Kernel: An Operating System Architecture for TEE to Resist Board Level Physical Attacks. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 105–120, Chaoyang District, Beijing, Sept. 2019. USENIX Association. ISBN 978-1-939133-07-6. URL https://www.usenix.org/conference/raid2019/presentation/zhao.

[185] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. SecTEE: A Software-Based Approach to Secure Enclave Architecture Using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS 19, page 17231740, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3363205. URL https://doi.org/10.1145/3319535.3363205.

VITA

VITA

Amit Ahlawat received his Bachelor of Technology degree in Computer Engineering from Maharshi Dayanand University, Haryana, India. He received his Masters of Science degree in Computer Engineering from Syracuse University (Syracuse, New York, USA). This dissertation was defended in October 2020 at Syracuse University.