A Trusted-Hardware Backed Secure Payments Platform for Android

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Technology (Research)

IN THE

Faculty of Engineering

BY

Rounak Agarwal

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

October, 2021

Declaration of Originality

I, **Rounak Agarwal**, with SR No. **04-04-00-10-22-17-1-14932** hereby declare that the material presented in the thesis titled

A Trusted-Hardware Backed Secure Payments Platform for Android

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018-21**.

With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

DEDICATED TO

my parents and my advisor

Acknowledgements

Firstly, I wish to express my gratitude for my advisor Prof. Vinod Ganapathy and my co-advisor Prof. K. Gopinath. They have been very supportive and patient with me throughout my time at Indian Institute of Science. I am also very grateful towards my employer NVIDIA Graphics Pvt. Ltd. and my manager Mr. Vipin Kumar for allowing me to continue working on my thesis concurrently with my work for NVIDIA. I wish to thank my labmates in Computer Systems Security Lab – Subhendu, Kripa, Aditya, Rakesh, Ajay, Arun, Nikita and Chinmay for making the lab feel like a second home. Lastly, I wish to thank the administrative staff at the Department of Computer Science and Automation for all that they do for the students.

Abstract

Digital payments using personal electronic devices have been steadily gaining in popularity for the last few years. While digital payments using smartphones are very convenient, they are also more susceptible to security vulnerabilities. Unlike devices dedicated to the purpose of payments (e.g. POS terminals), modern smartphones provide a large attack surface due to the presence of so many apps for various use cases and a complex feature-rich smartphone OS. Because it is the most popular smartphone OS by a huge margin, Android is the primary target of attackers. Although the security guarantees provided by the Android platform have improved significantly with each new release, we still see new vulnerabilities being reported every month. Vulnerabilities in the underlying Linux kernel are particularly dangerous because of their severe impact on app security. To protect against a compromised kernel, some critical functions of the Android platform such as cryptography and local user authentication have been moved to a Trusted Execution Environment (TEE) in the last few releases. But the Android platform does not yet provide a way to protect a user's confidential input meant for a remote server, or, the server's confidential output meant for the user, from a compromised kernel. Our work aims to address this gap in Android's use of TEEs for app security. We have proposed an API that a Trusted App running in a TEE can provide to the untrusted apps running in the REE (Rich Execution Environment). This API will allow app developers to leverage the TEE's protection for fetching confidential input from and showing confidential output to the user. We have described how this API can be used to implement a secure payment system that can prevent fraudulent transactions even in the presence of a compromised kernel. We have implemented the proposed API on a device with a TEE built on ARM's TrustZone technology.

Contents

List of Figures

Chapter 1

Introduction

Like a physical wallet allows the owner to store cash and plastic payment cards, a digital wallet allows storage of payment card details and digital cash which can then be used for digital payments. Some digital wallet services like Paytm [80], FreeCharge [75], PhonePe [82], Venmo [84] allow a user to transfer money from their bank accounts to wallets managed by these services using their phone numbers as an account identifier. This digital money can then be used to pay merchants or other users of that digital wallet. Some other digital wallet services manage payment card information instead of managing digital money. Google Pay [76] and Samsung Pay [83] fall in this category. These apps make it unnecessary to carry plastic cards thereby reducing the risk of card theft and card skimming.

The ubiquity of smartphones and the availability of affordable Internet have given a massive boost to cashless payments in recent years. Before smartphones became so commonplace, plastic payment cards were the dominant instrument for cashless payments. Nowadays digital wallet apps account for a significant chunk of cashless payments and this chunk is growing year on year. The Indian Government has been promoting cashless payments under its Digital India campaign [74]. The 2016 Indian banknote demonetisation [127] also increased the popularity of digital wallets. Unified Payments Interface (UPI) [73], launched by National Payments Corporation of India (NPCI) in 2016 and updated in 2018 has been an important contributor in the growth of cashless payments in India. It is a payment system that allows instantaneous fund transfer between bank accounts (possibly belonging

to two different banks) using a virtual payment address (VPA) that is linked to the recipient account. Use of VPA instead of phone numbers is better for privacy of users. The growing popularity of digital wallets has attracted the attention of cybercriminals.

Digital wallets handle private data such as passwords, payment card numbers, card security codes, transaction PINs, etc. Secure storage and transmission of this data is of utmost importance. If cybercriminals somehow get access to this confidential data, they can steal funds from a victim's wallet or bank account causing financial loss to the victim and loss of reputation to the service provider. To ensure the confidentiality of payment card information, all entities that participate in the payment system such as merchants, payment gateways, payment networks, banks, etc. who store and transmit card details, are required to comply with the PCI DSS (Payment Card Industry Data Security Standard) [52]. PCI DSS compliance is enforced by PCI SSC (PCI Security Standards Council)[81]. Some digital wallet services have opted to use tokenisation to avoid having to store card details. They use a token as a surrogate for the actual card number. The user enters card details in the app which is used to generate a token in a secure backend server. The app does not store the card number once the token is obtained. The token is used for payments and for actual fund transfer the token is detokenised to get the associated card number. Since payment is done using the token, the card number is never exposed. Google Pay and Samsung Pay make use of tokenisation. In 2014, EMVCo released a specification called EMV Payment Tokenisation Specification [28] that defines a technical framework for generation and use of payment tokens. In this framework entities called Token Service Providers (TSPs) handle the tokenisation/detokenisation and entities called Token Requestor request tokens which are used for payments. Google Pay is an example of a Token Requestor and Visa is an example of a TSP. While standards like PCI DSS and EMV Payment Tokenisation have resulted in significant improvement in the security of cashless payments, these standards were not meant to protect against attackers that have managed to gain control of user devices.

To secure a payment transaction, a digital wallet service needs to meet the following requirements:

1. Every transaction should be initiated by the user

2. The user's consent for the transaction must be obtained after displaying important details like transaction amount, purpose of transaction and the recipient

3. Any authentication credentials such as login passwords, PINs, CVVs, etc. must be kept confidential

If the Android OS is compromised by malware, then even a perfectly written app cannot meet these requirements and comply with the standards mentioned above. Using the compromised OS, an attacker can initiate transactions on behalf of the user, show the user false information on the UI to trick him into approving an undesirable transaction or steal confidential information like passwords. Monthly Android Security bulletins [19] almost always report vulnerabilities in the kernel or some other privileged component of the Android platform that can allow an attacker to compromise the security of any app. *Trusted Execution Environments* (TEEs) can help mitigate the risks posed by a compromised OS.

Trusted Execution Environments provide isolated execution of trusted software, secure storage of confidential data and remote attestation [126]. Since most Android smartphones have an ARM processor, ARM's implementation of TEE, called *ARM TrustZone* [88] is the most commonly deployed TEE in smartphones. If all the critical functionality of a payment app is moved to a TEE, then the app's security won't be compromised even in the event of OS compromise. Encryption and storage of confidential data such as card number can be done by the TEE. The TEE can present the user with a secure UI to input confidential data such as a transaction PIN. In the last few releases, Android OS has introduced security enhancements that make use of TEEs. *Android Oreo* (8.0) introduced *hardware attestation* of keys. This can allow a payment server to remotely verify whether the cryptography keys being used on the user's device are being stored securely in a TEE. *Android Pie* (9.0) introduced *Protected Confirmations*. Protected Confirmation enables a server to get the user's confirmation for any action (e.g. payment transaction) securely after informing the user about the important details of the action (e.g. transaction amount). Since the server can verify whether

the confirmation was sent by the TEE, a compromised OS cannot fool the server by sending a fake confirmation without the user's knowledge or consent. Protected Confirmation does not provide confidentiality. So, it cannot be used by an app to securely obtain secrets such as passwords and transaction PINs from the user. Till date, Android has not introduced any API that enables use of TEEs for obtaining confidential input from the user. In addition to confidential input, sometimes apps might need a way to show confidential output to the user. Android does not yet have a TEE-leveraging API for confidential output either. Some OEMs like Samsung have implemented confidential input functionality in their smartphones. But this functionality is either available only to apps developed by the OEMs themselves (e.g. Samsung Pay) or available to third party apps via SDKs that work only on the specific OEM's smartphones (e.g. Samsung KNOX SDK). To make TEE-protected confidential input/output available to third party apps in an OEM-independent way, an API needs to be added to the Android Platform. Our work attempts to address this gap.

The contribution of this work is as follows:

1. A platform-agnostic design for protecting a user's confidential input and an app's confidential output using a TEE

2. A payment system that makes use of this design to secure all payment transactions

3. An implementation of the design for Android devices with ARM TrustZone

The rest of this thesis is organized as follows. **Chapter 2** gives relevant background about Android, ARM TrustZone and important standards. **Chapter 3** describes the threat model and the design. **Chapter 4** gives the Android-specific and TrustZone-specific implementation details of our design and performance measurements. **Chapter 5** describes some related work and **Chapter 6** concludes the thesis.

Chapter 2

Background

2.1 Trusted Execution Environments

The term *Trusted Execution Environment* was introduced by Open Mobile Terminal Platform (OMTP) [48] in their standard *Advanced Trusted Environment : OMTP TR1* [47] published in 2009. The standard has two sets of security requirements named Profile 1 and Profile 2 and any execution environment that conforms to one of these two is considered a Trusted Execution Environment. GlobalPlatform [31], a standards organization, in its standard *TEE System Architecture* [32] describes Trusted Execution Environment as "an execution environment that runs alongside but isolated from an REE" where REE stands for Rich Execution Environment. The REE is supposed to run a Rich OS, which has rich functionality and provides a wide variety of features to applications, whereas the TEE typically runs a relatively simple OS called a Trusted OS. Figure 2.1 shows the typical architecture of a system which has a TEE. Based on the standards mentioned above and other works that describe TEEs [87, 121, 126], here are some essential properties of a TEE.

- **Secure Boot**: The Trusted OS that is executed in the TEE needs to be booted securely. The executable for the Trusted OS needs to be stored in a protected partition in the non-volatile storage available on the device. Only the Trusted OS should have write access to the protected partition to prevent a potentially malicious Rich OS from updating it with a compromised

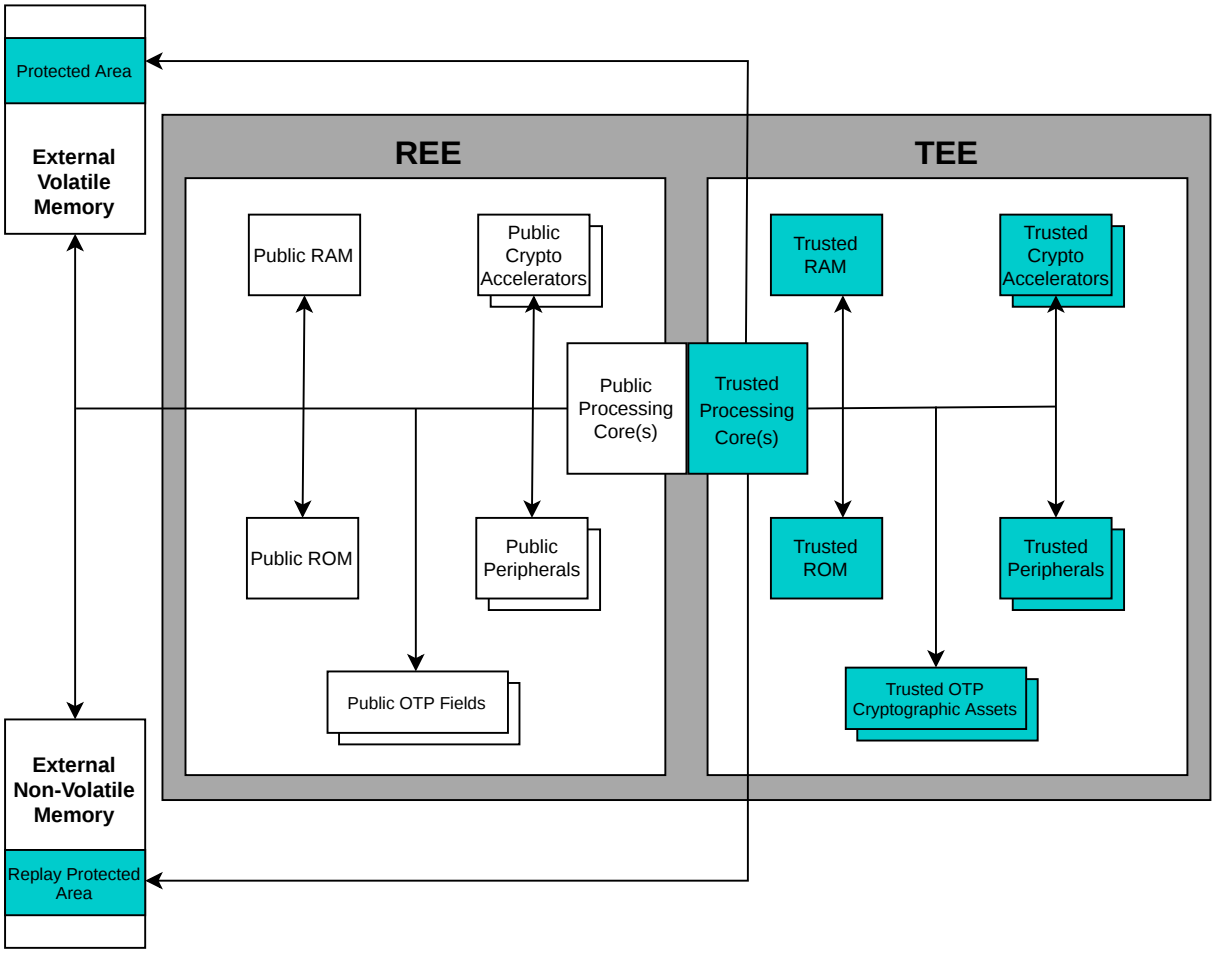

Figure 2.1: Hardware Architectural View of REE and TEE [32]

executable. The most common way of ensuring secure boot is to use a *chain of trust* rooted by a public key stored by the device manufacturer in tamper resistant read-only memory on the device. The corresponding private key is kept securely at the factory. The bootloader that runs immediately after a cold boot is signed by the manufacturer using this private key and during a cold boot the hardware must verify this signature using the public key stored in the device. The verified bootloader contains public key that is required to verify the signature of the next stage in the boot flow which in turn might have the public key for the next stage. In this way a chain of public keys is

used to ensure that only code trusted by the manufacturer runs in the TEE. The manufacturer (or any of its authorised partners) can release an updated version of any of the executables involved in booting, provided the cold-boot bootloader's signature is verifiable using the root public key stored in ROM. The root public key cannot be modified once the device leaves the factory.

- **Isolated execution**: The Trusted OS and the Trusted Applications (TAs) that run on top of it are isolated from the Rich OS and the untrusted applications. The TEE ensures the confidentiality and integrity of the code and data used by it. To enable this isolation, the hardware needs to have some way of partitioning the main memory, caches and the CPU registers into secured and unsecured parts. Only the Trusted OS should have access to the secured part. For I/O it would be impractical to have an entirely separate set of peripherals for the Trusted OS. A more practical solution is for the hardware to provide the Trusted OS with some way of seizing control of a peripheral from the untrusted OS and yield it once it is done using that peripheral. For example, the Trusted OS may need control over the display device to show some confidential information to the user.

- **Secure Storage**: To protect data at rest the Trusted OS needs access to a protected storage device or a protected region on shared storage device. Secure storage is necessary for storing cryptography keys, access credentials and other confidential data. To prevent downgrade attacks in which malicious actors attempt to modify code and/or data to an older valid (but more vulnerable) version, some form of rollback prevention is required. Some modern storage devices have a separate partition called Replay Protected Memory Block (RPMB). To write to RPMB, a MAC needs to be calculated using an authentication key. Both the TEE and the storage device are provisioned with an authentication key at the factory to enable the TEE to write to the RPMB. RPMB can be used to securely store counters, which in turn can help in preventing rollback attacks.

- **Remote Attestation**: The ability of a TEE to prove to a remote third party that a message was sent by the TEE is called remote attestation. Remote

attestation enables the TEE to share the networking hardware on the device with the REE. All messages exchanged between the remote third party and the TEE can go through untrusted networking hardware without getting (undetectably) tampered with. Remote attestation is possible only with an authenticated channel of communication between the TEE and the third party. This authenticated channel cannot be established using an untrusted channel during runtime. It has to be provisioned at the time of manufacturing. It is typically done with the help of an asymmetric key pair stored in secure storage at the factory. The public key in the pair is certified by a Certificate Authority (CA) that is trusted by the third party. During runtime, this certified key pair can be used in a key exchange protocol to agree on one or more shared secret key(s) which in turn help create a protected communication channel between the TEE and the third party.

- **Trusted Path**: A trusted path [50] is a protected channel of communication between the user and the TEE. The trusted path in combination with remote attestation helps in protected communication between a remote third party and the user. For example, the remote third party could be a bank asking for the user's consent to go ahead with a transaction on the user's account. The TEE can get confidential input from and show confidential output to a user using a trusted path. In addition to having control over an I/O device, the TEE needs a security indicator to inform the user that they are communicating with the TEE and not the REE. In absence of a security indicator, the untrusted OS can easily fool the user by mimicking the UI of a TA and get them to share confidential input.

Hardware technologies such as Platform Security Processor from AMD [7, 8], Software Guard eXtensions (SGX) from Intel [61], TrustZone from ARM [88] and MultiZone from Hex Five Security [45] have been developed to enable implementation of TEEs. We have used TrustZone in our implementation. But our design described in Chapter 3 can be implemented with any TEE that has secure access to a touchscreen.

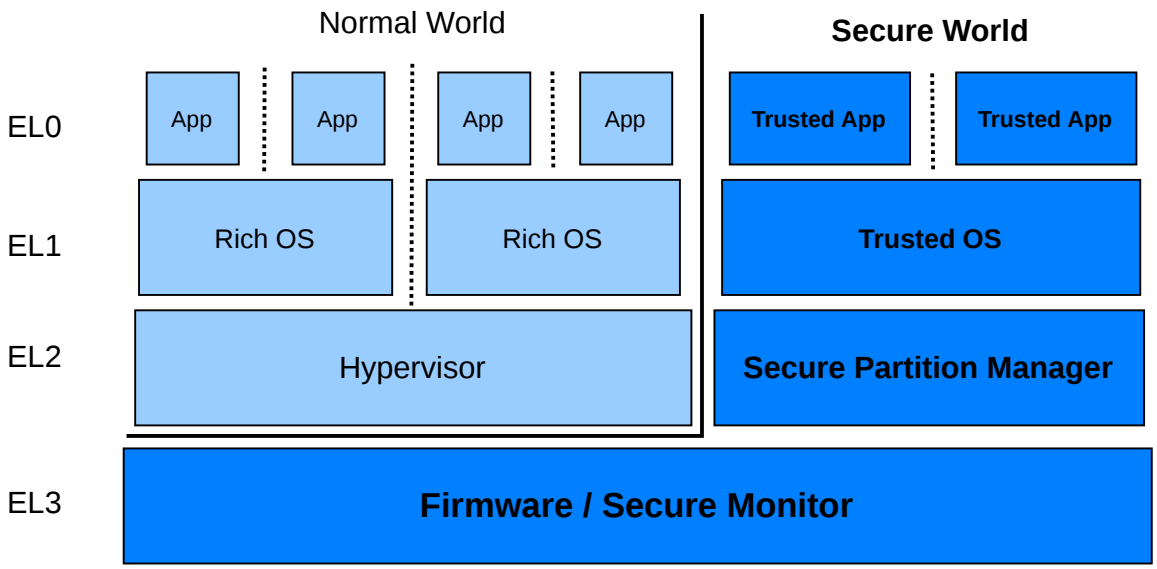

Figure 2.2: Exception Levels in ARMv8-A architecture [14]

2.2 ARM TrustZone

TrustZone [14, 88, 116] is the name of security extensions first made available in 2003 in ARM's ARM1176JZF-S processor [12] which implements the ARMv6 instruction set. Subsequently, ARM continued offering TrustZone in its Cortex-A processors that implement ARMv7-A or ARMv8-A instruction sets. TrustZone helps create two protection domains called *normal world* and *secure world*. At any given point in time, each processor core (referred to as PE in official documents) operates in one of these two domains. TrustZone introduced the Secure Configuration Register (SCR) in which the 0th bit is called NS (Non-secure) bit. When this bit is set, it means the core is in the normal world and if the bit is reset, it indicates the core is in the secure world. A new processor mode (exception level in ARMv8) called Monitor is responsible for preserving processor state when the processor switches between worlds. The monitor sets or resets SCR.NS bit to indicate which world the processor is executing in. The AMBA AXI system bus is TrustZone-aware and uses the NS bit as a 33rd address bit to determine whether an access request came from the secure or normal world. The two worlds share the

same caches in which the tags indicate which world a cache line belongs to. The two worlds have separate base registers pointing to translation tables (page tables) for virtual memory management. The Generic Interrupt Controller v3 (GICv3) supports TrustZone. The GIC can be programmed to treat some interrupt sources as secure so that the interrupts from those sources are delivered only to the secure world. Only software executing in the secure world is allowed to program the GIC. Figure 2.2 shows the four exception levels in ARMv8-A architecture along with the part of the software stack that is supposed to execute at each of the levels. EL2 is an optional extension that may or may not be available in an ARM processor. Until ARMv8-A v8.3, EL2 was only available in the normal world. ARMv8-A v8.4 introduced EL2 in the secure world to enable concurrent execution of multiple Trusted OSs (not shown in the figure). TrustZone added a new privileged instruction to ARM's instruction set - *SMC* (Secure Monitor Call). When the SMC instruction is executed in either of the two worlds, the Secure Monitor mode is entered and depending on the arguments provided with the instruction, the Secure Monitor software determines whether a world switch is required to serve the request. Typically it is the normal world that executes SMC to request some services from the firmware residing in EL3 or from the Trusted OS. EL0 cannot execute SMC in either of the worlds. ARM has published a specification named *SMC Calling Convention* [62] to standardize the interface between EL3 and other exception levels.

ARM has defined some TrustZone-aware peripherals to enable the secure world to create protected regions in memory and control access to peripherals. The TrustZone Address Space Controller (TZASC) allows marking some regions in the DRAM as secure and thus only accessible by the secure world. The TrustZone Memory Adapter (TZMA) performs the same function for on-chip SRAM. TrustZone Protection Controller (TZPC) allows the secure world to control access to various peripherals on the system e.g. the touchscreen. These three components are optional and are not necessarily present in all ARM-based Systems-on-Chip (SoCs).

2.3 GlobalPlatform API Standards

The Rich OS and the Trusted OS need to interoperate even when they are developed by different manufacturers. GlobalPlatform [31] has published several specifications with stardard APIs [33] which manufacturers can adhere to, to ensure interoperability with software developed by other manufacturers. The important API standards published by GlobalPlatform are listed below.

- **TEE Client API**: This standard defines the API used by the untrusted applications running on top of the Rich OS (called Client Applications in the standard) for communicating with and requesting services from Trusted Applications running in the TEE. This API serves as a base layer upon which higher level protocols for things like cryptography, secure storage can be built.
- **TEE Internal Core API**: This standard defines an API for developing Trusted Applications that run in the TEE. In addition to a core API, the standard also defines a Trusted Storage API, a Cryptographic Operations API, a Time API and an Arithmentic API.
- **Trusted User Interface API**: This standard defines an API that can be used by Trusted Applications running in a TEE on a device with a touchscreen (or a screen and a keyboard) for creating a UI for the user. This trusted UI can be used to display sensitive information to the user or fetch sensitive input from the user.
- **TEE Trusted User Interface Low-level API**: While Trusted User Interface API only supports creation of simple PIN entry screens and message boxes, this standard enables creation of more complex UIs.

2.4 ARM Trusted Firmware

To assist OEMs in adopting TrustZone technology for their SoCs, ARM developed and released an open source reference implementation of secure world software called *ARM Trusted Firmware* [15]. OEMs can use ATF as a good starting point for developing secure world software for their SoCs. ATF implements secure boot.

The bootloader stages are listed below in the order that they are executed in a cold boot.

- *Application Processor Trusted ROM (BL1)*: Unmodifiable code that performs the minimum initialization necessary to validate, load and hand off control to the 2nd stage loaded in RAM.
- *Trusted Boot Firmware (BL2)*: Performs additional initialization followed by validating and loading BL31 through BL33 stages.
- *EL3 Runtime Software (BL31)*: Unlike previous 2 stages this stage is not discarded after execution. It executes in EL3 and provides runtime services such as handling transitions between secure and normal worlds. During boot it initializes and hands off control to BL32 and BL33.
- *Secure-EL1 Payload (BL32)*: This is an optional stage. It is usually (but not necessarily) a Trusted OS.
- *Application Processor Normal World Firmware (BL33)*: Boots the Rich OS.

BL31 contains a Secure Payload Dispatcher (SPD) that dispatches service requests from the Rich OS to the Secure Payload. Different Secure Payloads have different SPDs and ATF needs to be compiled with the appropriate SPD before being flashed onto a device.

2.5 OP-TEE

Open Portable Trusted Execution Environment (OP-TEE) [49] is an open source Trusted OS that conforms to the GlobalPlatform standards. It was open sourced in June 2014 after being jointly developed by STMicroelectronics and Linaro's Security Working Group (SWG). Since 2015, SWG is the sole owner and maintainer of OP-TEE. Besides the Trusted OS, SWG has created a library *libutee* that implements *GP TEE Internal Core API*, another library *libteec* that implements the *GP TEE Client API* and a driver for Linux kernel (upstreamed in 2017) that helps normal world apps communicate with Trusted Apps that run on top of OP-TEE. Figure 2.3 shows the architecture of OP-TEE and how it interacts with Linux kernel. *tee-supplicant* is a normal world daemon that OP-TEE communicates with

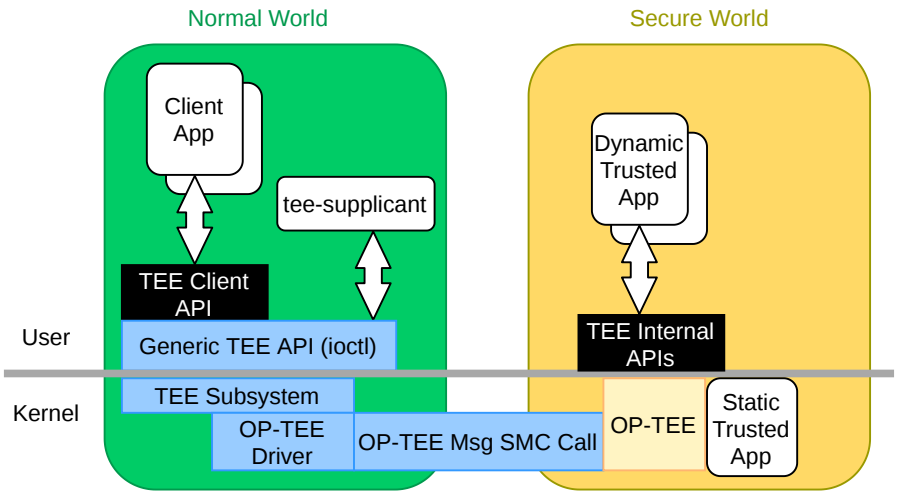

Figure 2.3: OP-TEE Architecture [41]

using RPC for some functionalities like secure storage which require participation of the normal world. A Static Trusted App (also called Pseudo Trusted App) is statically built into the OP-TEE OS binary. Pseudo TAs cannot make use of the TEE Internal Core API because libutee is not linked with the OS binary. Dynamic TAs (also called User mode TAs) are TAs in the true sense that link against libutee to make use of TEE Internal Core API. User mode TAs, as their name suggests, run in Secure-EL0 level unlike Static TAs which run in Secure-EL1. The SMC calls made by the kernel driver go through an SPD called OP-TEE Dispatcher (not shown in the figure) that is part of ATF.

2.6 Security Enhancements in Android

Since the Android platform was launched more than a decade ago, the security provided by the platform has improved with each new release due to various enhancements [23]. This section describes a few of those enhancements that are relevant to the security of payment applications.

2.6.1 Verified boot

Verified boot was introduced in Android 4.4 (KitKat) along with a kernel feature called *dm-verity* [24, 40]. dm-verity enables transparent integrity checking of block devices by the Device Mapper framework in the Linux kernel [38]. A hash tree

containing the hashes of all physical blocks (typically 4K in size) of a particular partition in the storage device is created at the time of building the OS. The root hash of this hash tree is signed by the device manufacturer. A table containing this root hash and some metadata about the partition is written to the disk in the block immediately following the partition which is to be verified. The public key required for verification of the hash is written to the *boot* partition which also contains the kernel. Only partitions that are mounted read-only can be verified using dm-verity because any modifications to the partition's contents will change the hashes. The *system* partition in an Android device contains the Android framework and is mounted read-only. Therefore it is an ideal candidate for dm-verity. For dm-verity to be reliable, the kernel needs to be trusted. This requirement is fulfilled using verified boot. The boot image is signed by the manufacturer and the signature is verified by the bootloader during device boot. Until Android 6 (Marshmallow), the user was warned when the verification of boot image failed but boot was allowed. In Android 7 (Nougat) strict verification was introduced because of which boot failed whenever verification failed. Android 7 also made changes to dm-verity to enable use of error-correction for recovery from minor disk corruptions. For the Android 8 (Oreo) release, the Android framework was completely re-architected as part of Project Treble [71], such that all vendor-independent components could live in the *system* partition and all vendor-dependent components in the newly introduced *vendor* partition. A reference implementation of verified boot called *Android Verified Boot* that works with Project Treble architecture was released with Android Oreo. AVB makes use of a data structure called VBMeta stored in a newly introduced *vbmeta* partition [16]. VBMeta contains the hash for the boot image and the root hashes of the hash trees for the system and vendor partitions. VBMeta also contains a *rollback index*, which is a number that is increased each time an updated image (with security fixes) is installed on the device. The rollback index is used to prevent rollback attacks in which an attacker attempts to install and boot an older image with known vulnerabilities. The vbmeta image is signed by the manufacturer and the signature is verified by the bootloader.

2.6.2 Hardware-backed Keystore

The Android Keystore system was introduced in Android 4 (Icecream Sandwich) [22]. The Keystore system provides a container for securely storing cryptographic keys, preventing extraction of keys from the device. Android 6 introduced the ability to require local user authentication (e.g. PIN entry) before a key can be used. Android 6 also introduced the ability to store keys in secure hardware such as a TEE or a Secure Element. With the release of Android 7, Google mandated that the Keystore system be protected using secure hardware [68]. Android 7 also introduced *Key Attestation* [18, 39]. Key attestation enables an Android app to prove to a remote server that a cryptographic key has been generated and stored in secure hardware. An app can request the hardware-backed Keystore system to create a chain of X.509 certificates that attest various properties of a key such as whether the key requires user authentication, whether it requires fingerprint authentication and other useful information. The first certificate in this chain contains a description of the key in an extension (RFC 5280 § 4.2 [57]). The format of data in this extension is specified in official reference documentation [39]. To prevent replay attacks, the app obtains a 64 bit nonce from the server which is included in the extension by the Keystore system. Android 8 made support for Key Attestation mandatory. Android 8 also introduced optional support for *ID Attestation*. ID Attestation allows a device to securely provide a proof for hardware identifiers such as IMEI numbers, serial numbers, etc. At the factory, the hardware identifiers are copied into secure storage protected by the TEE. ID Attestation builds on top of Key Attestation. The hardware identifiers are added to the extension that is created for Key Attestation. Android 9 (Pie) introduced the ability to securely import keys generated elsewhere into the hardware-backed Keystore. For this purpose, an app first requests the Keystore to generate a wrapping key pair and an attestation for the public key. The public key and the attesting certificate chain are sent to the server from which the key is to be imported. The server encrypts the key using the received public key. On the device, the wrapped key is unwrapped within the TEE. Android 9 also introduced support for storing keys in a hardware security module which has its own dedicated CPU and storage, unlike a TEE which shares CPU and storage with the untrusted

Android platform.

2.6.3 Biometric Authentication

Android 6 introduced support for fingerprint sensors. Google mandated the use of TEEs for storing and matching fingerprints. Along with the changes to Keystore, the support for fingerprint matching allowed an app to require authentication using fingerprints for usage of cryptographic keys. Fingerprint authentication is much faster and more convenient for a user compared to entering PINs/passwords.

2.6.4 Protected Confirmation

Android 9 introduced an optional feature called *Protected Confirmation* [54]. Protected Confirmation allows a remote server to securely obtain the user's consent for some action that it intends to perform for the user, e.g. executing a money transfer from the user's bank account. Protected Confirmation builds upon the Key Attestation feature introduced in Android 7. To make use of Protected Confirmation, an application needs to first enroll a public key with the server. The server also needs an attestation of the public key. After the server has received the public key and a valid attestation for it, the server can send any message for which it needs the user's confirmation. On the device, the app receives the message and requests the Protected Confirmation system to obtain the user's confirmation. The trusted application (TA) of the Protected Confirmation system takes control of the touchscreen to display the message to the user and if the user confirms the message, it returns a confirmation token which is an HMAC-SHA256 value computed over the message using a 256 bit secret key that is shared with the Keystore system. After receiving the confirmation token, the app sends the message and the token to the Keystore system which verifies the token using the shared secret key. If verification is successful, Keystore signs the message using the private key corresponding to the public key that was enrolled with the server. This signature can be sent to the server. The server verifies the signature using the enrolled public key and if the verification is successful the server is assured that the user's consent was obtained securely. Because only the TEE has access to the private key used for signing, there is no way a compromised Android platform can spoof the user's confirmation. At the time of writing, Protected Confirmation has only been made

available in Google's Pixel 3 and newer Pixel devices.

2.7 Payment Security Standards

This section describes two payment security standards that are relevant to our work.

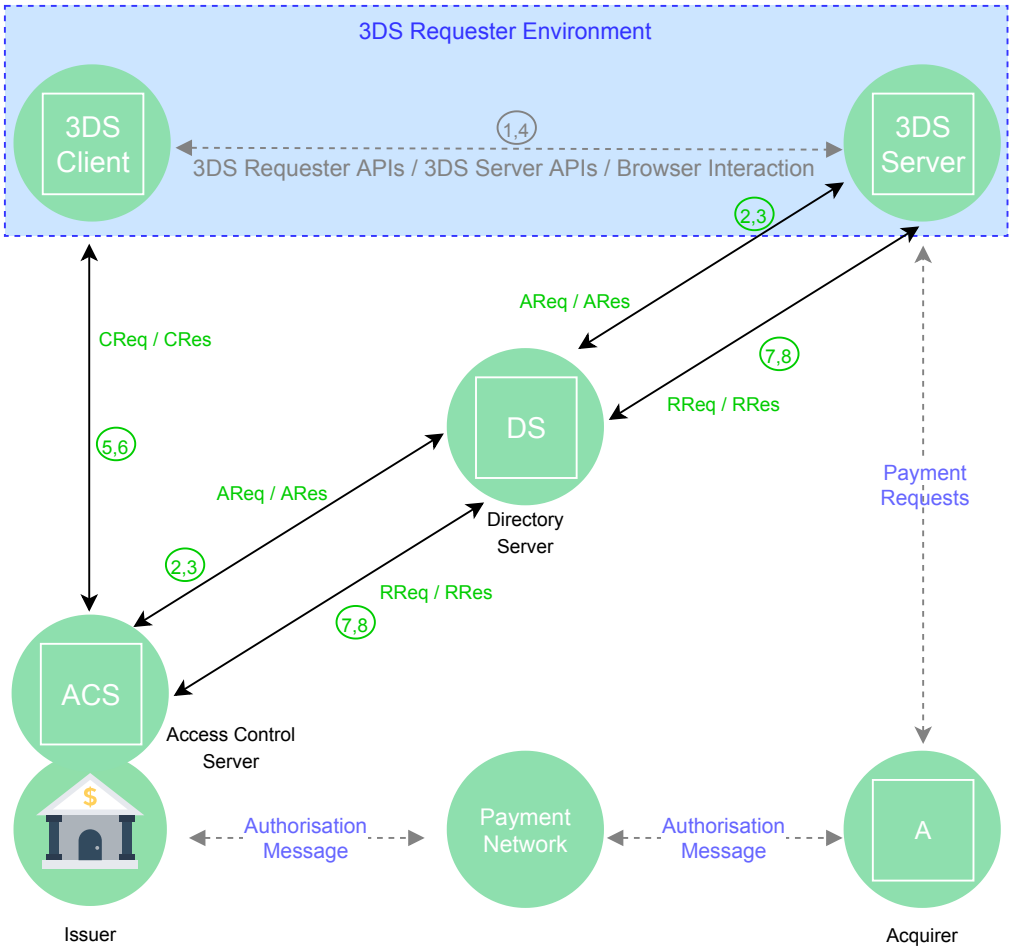

Figure 2.4: 3-D Secure Domains and Components [3]

2.7.1 EMV 3-D Secure 2.0

3-D Secure 2.0 is an authentication protocol created by EMVCo [17] for e-commerce transactions. It is based on a three-domain model. The three domains are

- Issuer Domain

- Acquirer Domain
- Interoperability Domain

Figure 2.4 shows the interaction between the three domains as specified in version 2.0 of the protocol [3]. The dashed lines in the figure show interactions that are not specified by the protocol, but are an essential part of a complete payment system. The Issuer domain has the Cardholder themself, the Cardholder's device used for payments, the Issuer bank which issued a card to the Cardholder and the Issuer's Access Control Server (ACS). The Acquirer domain has the 3DS Requestor (usually an e-commerce merchant), the 3DS Server deployed by the 3DS Requestor and the Acquirer bank with which the 3DS Requestor has an account. The Interoperability domain connects the other two domains. The Interoperability domain has a Directory Server (DS) usually run by payment networks such as VISA, Mastercard, etc. A 3-D Secure authentication flow is initiated with an *Authentication Request (AReq)* message created by the 3DS Server using information received from the 3DS Client ①, which is either an app that embeds an EMVCo-approved implementation of 3DS SDK specification [5] or a browser. ② The AReq message is sent to the ACS via the DS. Based on the information contained in the AReq message such as Cardholder identity, payment amount, etc., the ACS determines whether a *Frictionless flow* or a *Challenge flow* is required to authorize the transaction. Frictionless flow is one of the highlights of 3DS 2.0 vis-à-vis 3DS 1.0. In Frictionless flow, the transaction is immediately authorized on receiving the AReq message without any further interaction with the Cardholder. Challenge flow involves further interaction with the Cardholder to obtain some kind of credential (e.g. PIN, answer to questions). ③ The ACS communicates its decision to the 3DS Server by sending an *Authentication Response (ARes)* message via the DS. If a Challenge flow is required, then the 3DS Server sends the ACS's URL and other necessary data to the 3DS Client ④. A *Challenge Request (CReq)* message is created using information contained in ARes and sent to the ACS directly ⑤. The ACS responds with a *Challenge Response (CRes)* message ⑥. Depending on the requirements of the ACS there will be one or more exchanges of CReq-CRes pairs between the 3DS Client and the ACS. Finally, to complete the Challenge flow, the ACS sends a *Results Request (RReq)* message to the 3DS Server via the DS ⑦

and the 3DS Server responds to it with a *Results Response (RRes)* message ⑧ . 3DS 2.0 has a provision for *Decoupled Authentication* which does not utilise CReq and CRes messages for the Challenge flow and happens outside the 3-D Secure protocol. But RReq and RRes messages are sent even in the case of Decoupled Authentication.

2.7.2 PCI Data Security Standard

The Payment Card Industry Data Security Standard [52] was first published in 2004 and has since been updated multiple times with the latest release (3.2.1) in 2018. All entities involved in the storing, processing & transmission of payment card information - merchants, acquirers, issuers, payment networks, etc. are required to comply with PCI DSS. The compliance is enforced by the PCI Security Standards Council [81]. PCI DSS requirements are grouped into the following 6 goals.

1. Build and Maintain a Secure Network and Systems
2. Protect Cardholder Data
3. Maintain a Vulnerability Management Program
4. Implement Strong Access Control Measures
5. Regularly Monitor and Test Networks
6. Maintain an Information Security Policy

Out of the above 6 goals, only the 2nd one is relevant to our work. The requirements under this goal are as follows.

1. **Protect stored cardholder data** - Cardholder data should never be stored unencrypted. It should not be stored longer than necessary. Authentication data such as CVV numbers should not be stored at all, not even in encrypted form. Only a few authorized persons with a genuine business need should be able to see more than the first six and last four digits of the PAN (Primary Account Number).

2. **Encrypt transmission of cardholder data across open, public networks** - Strong cryptography should be used while transmitting PAN, authentication codes over public networks. Messaging systems auch as SMS, instant messaging apps, email, etc. should never be used to transmit unencrypted PANs.

Chapter 3

Design

3.1 Threat Model

The following entities are trusted in our design:

- The TEE (ARM TrustZone), the trusted OS (OP-TEE in our implementation), all the trusted apps running on top of it and all the data stored in TEE-protected secure storage
- The OEM which manufactured the smartphone and its supply chain
- The digital wallet service provider and the servers deployed by it
- The following entities involved in executing a transaction on the user's payment card:
  - The Issuer bank which issued the card and its Access Control Server
  - The card network (e.g. VISA, Mastercard, etc) to which the user's card belongs and its Directory Server
- The user of the digital wallet

We assume that the smartphone has verified boot feature that ensures the integrity of the code running in the secure world. The adversary is assumed to have complete control over the untrusted OS kernel and all the apps and services running on top of it (including the normal world app for the digital wallet).

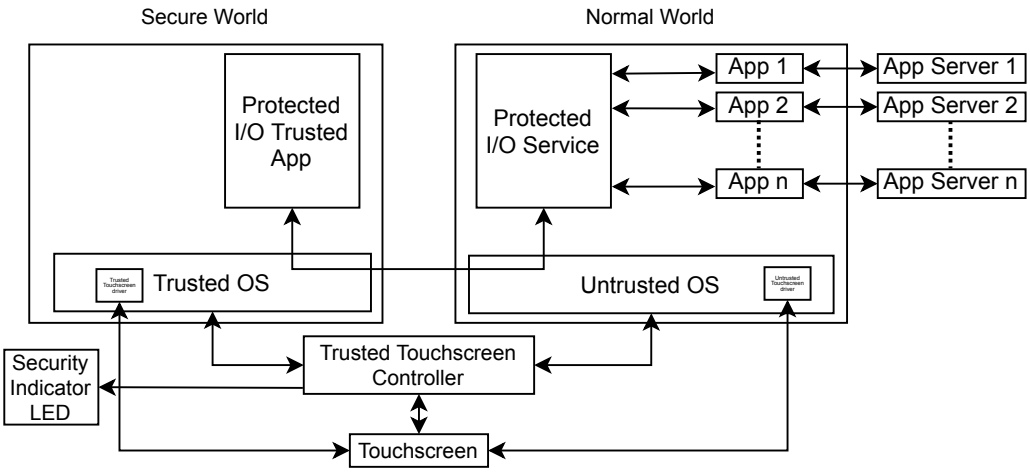

Figure 3.1: Protected I/O Framework

The aim of the design is to prevent an attacker from executing fraudulent transactions with the help of a compromised normal world. A transaction is fraudulent if the user does not give informed consent to it. The design cannot prevent (or even detect) DOS (Denial of Service) attacks. Since all communication between the application server and the secure world goes through the normal world, it is trivial for a compromised normal world to stop this communication thus making transactions impossible. Side channel attacks, exploitation of vulnerable implementations of standard cryptography algorithms and sophisticated hardware attacks on the TEE are outside the scope of this work.

3.2 Protected I/O

This section describes the various components involved in Protected I/O, the API exposed by the TEE to the normal world and the flow of messages between the Application Server and the TEE. Subsequent sections describe how Protected I/O can be used to secure payment transactions.

3.2.1 Protected I/O Framework

Figure 3.1 shows the various components involved in Protected I/O. The *Trusted OS* runs within the secure world i.e. the TEE. It is supposed to be very lightweight — both in terms of the amount of code and the memory usage — with a minimal

set of features. It is more privileged than the *Untrusted OS*. It has access to all of the RAM on the device, some of which is only accessible to it and not the Untrusted OS. It has a *trusted touchscreen driver* that is used in secure entry of confidential user input and secure display of confidential output. In our implementation, OP-TEE is used as the Trusted OS. The Untrusted OS runs within the normal world. As stated in section 3.1, the Untrusted OS is assumed to be under the control of the adversary. Therefore adversary has access to all the input received from the user via the *untrusted touchscreen driver* and the ability to show misleading output to the user to get confidential information and/or to execute fraudulent transactions. In our implementation, Android is the Untrusted OS. There is a *Protected I/O Service* that runs in the normal world and talks to the *Protected I/O Trusted App* running in the secure world via the two OSes. The normal world apps request the services of the Protected I/O TA via the Protected I/O Service. The Protected I/O TA has access to TEE-protected *Secure Storage*. The Secure Storage contains a set of root CA certificates for verifying the publick key certificates of *App Server*s. Additionally, the Secure Storage has chain of certificates loaded into the device at the factory by the OEM, that is used to attest the security of any public key generated by Protected I/O TA. App Servers that use the public keys generated by Protected I/O TA, need to trust the root certificate of the certificate chain. Hence the root certificate needs to be from a CA that is very well known and trusted by all the service providers whose services the user might need. For Android devices, Google has mandated the use of Google Hardware Attestation Root certificate in order for the device to be certified for Google Play services [21]. Secure Storage is also used to store the private key of every asymmetric key pair generated by the Protected I/O TA. The device has a *Trusted Touchscreen Controller* which multiplexes the device's touchscreen between the Trusted and Untrusted OSes. The Trusted OS can use it to wrest control of the touchscreen from a potentially compromised Untrusted OS, or the Untrusted OS can voluntarily yield control to it. On a device with ARM TrustZone, TrustZone Protection Controller (TZPC) [70] can be used to perform the function of Trusted Touchscreen Controller. The Trusted Touchscreen Controller ensures that when (and only when) the touchscreen is under the control of the Trusted OS, the *Security Indicator LED* is lit. The LED gives the user an

assurance that any information he enters while it is lit, is protected by the TEE. Since the Untrusted OS has no control over the LED, any attempt by the adversary to mislead the user with an imitation of the user interface of Protected I/O TA will be thwarted by an attentive user.

3.2.2 Protected I/O API

This section describes the API used by the Protected I/O Service (PIO Service) to request the services of the Protected I/O TA (PIO TA). Details of encryption and signing of messages are described in the next section. PIO TA and PIO Service communicate using shared memory. The data exchanged between them is formatted according to the Concise Binary Object Representation (CBOR) [26] specification for serialization of data. Appendix A shows various data types used in the exchange of data between PIO Service and PIO TA expressed using the Concise Data Definition Language (RFC 8610) [27]. We now describe the API calls exposed by the PIO TA alongwith details of input and output parameters for each of them.

For the data types written in `monospace` please refer to Appendix A.

- **`GENERATE_KEY_PAIR_FOR_SERVER`**

  Input: `{server_info, key_params}`

  Output: `{error_code, pub_key}`

  Given an App Server's Common Name and public key certificate chain and desired key parameters, this method first checks if the certificate chain is trusted and anchored by one of the root certificates available in Secure Storage. If so, it generates a new pair of elliptic curve keys and returns the public key in the output. The private key, the Server's Common Name and the Server's public key (extracted from the certificate chain) are stored together in Secure Storage. If the curve or the key size is not supported or if the certificate chain is not trusted, the method returns the appropriate `error_code` indicating reason for failure.

- **`GET_PUBLIC_KEY_FOR_SERVER`**

  Input: `{common_name}`

Output: `{error_code, pub_key}`

If a key pair has been generated for the server identified by the Common Name, the public key is returned as output, else an error is returned.

- **CHANGE_PUBLIC_KEY_FOR_SERVER**

  Input: `{server_info}`

  Output: `{error_code}`

  This method can be used to change the saved public key for an App Server.

- **DELETE_KEY_PAIR_FOR_SERVER**

  Input: `{common_name}`

  Output: `{error_code}`

  Given an App Server's Common Name, this method deletes the generated keypair as well as the saved public key for the App Server. It asks the user for confirmation before deleting.

- **GET_ATTESTATION_FOR_SERVER**

  Input: `{common_name, attestation_challenge}`

  Output: `{error_code, cert_chain}`

  This method returns a public key certificate chain (with one or more certificates) attesting the key pair (or more specifically the public key) generated for the App Server identified by `common_name`. This certificate chain acts as a proof of the fact that the private key of key pair for the Server is stored in Secure Storage. The certificate chain can be sent to the App Server which will validate it to decide whether or not to use the generated public key for Protected I/O. The first certificate in the chain (the one with the generated public key) contains `attestation_challenge` in an extension (RFC 5280 § 4.2 [57]). This API is similar to the *attestKey* API specified in the Keymaster 3 HIDL specification [20]. *attestKey* expects the key and a list of characteristics (e.g. whether the key requires user authentication) to attest as input from the normal world. But this API just expects the Server's Common Name and the attestation challenge.

- **GET_SIGNED_INPUT_FOR_SERVER**

  Input: `{common_name, signed_message_from_server}`

  Output: `{error_code, signed_message_for_server}`

  This method expects an `input_form` in the `data` field of `signed_message_from_server` and the `is_confidential` boolean in it to be set to false. The method first checks whether a key pair exists for the Server identified by `common_name`. If not, it returns an error. Otherwise, it verifies the signature in `signed_message_from_server` using the Server's public key that was stored during key pair generation. If the signature is valid, the TA takes control of the touchscreen and displays the input form to the user along with Submit and Cancel buttons and the Server's Common Name (to prevent phishing). After the user enters the requested information and taps Submit, an instance of `filled_input_form` is created and put in the `data` field of `signed_message_for_server`, which is returned as output. The nonce from `signed_message_from_server` instance is copied into the `signed_message_for_server` instance. This method is meant for receiving non-confidential input from the user.

- **GET_SECRET_INPUT_FOR_SERVER**

  Input: `{common_name, signed_message_from_server}`

  Output: `{error_code, encrypted_message_for_server}`

  This method expects an `input_form` in the `data` field of `signed_message_from_server` and the `is_confidential` boolean in it to be set to true. The method first checks whether a key pair exists for the Server identified by `common_name`. If not, it returns an error. Otherwise, it verifies the signature in `signed_message_from_server` using the Server's public key that was stored during key pair generation. If the signature is valid, the TA takes control of the touchscreen and displays the input form to the user along with Submit and Cancel buttons and the Server's Common Name (to prevent phishing). After the user enters the requested information and taps Submit, an instance of `filled_input_form` is created, encrypted and put in the `encrypted_data` field of `encrypted_message_for_server`,

which is returned as output. The nonce from `signed_message_from_server` instance is copied into the `encrypted_message_for_server` instance.

- **DISPLAY_MESSAGE_FROM_SERVER**

  Input: `{common_name, signed_message_from_server}`

  Output: `{error_code, signed_message_for_server}`

  This method expects a UTF-8 encoded string in the `data` field of `signed_message_from_server`. The signature is verified and if it is valid, the message is displayed to the user with an OK button and the Server's Common Name. When the user taps the OK button, the same message is signed with the private key generated for the Server identified by `common_name`, included in the `data` field of `signed_message_for_server` and returned as output. The nonce from `signed_message_from_server` instance is copied into the `signed_message_for_server` instance. The output can optionally be sent to the Server by the App to prove that the message was indeed shown to the user.

- **DISPLAY_SECRET_MESSAGE_FROM_SERVER**

  Input: `{common_name, encrypted_message_from_server}`

  Output: `{error_code, encrypted_message_for_server}`

  This method expects an encrypted UTF-8 encoded string in the `encrypted_data` field of `encrypted_message_from_server`. The string is decrypted and displayed to the user with an OK button and the Server's Common Name. When the user taps OK button, the same string is encrypted again for the Server and written to the `encrypted_data` field of an `encrypted_message_for_server` instance which is returned as output. The nonce is copied from the `encrypted_message_from_server` instance to the `encrypted_message_for_server` instance. The output can optionally be sent to the Server by the App to prove that the secret message was indeed shown to the user.

- **GET_USER_CONFIRM_FOR_SERVER**

Input: `{common_name, signed_message_from_server}`

Output: `{error_code, signed_message_for_server}`

This method expects a UTF-8 encoded string in the `data` field of `signed_message_from_server`. The signature is verified and the string is displayed to the user along with the Server's Common Name, a Confirm and a Deny button. Only if the user taps the Confirm button, the string is copied into the `data` field of a `signed_message_for_server` instance which is returned as output. The nonce is copied from the `signed_message_from_server` instance to the `signed_message_for_server` instance. If the user taps the Deny button, an error code indicating that is returned. This method can be used to securely acquire the user's consent for some action (e.g. a transaction on the user's bank account). It is slightly different from Android's Protected Confirmation because it requires the message received from the server to be signed.

- **`DISPLAY_PUBLIC_KEY_FOR_SERVER`**

  Input: `{common_name}`

  Output: `{error_code}`

  If a key pair has been generated for the Server identified by `common_name`, then PIO TA asks the User to confirm whether he wishes to display the public key on the screen and on obtaining confirmation it takes control of the touchscreen and displays the public key as a JAB Code [36] along with a Close button. When user presses Close button, the method returns. If no key pair exists, an error is returned.

- **`ABORT`**

  Input: No input required

  Output: `{error_code}`

  This method can be used by the normal world to abort any ongoing operation of the Protected I/O TA that is waiting for user input. The need for aborting may arise when the normal world urgently needs control of the touchscreen,

e.g. to allow the user to receive a phone call. For reasons explained in Section 3.3.1, the `DISPLAY_PUBLIC_KEY_FOR_SERVER` call cannot be aborted.

Except for the `DISPLAY_PUBLIC_KEY_FOR_SERVER` method, all methods mentioned above have been implemented by us in a Trusted App. Out of the six form field types proposed by us, our Trusted UI implementation supports three - *text*, *password* and *integer*.

3.2.3 Encryption and Signing

In our design, Elliptic Curve cryptography is used for encryption and signing of data exchanged between PIO TA and App Server. Compared to non-EC cryptography, EC cryptography requires much smaller keys for equivalent security. Large keys cannot be displayed as JAB codes in the limited screen space available in smartphones. Therefore, it is necessary to use EC cryptography. PIO TA generates and stores an asymmetric key pair ($AppPrivKey_{PIO}$, $AppPubKey_{PIO}$) for each App Server that it communicates with. PIO TA gets the App Server's public key $ServerPubKey_{PIO}$ from its certificate chain. The App fetches public key $AppPubKey_{PIO}$ and sends it to the App Server before any confidential data is exchanged. $AppPubKey_{PIO}$ & $ServerPubKey_{PIO}$ cannot be directly used for encrypting messages of arbitrary length. So a hybrid approach involving both symmetric and asymmetric cryptography is used in our design.

Whenever PIO TA needs to encrypt data for an App Server, it first generates an ephemeral key pair ($AppPrivKeyEphe_{PIO}$, $AppPubKeyEphe_{PIO}$). Then the Elliptic Curve Diffie Hellman key exchange protocol is executed to derive a shared secret $EpheSharedSecret$ using $AppPubKeyEphe_{PIO}$ and $ServerPubKey_{PIO}$. The SHA-256 hash of $EpheSharedSecret$ gives a 256-bit symmetric key $EpheSymKey$ that is used for AES-256 encryption of messages of arbitrary length. The encrypted message and $AppPubKeyEphe_{PIO}$ are sent to the App Server. The Server can derive $EpheSharedSecret$ by executing the ECDH protocol on $AppPubKeyEphe_{PIO}$ and $ServerPubKey_{PIO}$ and from that it can derive the key $EpheSymKey$ for decryption. When the Server needs to encrypt data for PIO TA, the process is similar except this time the ephemeral key pair is generated by the Server. All messages exchanged between PIO TA and the App Server are signed by the sender using the

Elliptic Curve Digital Signature Algorithm (ECDSA). For signing, PIO TA uses $AppPrivKey_{PIO}$ and the Server uses $ServerPrivKey_{PIO}$.

3.2.4 Protected I/O Protocol

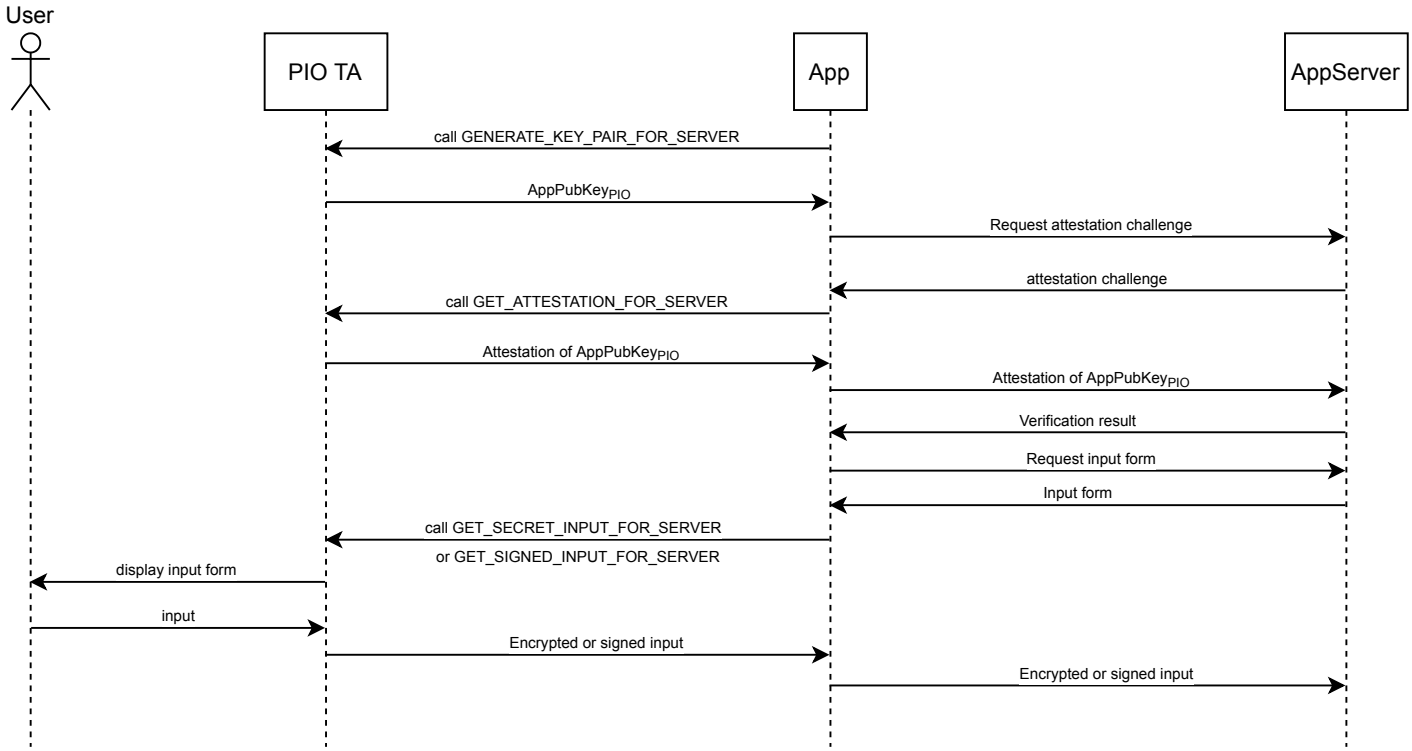

Figure 3.2: Protected I/O Protocol

To receive confidential input from a user, the App Server first needs the public key $AppPubKey_{PIO}$ and also needs to ensure that the corresponding private key $AppPrivKey_{PIO}$ is held in TEE-protected secure storage. The App running in the normal world first requests the Protected I/O TA to generate a key pair for the App Server using the `GENERATE_KEY_PAIR_FOR_SERVER` API call (Figure 3.2). The App next fetches the attestation challenge from the App Server and then uses the challenge received to call `GET_ATTESTATION_FOR_SERVER` for obtaining an attestation for $AppPubKey_{PIO}$ from the TA. Since the TA stored the Server's Common Name during key generation, it knows which key to attest. On obtaining the attes-

tation, the App sends it to the App Server. The App Server verifies the certificate chain and if found valid, extracts $AppPubKey_{PIO}$ from it and associates it with its current session with the App. Subsequently, in this session, all Protected I/O related data exchanged between PIO TA and the Server needs to be signed by the sender using its private key. When the session expires due to inactivity or other reasons, in a new session, key attestation needs to be repeated. With the public key $AppPubKey_{PIO}$ enrolled, the Server can start receiving confidential input from the user. Next, the App requests an `input_form` from the Server. Depending on requirements, the Server sets the `is_confidential` field in the form to true or false. On receiving the form, the App checks the value of `is_confidential` and either calls `GET_SIGNED_INPUT_FOR_SERVER` for non-confidential input or `GET_SECRET_INPUT_FOR_SERVER` for confidential input. A compromised normal world might call `GET_SIGNED_INPUT_FOR_SERVER` even if `is_confidential` is set to true, to try and obtain the User's confidential input in plaintext. Therefore, the PIO TA needs to validate the value of `is_confidential`. The `filled_form` received from PIO TA is sent to the Server. If required, the Server can respond with a signed message acknowledging the receipt of the `filled_form` which the App can show to the user by calling `DISPLAY_MESSAGE_FROM_SERVER`. The nonce included in the `input_form` is used for prevention of replay attacks. A compromised normal world can make copies of signed messages from the PIO TA and try to resend them without the User's consent. For example, a login request with confidential credentials may be recorded and replayed. The Server remembers the nonce included in `input_form` and if the `filled_form` does not contain the same nonce, it is rejected.

3.3 Secure Payments

Section 3.2 described how any app can make use of the Protected I/O facility to receive authenticated and confidential inputs from a user. This section shows how digital payments can be made more secure by leveraging our Protected I/O design. In our design, three Apps and their corresponding App Servers are involved in securing digital payments.

- *WalletApp*: A digital wallet app that allows a user to load digital cash

in it using Net Banking, Card payments, etc. and use this cash to pay various Merchants or other users of the app. WalletApp communicates with *WalletServer*. Paytm is an example of a digital wallet app.

- *BankApp*: An app published by the wallet user's bank. It is required for authorizing Net Banking and Card transactions. BankApp communicates with *BankServer*.

- *KYCApp*: An app published by an organization in charge of issuing unique IDs that are allowed to be used for KYC (Know Your Customer) process in the jurisdiction the user resides in. The KYCApp communicates with *KYCServer*. For example, in India, Aadhaar numbers issued by Unique Identification Authority of India (UIDAI) [72] can be used for KYC. The mAadhaar app [78], if modified to conform to our design of Protected I/O, can fulfill the role of KYCApp.

The following subsections describe how these three Apps work together to secure digital payments.

3.3.1 KYC Process

In most jurisdictions across the world, digital payment service providers are required to comply with regulations mandating a KYC (Know Your Customer) process for their users. In India, the *Reserve Bank of India* has the authority to issue rules for KYC [55]. The European Union, under its *Single Euro Payments Area* [59] initiative, issued the *Payment Services Directive* [53] for governing payment service providers. Even when not required by regulations, it can be very fruitful for a payment service provider to conduct a KYC process for associating a real identity with each of its user accounts. This association of an identity with a user account can be used to let users recover access to their accounts after they have lost access due to forgotten credentials, stolen devices, etc. Currently, a very common way of ensuring this association is to have the user enter an OTP (One Time Password) sent via SMS to a phone number registered at the time of issuing the unique ID (e.g. Aadhaar number in India) allowed for KYC. The phone number acts as a proxy for the unique ID and submission of the OTP proves that

the user owns the phone number. But SMS is accessible to the normal world and therefore to an attacker who controls the normal world. So a KYC process that relies on SMS-delivered OTP is not secure against a compromised normal world. An attacker can impersonate the owner of the phone number by stealing OTPs. In our design, the Protected I/O API is used for a more secure delivery of OTPs. To enable this secure delivery of OTPs the user first needs to enroll a public key $PubKey_{ID}$. The entities involved in the process are:

- Protected I/O Trusted App (**PIO TA**)
- **KYCApp** mentioned earlier
- An Enrolment Device (**EDev**) at an Enrolment Centre run by the organization that issues unique IDs to users (e.g. UIDAI)
- The **KYCServer** mentioned earlier, which can communicate with *KYCApp* as well as *EDev*

The process of registration is as follows (Figure 3.3). The User visits the nearest Enrolment Centre with the ID document issued to him and his smartphone which has the *KYCApp* installed in it. The User enters his unique ID number in *EDev*. *EDev* prompts User to present biometrics such as fingerprints, iris, etc. that were enrolled at the time of issuing the unique ID. The User presents required biometrics to *EDev*. The ID number along with the captured biometrics are sent to *KYCServer* for verification. *KYCServer* verifies the ID number and biometrics and sends the result to *EDev*. If verification is successful, *EDev* prompts user to enroll his public key. Using the *KYCApp*, User initiates generation of key pair ($PubKey_{ID}$, $PrivKey_{ID}$). *KYCApp* uses the API method `GENERATE_NEW_KEY_PAIR_FOR_SERVER` to request the creation of the key pair as described in Section 3.2.4. *PIO TA* generates key pair ($PubKey_{ID}$, $PrivKey_{ID}$) and *KYCApp* receives $PubKey_{ID}$ as response. *KYCApp* next sends a request for displaying $PubKey_{ID}$ to *PIO TA* using the API method `DISPLAY_PUBLIC_KEY_FOR_SERVER`. After obtaining User's confirmation *PIO TA* securely displays $PubKey_{ID}$ as a JAB Code [36] and enables the security indicator

LED. User verifies that the security indicator LED is on and presents his smartphone to *EDev* for scanning of $PubKey_{ID}$ off the screen. *EDev* captures $PubKey_{ID}$ off the screen, validates it according to expected elliptic curve and number of bits, and sends it to *KYCServer*. If, unknown to the User, the normal world is compromised, it may attempt to abort the `DISPLAY_PUBLIC_KEY_FOR_SERVER` call using the `ABORT` call and display a different public key on the screen just before the scanning happens and the User might fail to notice the change. To prevent this attack, we have disallowed aborting of public key display in our design. *KYCServer* stores $PubKey_{ID}$ in its database with the User's unique ID, generates an OTP and sends it to *EDev*. *EDev* prompts user to fetch OTP from the server and enter it. User initiates request for OTP using *KYCApp*. *KYCApp* sends User's unique ID to *KYCServer*. *KYCServer* sends the OTP generated previously, encrypted with $PubKey_{ID}$ as response to *KYCApp*. *KYCApp* asks *PIO TA* to display the decrypted OTP to the User using the `DISPLAY_SECRET_MESSAGE_FROM_SERVER` method. *PIO TA* displays the decrypted OTP to the User. User enters OTP in *EDev*. *EDev* checks if the entered OTP matches OTP received from *KYCServer* and sends the result to *KYCServer*. If the OTPs match, it is established that the User possesses $PrivKey_{ID}$ corresponding to $PubKey_{ID}$ because without it the decryption of the OTP would have been impossible. *KYCServer* relies on the User to ensure that only a public key whose corresponding private key is securely stored is enrolled, and therefore *KYCApp* does not request attestation of $PubKey_{ID}$. The User ensures this by checking the security indicator LED before letting *EDev* scan the public key. *KYCServer* next generates a deregistration code $DeRegCode_{ID}$, saves it in its database with the User's unique ID and sends it to *EDev*. *EDev* informs user that his public key is registered and generates a printout with $DeRegCode_{ID}$ on it.

The User is expected to keep $DeRegCode_{ID}$ safe. If and when the User's smartphone is stolen, $DeRegCode_{ID}$ can be used to authorize deregistration of the registered public key $PubKey_{ID}$ to prevent abuse. The organization handling the registration of public keys can deploy a website and/or an IVRS (Interactive Voice Response System) to facilitate deregistration. For registering a new public key, the User needs to revisit an Enrolment centre.

Once $PubKey_{ID}$ is enrolled with the *KYCServer*, any service provider that needs

to verify the identity of the User can request the User for his unique ID and then use the ID to request *KYCServer* to securely deliver an OTP generated by the provider to the User. After that, using its app installed on the User's smartphone, the service provider can request User for that OTP. Protected I/O can be used to ensure that an attacker is unable to change the ID entered by the User and also unable to steal the OTP.

3.3.2 Account Creation and Sign In

After installing *WalletApp* on his smartphone, the user needs to create an account with the payment service provider. Figure 3.4 shows the steps in the account creation process. When the app is run for the very first time, it requests generation of a key pair (*WalletAppPubKey*, *WalletAppPrivKey*) for *WalletServer* and enrolls it with the server as described in Section 3.2.4. After the public key is enrolled, *WalletApp* can request *WalletServer* for an `input_form`. For creation of a new account, the `input_form` needs at least three fields for username, password and the unique ID that will be used for the KYC process. The `is_confidential` boolean in `input_form` needs to be true because the password is supposed to be confidential. After the User fills and submits the form securely displayed by PIO TA, a `filled_form` that is encrypted and signed as described in Section 3.2.3, is sent to *WalletServer*. *WalletServer* validates the User's input (e.g. checking if the username is already taken). If validation fails, a signed error message is returned to *WalletApp* which can use the `DISPLAY_MESSAGE_FROM_SERVER` call to show it to the user. If validation succeeds, *WalletServer* sends an `input_form` in the response for getting an OTP from the user. This OTP is delivered to the user's smartphone with the help of *KYCServer* as described in Section 3.3.1. After receiving the correct OTP, *WalletServer* creates a new account for the user and associates the public key *WalletAppPubKey* & the verified identity with this account. *WalletAppPrivKey* (and by extension the smartphone in which it is securely stored) acts as a second factor (the "something you have" factor) for authentication. After account creation, whenever the user needs to sign into their account, *WalletApp* requests an `input_form` from *WalletServer* to be sent to PIO TA. When the user tries to sign in to the account from another device, then *WalletServer* can directly send an OTP to the device with *WalletAppPrivKey*

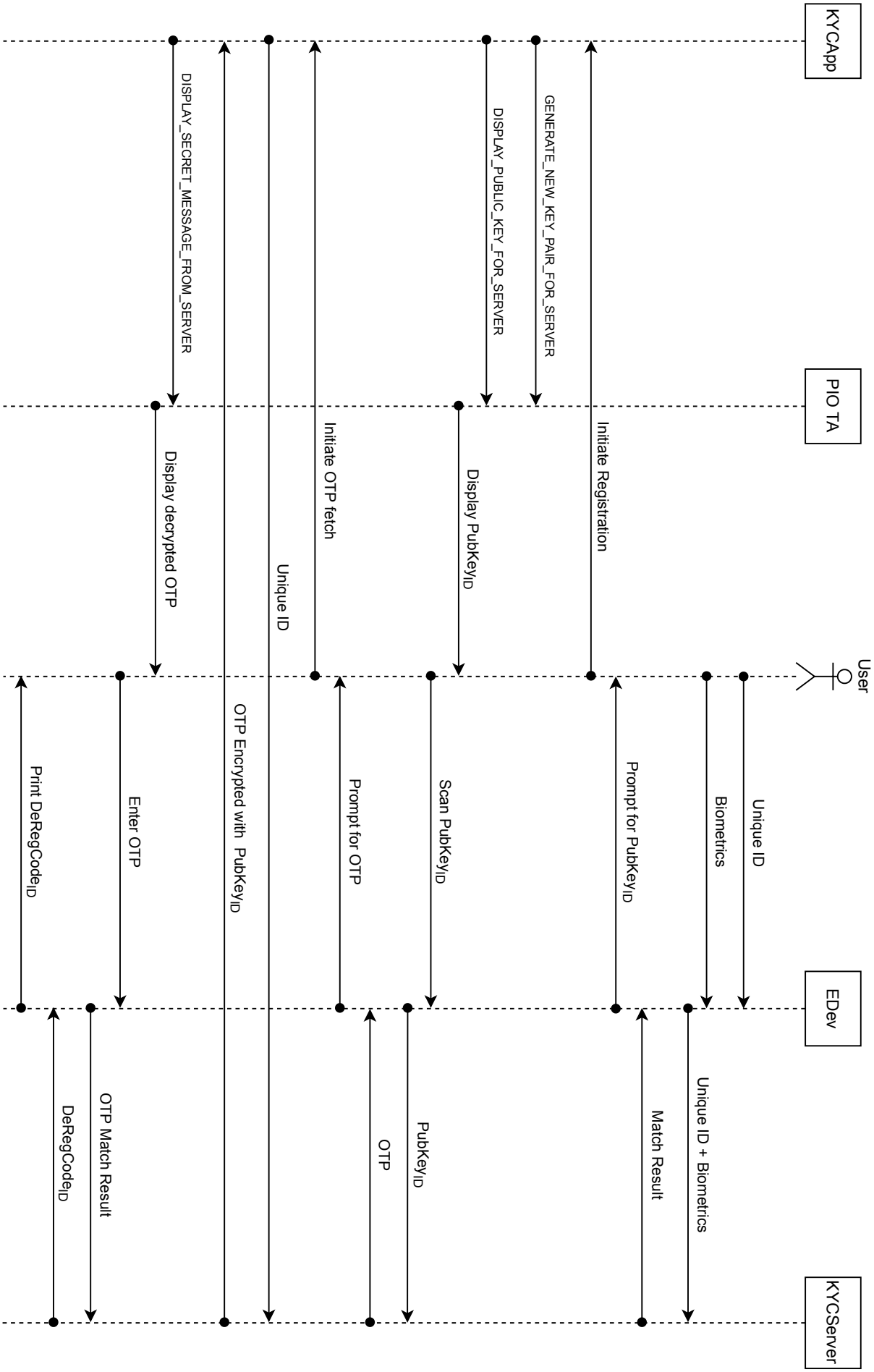

Figure 3.3: Registration of $\mathbf{PubKey_{ID}}$

for authentication. If this other device happens to be a smartphone capable of Protected I/O, then *WalletApp* on this device can enroll a new public key which can be authenticated by sending an OTP to the previous smartphone. Depending on the policies of the payment service provider, on the enrollment of the new public key, the older one may or may not be deregistered.

3.3.3 Adding Money to Wallet

Before the user can add money to his digital wallet, he needs to use *BankApp* to generate a key pair (*BankPubKey*, *BankPrivKey*) and enroll *BankPubKey* with the *BankServer*. The process of enrolling is exactly like the one described in Section 3.3.2 for *WalletAppPubKey*. After *BankPubKey* is enrolled, any withdrawal of money from the User's bank account can be authorized with the help of *BankApp*. The two most common ways of adding money to digital wallets are Net Banking and CNP (Card Not Present) transactions using credit/debit cards.

In case of Net Banking, for authorizing transactions, instead of showing the bank's web page in an embedded browser, as is currently done by wallet apps, the *WalletApp* can request the *BankServer* (via *WalletServer*) for a signed message describing the transaction and then use the `GET_USER_CONFIRM_FOR_SERVER` call to get the user to authorize the transaction. If *BankServer* determines that a PIN needs to be entered for authorizing the transaction, then it sends an `input_form` for the PIN with the transaction details in the `description` field and *WalletApp* calls `GET_SECRET_INPUT_FOR_SERVER` to fetch the PIN from the User. *WalletApp* needs to get the User's username for their bank account to request signed transaction details or signed `input_form` from the *BankServer*, otherwise *BankServer* would not know which public key to use for validating the `filled_form` or the signed user confirmation received as response from *WalletApp*. This process has been shown in Figure 3.5.

For authorization of CNP transactions, EMVCo [17] released version 2.0 of the 3-D Secure Protocol in 2016 [1, 2, 4]. Neither 3-D Secure 1.0, nor the newer 3-D Secure 2.0 deal with the threat model that our design is trying to deal with. The protocol specification does not take into consideration the possibility of the user's device being under the control of an attacker. It was not written to leverage a TEE for transaction security. To authenticate CNP transactions using Protected

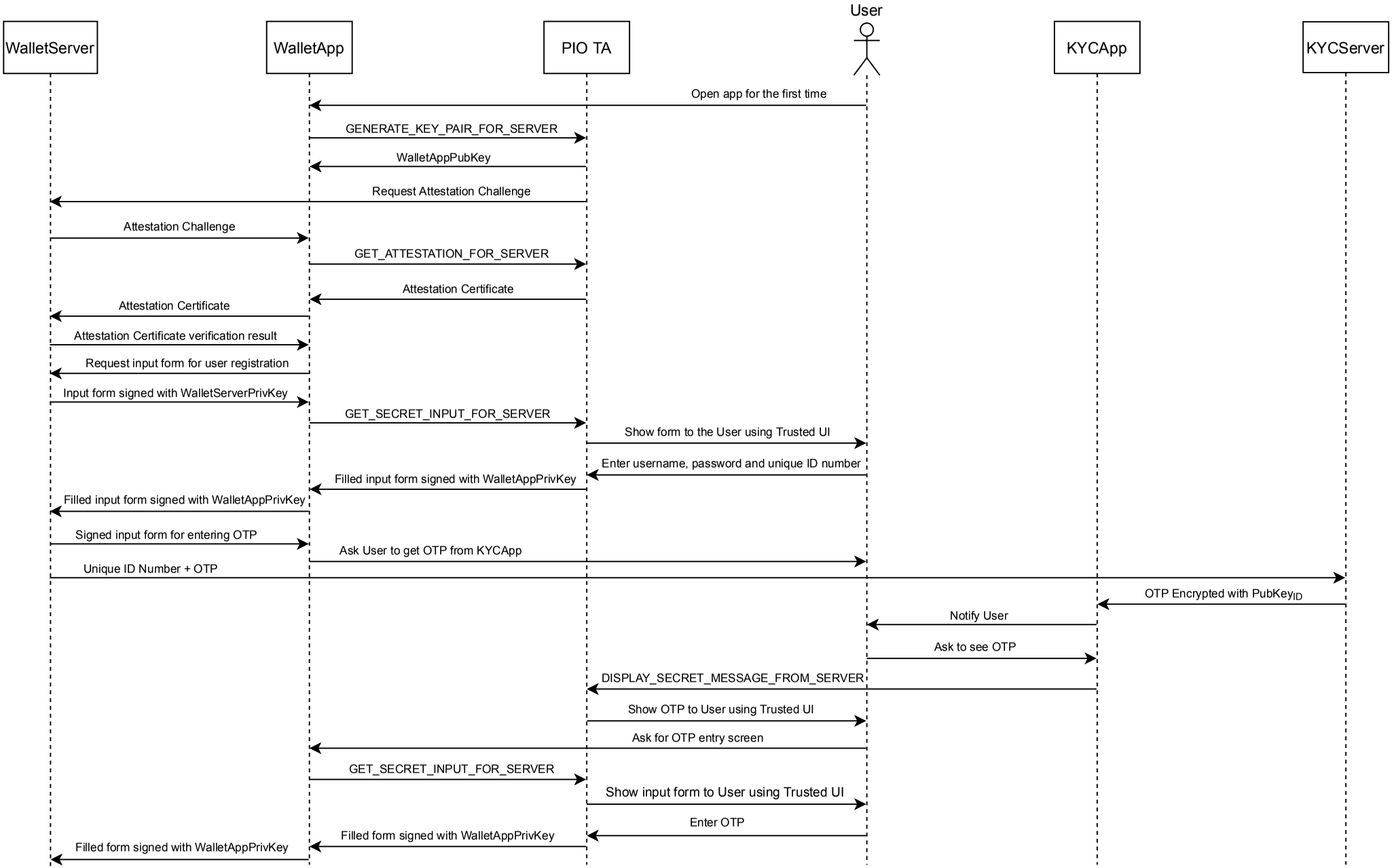

Figure 3.4: User Account Creation

I/O in the same way as Net Banking transactions, significant changes will need to be made to the 3-D Secure specification. Thankfully, 3-D Secure 2.0 allows Out-of-Band authentication for transactions. Therefore Protected I/O can be used to authenticate CNP transactions without making any changes to the 3-D Secure protocol. According to the terminology used in the specification, *WalletApp* is a 3DS Requestor App, *WalletServer* is a 3DS Server and *BankServer* is an ACS (Access Control Server). According to the protocol specification, to initiate a transaction, the 3DS Server needs to create an Authentication Request (*AReq*) message using information provided by the 3DS Requestor App and a 3DS SDK embedded within the App. The PAN (Permanent Account Number) of the user's credit/debit card is the most sensitive piece of information that 3DS Requestor App sends to 3DS Server. Since the specification does not specify the exact communication protocol between 3DS Requestor App and 3DS Server, we can use Protected I/O to securely send the PAN to the 3DS Server, thus preventing an attacker from stealing it. The ACS, instead of participating in a Elliptic-Curve Diffie Hellman key exchange with the 3DS SDK for in-band authentication, uses OOB authentication in which key pair (*BankPubKey*, *BankPrivKey*) and *BankApp* are used for Protected I/O. Just like in the case of Net Banking, either a signed message or an `input_form` is sent to *BankApp* by *BankServer*.

3.3.4 Payment Transaction

Once some money is loaded into the wallet, the User can make payments to merchants and other users of the same wallet provider. Usernames, phone numbers or QR Codes can be used for identifying the recipients when making a payment. For securing payments, it is necessary to prevent an attacker from modifying the recipient and/or the amount in the transaction. This is achieved with the use of `GET_USER_CONFIRM_FOR_SERVER` call. Entry of the recipient's identifier and the transaction amount can be handled by the untrusted *WalletApp*. The trusted *WalletServer* sends a signed message containing these details as a response. This signed message is used in the call to `GET_USER_CONFIRM_FOR_SERVER`. If a compromised *WalletApp* makes changes to the transaction details before sending them to *WalletServer*, these changes will be caught by the User when the signed message is securely displayed by PIO TA and the User can refrain from giving confirmation

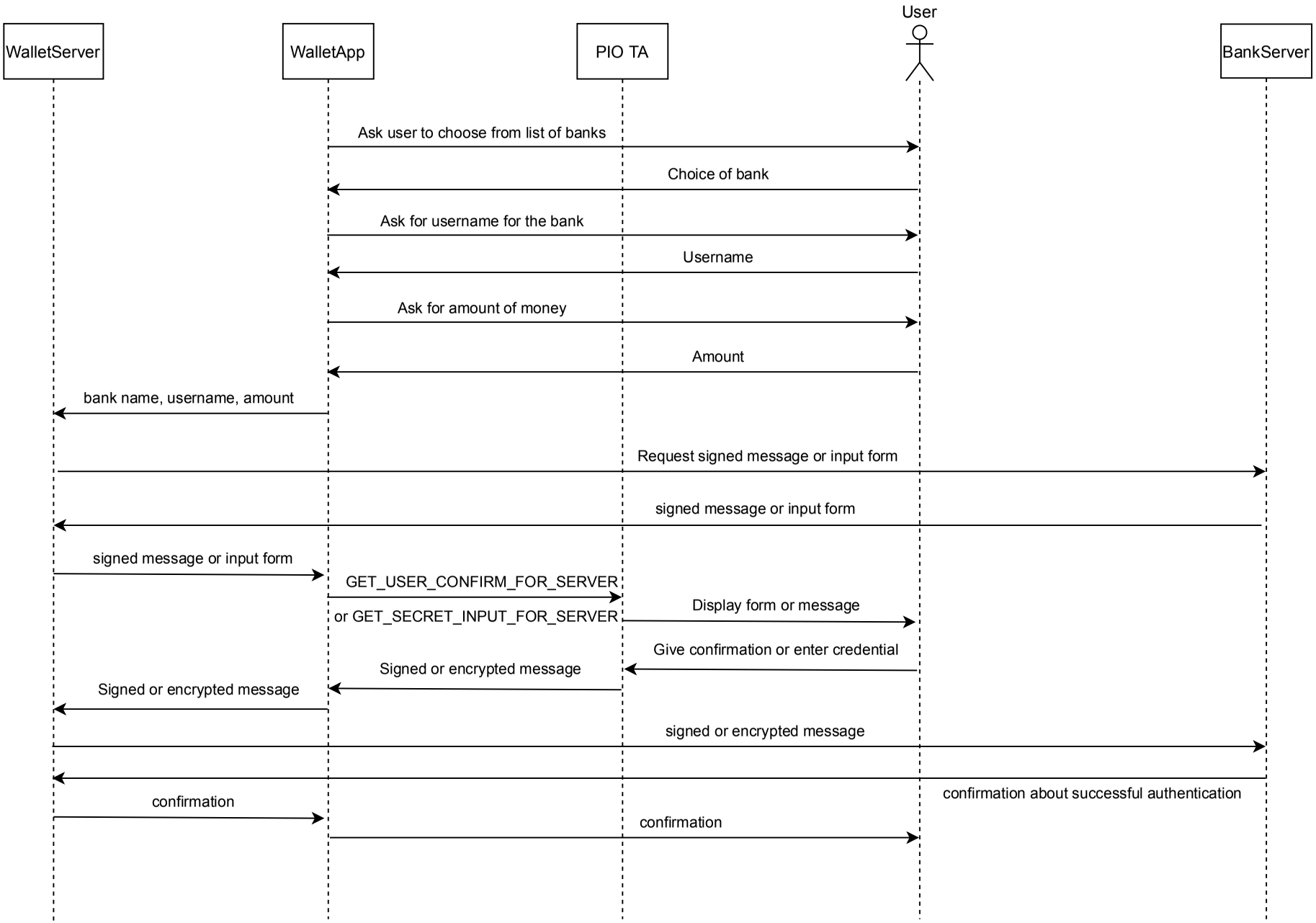

Figure 3.5: Adding Money to Wallet

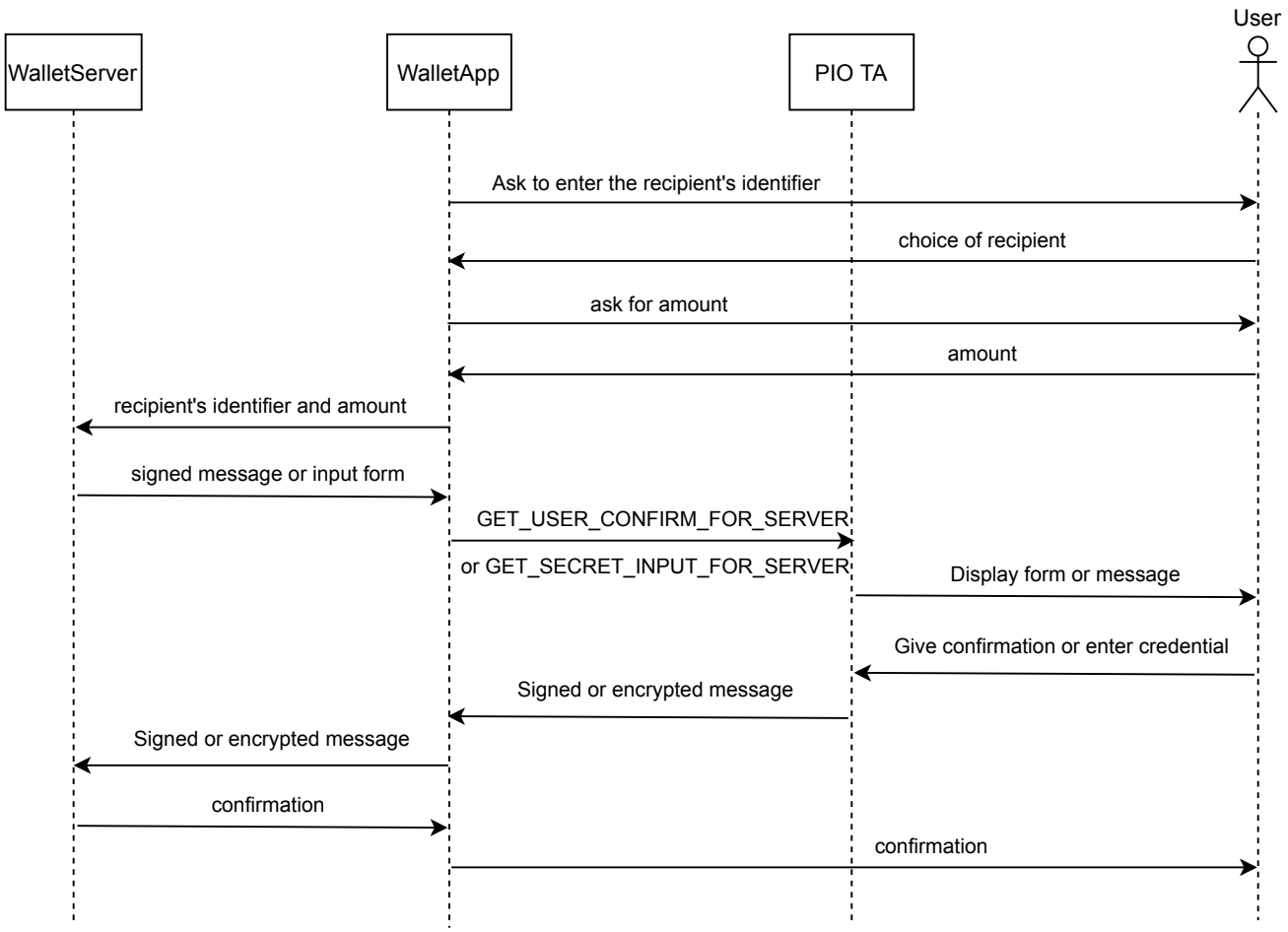

Figure 3.6: Transfering funds to another wallet user

for it. If the transaction requires authorization with a PIN, *WalletServer* can send a signed `input_form` with the transaction details in the `description` field, and `GET_SECRET_INPUT_FOR_SERVER` can be used to securely obtain the PIN from the User. On receiving the confirmation or the PIN from *WalletApp*, *WalletServer* validates it and if found valid, it executes the transaction. This process has been shown in Figure 3.6. If the User cancels the transaction, or if the PIN is incorrect or if the User has insufficient balance, *WalletServer* terminates the transaction. *WalletServer* sends a response to *WalletApp* about whether the transaction was executed or terminated. A compromised *WalletApp* can withhold information

about successful execution of the transaction from the User to trick them into unknowingly repeating the transaction. *WalletServer* can protect the User from being tricked like this by simply adding the User's current balance in the signed message mentioned above. When the User notices a reduction in their balance, they can infer that the previous transaction was executed successfully even though *WalletApp* did not inform them about it. If due to network issues, any message exchanged between the *WalletApp* and *WalletServer* is lost and the transaction is not completed, then, after an implementation defined timeout, *WalletApp* can send the transaction details again. When *WalletServer* receives details for a new transaction while waiting for User's confirmation/PIN for a previously initiated transaction, it can infer that some message was lost and it sends a new signed message or a new signed `input_form` with a new nonce. In other words, *WalletServer* supports at most one pending transaction at any given time.

3.3.5 Password Reset and Device Theft

If the user forgets his login password or transaction PIN and needs to reset it, then *WalletServer* reverifies the user's identity by sending an OTP via *KYCServer* before allowing him to set a new password/PIN. Unfortunately an application cannot differentiate between the actual owner of a smartphone and someone who has stolen the owner's smartphone and attempting to reset the password to be able to execute transactions. A user's primary protection against device theft is on-device authentication using PIN and/or biometrics such as face scans, iris scans & fingerprints. If the thief somehow manages to unlock the device, then transaction PINs can act as a secondary protection. The payment service provider has to require the user to enter a PIN for transfering a large sum of money, which will limit the user's loss in the event of device theft. If the user deregisters his $PubKey_{ID}$ using $DeRegCode_{ID}$ (Section 3.3.1) soon after theft, then the thief will fail to reset transaction PIN because *KYCServer* will be unable to send an encrypted OTP after deregistration of the required public key.

3.4 Security provided by Protected I/O

The Protected I/O API proposed by us can fully protect a user from fraudulent transactions even if the normal world software is already compromised before

the user installs the digital wallet app. Protected I/O helps in establishing a secure authenticated channel of communication between the user and the wallet provider's server. This is done by first associating a public key $PubKey_{ID}$ with the user's identity as described in Section 3.3.1. The process involves user's physical presence along with the device that securely stores the $PrivKey_{ID}$. The `DISPLAY_PUBLIC_KEY_FOR_SERVER` API ensures that the attacker cannot fool the user into enrolling a public key for which the corresponding private key is owned by the attacker. Once $PubKey_{ID}$ is enrolled, *KYCServer* has the ability to deliver secret information to the user. This ability can then be used by a wallet service provider (or other service providers) to securely deliver secret OTPs to the user, which was not possible with SMS alone. Securely delivered OTPs are required for reliably associating *WalletAppPubKey* with the user's identity. Once that is done, any message signed with *WalletAppPrivKey* can be assumed to have the user's concurrence. Enrolment of *WalletAppPubKey* and signing of messages using *WalletAppPrivKey* was already possible with Android's hardware-backed Keystore. Protected Confirmation, which was introduced with Android Pie shortly after we started our work on secure payments, helps in ensuring that messages are signed only after the user has seen them and given their consent. But Protected Confirmation can only associate a public key with a device, it cannot verify whether the right person owns the device. That is where the `DISPLAY_PUBLIC_KEY_FOR_SERVER` and the `DISPLAY_SECRET_MESSAGE_FROM_SERVER` APIs come in. Once the wallet provider is assured that *WalletAppPrivKey* resides in the correct user's device, *WalletAppPubKey* becomes a reliable proxy for the user. The `GET_USER_CONFIRM_FOR_SERVER` serves the same function as Android's Protected Confirmation except that unlike in case of Protected Confirmation, the message seen by the user is verified by the TEE using the server's public key stored in the TEE. The `DISPLAY_MESSAGE_FROM_SERVER` method is meant to ensure that an attacker is unable to use the compromised normal world app to give incorrect instructions to the user about using the wallet service properly.

Like any security solution, the security provided by our design does not exist in a vacuum. The design assumes that the people in charge of enrolment of $PubKey_{ID}$ at the Enrolment Centre are trustworthy. The security indicator used in our design is useful only if the user pays attention to it.

3.5 Portability

Although our work targets Android OS, the design we have proposed is not tightly coupled with it. On the normal world side it requires a background system service and a kernel that can communicate with a TEE. Within the TEE it requires an ability to boot the device with signed binaries, to store keys in a secure location, to access a dedicated partition in the RAM and to wrest control of the touchscreen from an untrusted kernel. None of these things are unique to Android or to ARM TrustZone. While the details on internals of iOS are scant compared to Android, it is known that modern iOS devices have a dedicated subsystem on the SoC for trusted boot and cryptography called Secure Enclave in the official documentation [67]. The official documents do not mention any UI capability for the Secure Enclave which makes it unusable for our design. But missing hardware capabilities can be introduced in newer device models. iOS devices also use ARM processors just like Android and even if they didn't, TrustZone is not the only TEE system in existence. Even for implementing our design on Android, the hardware will need some changes. Android devices don't have a TEE-protected LED to serve as a security indicator yet. Our work makes a good case for introducing this change in new smartphones.

Chapter 4

Implementation and Evaluation

Chapter 3 described a platform-agnostic design for a secure digital payments system. The design is not tied to any specific TEE implementation. Any TEE implementation that provides a remotely verifiable root of trust and can give the trusted OS exclusive control over a display device, an input device & a security indicator, can be used to secure digital payments using our design. We have chosen to implement the design using ARM TrustZone [88] which is the most widely deployed TEE implementation because ARM processors dominate the smartphone market. For the untrusted OS, we have chosen Android, which is used in more than 80% smartphones. For the trusted OS we have chosen OP-TEE [49] as it is open source and has a large community of developers supporting it.

For our PoC implementation we have used the HiKey 960 development board [11] which has the Kirin 960 SoC from HiSilicon. Kirin 960 is an ARM-based octa-core 64-bit SoC. For the software stack we have made use of existing work done by Linaro's Security Working Group [63, 66] to integrate Android Open Source Project (AOSP) [9, 10] and OP-TEE [49] for HiKey boards. AOSP is the official upstream code collection for Android maintained by Google. Linaro SWG maintains a repository named *optee_android_manifest* on Github [65] that helps compile OP-TEE OS and ARM Trusted Firmware along with Android OS. We have used version 3.4.2 of this repository which combines the following major components

- Android Pie (9) revision r30

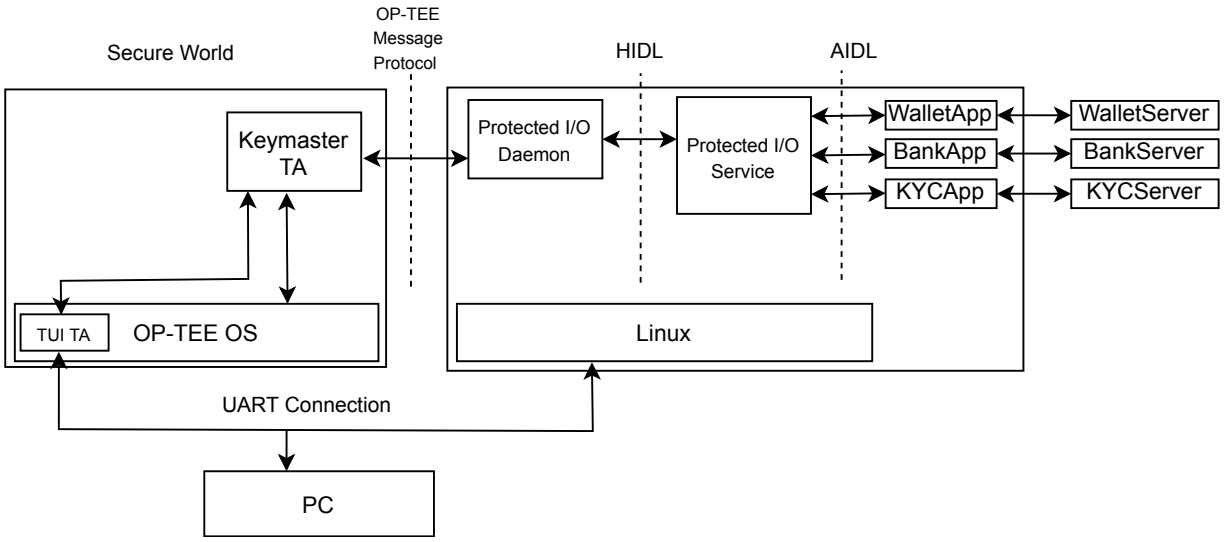

Figure 4.1: Protected I/O implementation

- Android Kernel 4.14
- OP-TEE OS 3.4.0
- ARM Trusted Firmware 2.0

There is no publicly available graphics driver for the secure world on HiKey 960 board and only the hardware vendor is in a position to develop such a driver from scratch. So, our PoC implementation does not have a touchscreen-based trusted UI. We have tried to approximate a Trusted UI over a UART connection with a PC. This Trusted UI is text based and therefore very primitive compared to a graphical UI that could have been implemented if we had access to a graphics driver for the secure world. Figure 4.2 shows a screenshot of the Trusted UI displaying an input form. Because a UART connection is very slow compared to an HDMI or a MIPI-DSI connection, our Trusted UI implementation is not very performant. The Kirin 960 SoC does have a TZPC (TrustZone Protection Controller) but HiSilicon has not published the documentation for it. TZPC can be used to give the secure world exclusive control over the UART interface and protect it from a potentially compromised normal world. But due to unavailability of documentation, we have not been able to make use of the TZPC on the HiKey 960. Our attempts to reach

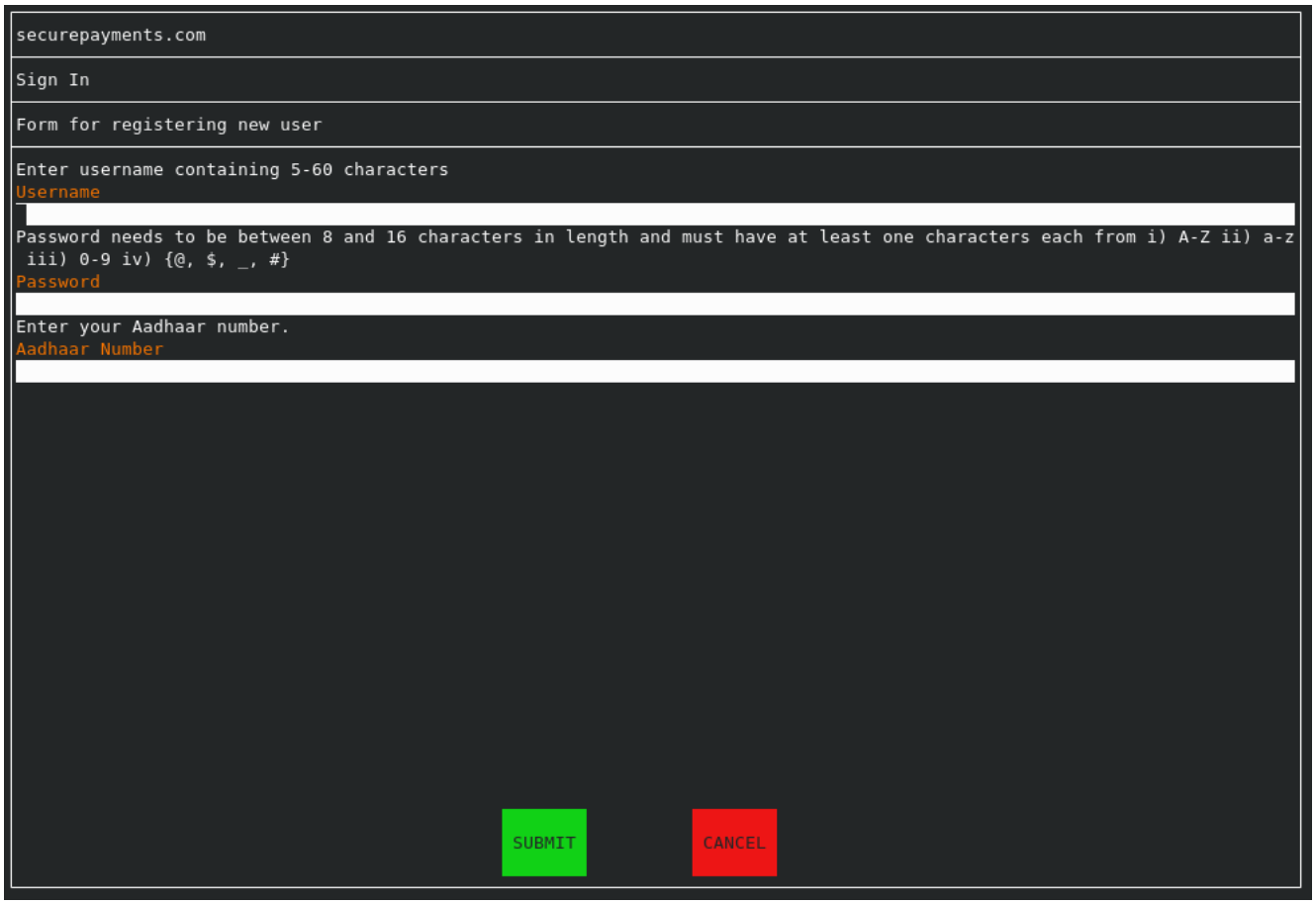

Figure 4.2: Text based Trusted UI

out to HiSilicon via relevant online forums did not receive any response. As a result, in our implementation, the secure world cannot prevent the normal world from accessing the UART interface. Hikey 960 also lacks the ability to protect regions of the DRAM from the normal world.

4.1 Trusted Application

Instead of developing the Protected I/O TA as a separate app, we have chosen to modify the Keymaster TA developed by Linaro SWG [64] as part of their implementation of version 3.0 of the Keymaster HAL specification [20]. The Keymaster TA is the secure world component for Android's hardware-backed Keystore system

discussed in Section 2.6.2. The Keymaster TA developed by Linaro SWG recognizes 15 commands from the normal world for implementing Keystore system. We introduced 12 new commands corresponding to the 12 API methods described in Section 3.2.2. All these commands expect exactly one CBOR-encoded map containing the arguments and return a CBOR-encoded map containing the output. The output map always has at least one key, *error_code*, which indicates the type of error (if any) that happened during execution of the command. For CBOR encoding/decoding we have integrated a lightweight open-source CBOR library called *NanoCBOR* [46] into the Keymaster TA. For cryptographic operations and X.509 certificate handling we have used *mbedTLS* 2.6.1 [79] and *LibTomCrypt* 1.17 [42] both of which were already integrated with version 3.4.0 of OP-TEE OS. As it is not possible to display a JAB code on our text-based trusted UI, we have not implemented `DISPLAY_PUBLIC_KEY_FOR_SERVER` API method. Also, since there is no fingerprint reader on the Hikey960, we have not implemented biometric authentication for the keys generated for Protected I/O. The Gatekeeper TA (also developed by Linaro SWG) has been modified to communicate the current time to the Keymaster TA whenever the user successfully authenticates using a PIN / Password. This helps in deploying some protection against device theft. If it has been too long since the last time the user entered the correct password / PIN, the Keymaster TA can refuse to sign/encrypt Protected I/O messages and require the user to reauthenticate. For creating attestation certificates we have made use of the attestation functionality already implemented in SWG's Keymaster TA. For validating server certificates, we have hardcoded a single root CA certificate in the Keymaster TA. Since user input will be available from the PC over UART, we did not have to implement on-screen keyboard in the TA. But a commercial implementation will need to display an on-screen keyboard in the trusted UI.

4.2 TEE Kernel Driver

When the secure world needs access to the touchscreen, it should inform the normal world so that the normal world can refrain from trying to access it and then inform the normal world again after it is done using it. In our implementation, a PC connected to Hikey960 via UART acts as a crude substitute for a touchscreen. On the Hikey960, the UART channel is used for the serial console by the Linux kernel.

The secure world also uses the same UART channel for all its log output. We had to make some minor modifications to OP-TEE's kernel driver to prevent the output of the serial console from the Linux kernel from ruining the display of trusted UI from the secure world. Two commands were added to the RPC interface between OP-TEE's kernel driver and the OP-TEE OS running in the secure world. These two commands use Linux kernel's `console_lock` and `console_unlock` methods to obtain and release mutex locks on the serial console. Some changes were also made to the logging mechanism in OP-TEE OS. Since the same UART channel is to be used for logging and for the trusted UI, locking was introduced in the logging system of OP-TEE OS to allow synchronization with the trusted UI system. When the secure world needs to display something to the user or get some input from the user using the trusted UI, it uses RPC to obtain a lock on the serial console of Linux kernel to prevent any further output from it and then the trusted UI implementation also obtains a lock on OP-TEE's own serial console to prevent OP-TEE from logging anything while the trusted UI is being displayed to the user. In a commercial implementation of our design, similar approach will be required to share the same touchscreen between the two worlds.

4.3 HIDL Interface

HIDL (HAL Interface Description Language) [34] was created by Google to facilitate communication between the vendor-independent and vendor-dependent parts of Android platform. It was introduced in Android Oreo as part of Project Treble. We have added a new HIDL interface for Protected I/O and we have also implemented this interface. The following listing shows the methods and types in our HIDL interface.

Listing 4.1: HIDL interface for Protected I/O

```
import android.hardware.keymaster@3.0::KeyParameter;

struct ServerInfo {
    string url;
    vec<vec<uint8_t>> certChain;
}
```

```
enum ProtectedIOErrorCode : uint8_t {
    SUCCESS = 1,
    ABORTED = 2,
    SYSTEM_ERROR = 3,
    INVALID_SIGNATURE = 4,
    UNSUPPORTED_KEY_SIZE = 5,
    UNKNOWN_ELLIPTIC_CURVE = 6,
    URL_FORMAT_INVALID = 7,
    KEY_PAIR_NOT_GENERATED = 8,
    INVALID_SERVER_CERTIFICATE = 9,
    USER_CANCELED = 10,
    MESSAGE_TOO_LONG = 11
}

interface IProtectedIO {
    generateKeyPairForServer(ServerInfo serverInfo,
            vec<KeyParameter> keyParams)
        generates (ProtectedIOErrorCode error, vec<uint8_t> pubKey)

    getPublicKeyForServer(ServerInfo serverInfo)
        generates (ProtectedIOErrorCode error, vec<uint8_t> pubKey)

    changePublicKeyForServer(ServerInfo serverInfo)
            generates (ProtectedIOErrorCode error)

    getAttestationForServer(string url, vec<uint8_t> challenge)
        generates (ProtectedIOErrorCode error,
                vec<vec<uint8_t>> certChain)

    getSignedInputForServer(string url, vec<uint8_t> fromServer)
        generates (ProtectedIOErrorCode error, vec<uint8_t> forServer)

    getSecretInputForServer(string url, vec<uint8_t> fromServer)
        generates (ProtectedIOErrorCode error, vec<uint8_t> forServer)

    displayMessageFromServer(string url, vec<uint8_t> fromServer)
        generates (ProtectedIOErrorCode error, vec<uint8_t> forServer)
```

```
    displaySecretMessageFromServer(string url, vec<uint8_t> fromServer)
        generates (ProtectedIOErrorCode error, vec<uint8_t> forServer)

    getUserConfirmForServer(string url, vec<uint8_t> fromServer)
        generates (ProtectedIOErrorCode error, vec<uint8_t> forServer)

    displayPublicKeyForServer(string url)
        generates (ProtectedIOErrorCode error)

    abort(vec<uint8_t> input) generates (ProtectedIOErrorCode error)
}
```

4.4 Android System Service

A new System Service (privileged background process), *ProtectedIOService*, has been added to the vendor-independent part of Android platform. This System Service allows applications to communicate with the Keymaster TA via the HIDL interface to make use of Protected I/O. Applications communicate with the System Service using Android's Binder IPC mechanism [25]. To enable this communication we introduced an AIDL (Android Interface Description Language) [6] interface and implemented it in C++. The following listing shows the AIDL interface.

Listing 4.2: AIDL interface for Protected I/O

```
package android.security;


interface IProtectedIOService {
    int generateKeyPairForServer(
            String url, in List<byte[]> certChain,
            String curveName, boolean biometricAuthReqd,
            out byte[] pubKey);

    int getPublicKeyForServer(String url, out byte[] pubKey);

    int changePublicKeyForServer(String url, in List<byte[]> certChain);

    int getAttestationForServer(String url, in byte[] challenge,
            out List<byte[]> certChain);
```

```
    int getSignedInputForServer(String url, in byte[] fromServer,
            out byte[] forServer);

    int getSecretInputForServer(String url, in byte[] fromServer,
            out byte[] forServer);

    int displayMessageFromServer(String url, in byte[] fromServer,
            out byte[] forServer);

    int displaySecretMessageFromServer(String url, in byte[] fromServer,
            out byte[] forServer);

    int getUserConfirmForServer(String url, in byte[] fromServer,
            out byte[] forServer);

    int displayPublicKeyForServer(String url);

    int abort();
}
```

4.5 Android Apps

We have implemented three Android apps.

- WalletApp
- BankApp
- KYCApp

The roles of these three apps have been described in Section 3.3. The three apps make use of the Protected I/O functionality via the new System Service created by us. AOSP does not provide a push message service out of the box. In Android smartphones, a push message service is available as part of Google Mobile Services [30] which is a collection of proprietary apps and services created by Google. GMS is not available on developer hardware such as Hikey960. Due to lack of a push message service, we have not been able to implement push notifications in the three apps. Push notifications are required for delivering messages from the server

e.g. the OTP from the KYCServer (see Section 3.3.2). Since no push message service is available, in our implementation, the apps request messages from the server when required. Each of the three apps has a hardcoded X.509 certificate for the corresponding server. These certificates are required to initiate generation of Protected I/O public keys for the servers. These certificates are signed using a root CA which is hardcoded into the Keymaster TA.

4.6 Application Servers

Corresponding to the three apps mentioned in the previous section, we have implemented three server programs.

- WalletServer
- BankServer
- KYCServer

The server programs are written in Python and served using the Apache web server. Since we did not implement the `DISPLAY_PUBLIC_KEY_FOR_SERVER` API method, the enrolment of public key with the KYCServer can't happen as described in Section 3.3.1. For the purpose of demonstration, the KYCServer is programmed to accept any public key received from the KYCApp without performing any verification. The BankServer only supports NetBanking transactions. We haven't implemented EMV CNP (Card Not Present) functionality.

The source code of our implementation will be made available on Github [51] along with instructions to build and use it.

4.7 Performance

We have measured the time spent in the secure world for key pair generation, public key fetching, key pair deletion and key attestation. The measurements were taken over 1000 runs each of the 4 operations. A small normal world client program with a hardcoded server certificate for a P-384 elliptic curve was used for this purpose.

OPERATION	Min (μs)	Max (μs)	Mean (μs)
GENERATE_KEY_PAIR_FOR_SERVER	534139	841592	613053
GET_PUBLIC_KEY_FOR_SERVER	1843.75	13454.2	4813.47
DELETE_KEY_PAIR_FOR_SERVER	9293.23	112708	28202.4
GET_ATTESTATION_FOR_SERVER	97651	223157	120597

For a thorough evaluation of the security of our design, we need hardware that can protect a dedicated part of the memory from the untrusted normal world software instead of relying on the normal world kernel to voluntarily keep off the memory it is not supposed to access. We need the hardware to be able to take exclusive control of the touchscreen. We also need the ability to program a public key in secure storage to act as the root of trust for trusted boot to enable us to boot binaries signed by us. Not only should the hardware be capable of all these things, we also need the official documentation that describes how to use them. Unfortunately, this information is usually protected by non-disclosure agreements and only made available to a few organizations. As mentioned earlier, the Hikey960 board does have a TZPC. But the documentation for it is not public, presumably because the same SoC is used in some commercially available Huawei phones and tablets. On the right hardware, we can evaluate our design by booting a kernel with known vulnerabilities and exploits and attempting to steal information from the TEE.

Chapter 5

Related Work

5.1 Payment Security

Reaves et al. [118, 119] did two studies, one year apart, on the security of some popular mobile banking applications. They used automated and manual analysis for finding common vulnerabilities such as improper TLS certificate validation, use of client side authentication, information exposure through logs, etc. In the first study they found that several popular apps had glaring vulnerabilities. They reported these vulnerabilities to the developers. Unfortunately, in the second study, only a few of the reported vulnerabilities were found to be fixed. Castle et al. [95] conducted a large-scale analysis of many Android apps that provide financial services and interviewed seven developers from Africa and South America. They have given a detailed threat model with users, agents and organization employees as potential adversaries and a list of potential attacks that they can carry out on a financial service. From their analysis of apps, they found that some vulnerabilities can be fixed by simply dropping support for really old versions of Android. From the interviews, they found that vulnerabilities are caused by incomplete threat models (e.g. focusing on threats to the organization but ignoring threats to the users), limited security education, budget constraints, restrictions imposed by partner organizations (e.g. avoiding biometric authentication) and overuse of code from Stack Overflow [69].

Rahaman et al. [117] have assessed the real-world enforcement of PCI Data Security Standard. They created an e-commerce web application called *BuggyCart*

with PCI DSS related vulnerabilities and evaluated its standard compliance using PCI scanners. They found that these scanners certified BuggyCart as compliant despite presence of vulnerabilities. They also created their own scanner called *PCI-CheckerLite* using which they scanned more than a thousand payments-card-taking websites, more than 80% of which were found to be non-compliant. Mahmud et al. [112] designed a tool called *Cardpliance* to check PCI DSS compliance of Android apps that take credit card numbers from the users. The tool uses static program analysis to identify potential PCI DSS violations. The authors found that more than 98% of the 358 apps they analyzed with Cardpliance handled credit card numbers correctly.

Murdoch et al. [114] have criticised the 3-D Secure Protocol (3DS) 1.0 that was made available by VISA under the brand name *Verified by VISA*, by MasterCard under the name *SecureCode* and by Discover under the name *ProtectBuy*. The 3DS form in which customers enter card details and authentication credentials is made available by merchants in an iframe on their websites. Since iframes do not have an address bar of their own, customers have no way of verifying whether the details they are entering are being sent to the issuing bank (or someone authorised by them). This made customers vulnerable to phishing attacks. 3-D Secure Protocol 2.0 [2, 3, 4] is a major improvement over the 1.0. Corella et al. [98] found issues in 3DS 2.0. The fundamental issue is that the 3DS Server cannot verify whether the 3DS Requestor app is using an EMVCo-approved 3DS SDK, which is a requirement in the protocol specification. In the current mobile app ecosystem, it is not possible to verify the integrity of individual libraries loaded by apps. The app as a whole is signed by developers and the signature is verified by mobile OS (Android and iOS) during app installation. But even if the supplier of the 3DS SDK signs the SDK library, the app store cannot verify whether this signed copy is embedded in the 3DS Requestor app. Thus a malicious merchant can develop and publish a 3DS Requestor app that has a modified 3DS SDK embedded in it and use it to steal sensitive information from cardholders.

Kumar et al. [104] have analysed the security of UPI 1.0. Because the protocol details are not public, they had to reverse engineer several UPI payment apps (e.g. BHIM). They found security holes in the protocol and showed potential attacks that use an attacker controlled app on the victim's device to discover the victim's

bank accounts, which is sensitive private information, and in the worst case execute transactions from the victim's account. They were unable to reproduce the same attacks on the updated UPI 2.0 protocol.

5.2 TEE-based Security

TZ-RKP [89] protects the normal world kernel using TrustZone. By instrumenting the normal world kernel, TZ-RKP prevents it from performing certain critical tasks such as page table updates. These tasks are performed by the secure world instead. Because page table updates are made by the secure world, it is able to prevent the double mapping of physical memory containing critical kernel data into user space virtual memory. The secure world also prevents code injection into normal world kernel by refusing to set the executable permission on any new page. TZ-RKP is deployed in Samsung devices under the name Samsung Knox. *SPROBES* [100] is very similar to TZ-RKP.

Brasser et al. [93] have shown how TrustZone can be leveraged to regulate the behaviour of smart devices in restricted spaces such as examination halls, private meetings, etc. Their system has a policy server that uses its read/write access to the device memory to enforce the host's policy and a vetting server which inspects the policy to prevent malicious hosts from compromising the guest's privacy and security. TrustZone helps in establishing trust in the device's policy enforcement code. In Privaros [90], the remote attestation ability provided by a TrustZone-based TEE is used to verify the integrity of a software stack running in the normal world on delivery drones. The host of the airspace over which the drones need to fly, can use remote attestation to verify the integrity of the software stack and be assured that their privacy policy will be correctly enforced on the drone.

SeCloak [105] is a solution for reliable on-off control over smartphone peripherals such as camera, microphone, etc. even in the presence of a compromised normal world kernel. It uses TrustZone to run a secure kernel which is responsible for securely displaying the user's peripheral control preferences received from a normal world app and getting the user's confirmation for enforcing those preferences. An LED protected by secure kernel is used as security indicator to assure the user that the display device is under the secure kernel's control. As long as there is at least one disabled peripheral, the secure world prevents a device reboot to ensure that

a compromised normal world kernel is unable to re-enable a disabled peripheral. Liu et al. [110] proposed two software abstractions for exposing sensors present in mobile devices to applications and cloud services - *sensor attestation* and *sensor sealing*. Sensor attestation protects the integrity and authenticity of sensor readings by attesting both the reading and the code that produced it. Sensor sealing takes a secret and a policy as input and seals it in such a way that unsealing only happens when the sensor reading conforms to the policy. The authors used TrustZone to implement these two abstractions on ARM platform and Intel's Trusted eXecution Technology [35] for the x86 platform.

Several researchers have worked on the problem of establishing a *trusted path* between a user device and an Internet service. TrustUI [106] achieves a very small TCB by reusing the input, display and network drivers of the normal world instead of having separate drivers in the secure world. The unmodified normal world driver acts as a backend and it has a corresponding frontend in the secure world that requests its services. Both parts communicate with each other using proxies in both worlds that exchange data using shared memory. For secure display, the frontend requests a framebuffer from the backend and on receiving it, uses TZASC to configure that memory region to be accessible to secure world only. Using TZPC, the secure world prevents the normal world from accessing the display controller. This prevents a screen capture attack. But a compromised normal world can still mount an overlay attack by passing a low priority framebuffer to the secure world and itself writing to a higher priority framebuffer to affect what the user eventually sees. To prevent this the authors have suggested the use of two multicolour LEDs - one each for foreground and background colour - as security indicators. These two LEDs will be exclusively controlled by the secure world. The user is expected to match the colour of the LEDs with the foreground and background colour of the display to assure himself that it is a secure display. The colour is randomized to prevent a compromised normal world from fooling the user by matching the colours used by the secure world. Because all touch inputs are forwarded to the secure world by the normal world, to prevent the normal world from recording keystrokes, the on screen keyboard layout is changed with every key press. As opposed to TrustUI, in our design we assume the existence of dedicated display and touch drivers in the secure world. Thus we need only one LED of a single

colour and there is no keyboard randomization. The cost for this difference is a larger TCB. In VeriUI [109], a stripped-down browser called SecureWebkit is used for secure I/O. No security indicator is used in that design.

TruZ-Droid [129] is very closely related to our work. The design allows any app to leverage TrustZone for protecting the user's secret input & confirmation and to send the secrets to a user-authorized server. To protect SSL connections from being sabotaged by a malicious normal world, the authors have proposed splitting SSL functionality such that all cryptographic operations (e.g. X.509 certificate validation, encryption, etc.) are done in the secure world. The authors have reused the HTTP stack and TCP/IP stack of the normal world with some minor changes to allow carrying of encrypted data from the secure world. They have used an LED controlled by secure world as a security indicator. Unlike TruZ-Droid, in our design, there is no need of a persistent SSL connection for the secure world to send data to the server. TruZ-View [130] is another closely related work. In TruZ-View, the authors propose reusing the UI stack of the normal world instead of creating a completely independent UI stack for the secure world. In their design the normal world first renders the whole UI of an app except for the parts that either display sensitive information or take sensitive input from the user. The rendered UI is handed over to the secure world in the form of a screenshot along with co-ordinates of UI region(s) that will be used for confidential display, confidential input or protected user confirmation. The secure world then overlays protected UI elements on top the screenshot sent by the normal world and displays the result to the user. Again, an LED controlled exclusively by the secure world is used as a security indicator to the user. Any input entered by the user is stored securely by the TEE and only a reference to it is returned to the normal world. For sending confidential data to the server, the authors have made use of the split SSL design of TruZ-Droid. Unlike TruZ-View, our design uses a completely independent UI stack in the secure world which can only render a simple form and a keyboard as opposed to the rich UI possible with TruZ-View. VButton [107] proposes using signed images received from a trusted server to create the UI in the secure world. A signed image from the server can be used to show a confirmation dialog to the user and if the user confirms the action, the TEE can send an attestation of the image to the normal world which can pass it on to

the server for verification. If the attested image matches the one sent earlier by the server, then the server is assured that the user was shown correct information. Unlike VButton, our design only needs signed text from the server and not images. This means less work for the server but also means less control over the look and feel of the secure world's UI. SchrodinText [122] deals with only secure display and not secure input. In SchrodinText, confidential text is encrypted by the app's backend server using a symmetric key established in a handshake with the secure monitor during install/login time. The normal world sends the rasterized glyphs for all the characters of the currently chosen font style and font size and layout information to the secure world. The secure world decrypts the text received from the backend server, chooses the correct glyphs from the set of glyphs received from the normal world and places the glyphs in the correct position in the framebuffer using the layout information received. A novel technique that the authors have named *multi-view page* is used to prevent the potentially malicious normal world OS from being able to access the confidential contents of the framebuffer. The MMU and IOMMU on the device are programmed by the secure monitor to show a protected view of the framebuffer (with all the confidential text) to the display device and an unprotected view of the same framebuffer to the OS, the GPU and other I/O devices on the system. Thus without using an additional framebuffer secure world manages to prevent the normal world from accessing confidential information. But this design requires significant changes to the normal world rendering libraries and the OS kernel. Additionally, the compositing of glyphs by the secure monitor increases the latency for rendering text.

Zheng et al. [131] proposed a secure mobile payments system which uses a Mobile Trusted Module (MTM) [43] service and a payment service running in the secure world. The MTM service provides key generation, encryption and remote attestation and the payment services handles displaying of payment information and user input. The remote attestation method provided by the MTM service in their design is different from our design. In their design the device is attested as a whole by the MTM service that functions like a Trusted Platform Module. In our design an individual public key is attested using a certificate present in a secure storage. For authenticating a server, their design proposes encrypting the server's public key using a public key generated by the MTM service and sent to the server.

This encryption of the server's public key is unnecessary and also ineffective for verifying the identity of the server. We believe that our approach of using PKI certificates for verification is more secure. Unlike our design, in their design, the payment information sent by the normal world app to the secure world is not signed by the app's server. Our design is more general than theirs because there is no need for a separate payment TA. Zheng et al. published one more work [128] that proposed a secure payment system called TrustPay. Just like the previously described work, this work proposed encryption of the server's public key using a public key generated in the secure world and sent to the server. In TrustPay the authors have not made use of remote attestation. In TrustOTP [124], authors have suggested running HOTP [56] and TOTP [58] algorithms in the secure world along with a secure touchscreen driver for display and user input. In our design the OTPs are not generated on the smartphone but on the app's server.

Some authors have tried to tackle the problem of using TEEs for protecting legacy applications which were written before TEEs became widely deployed with no/minimal modifications to the code. In TrustShadow [101], the authors have proposed a runtime system that protects applications from a malicious OS without requiring any changes in the applications. The runtime system executes a 'shadow' process in the secure world that corresponds to a 'zombie' process in the normal world that never gets scheduled to run. The runtime system manages the page tables of the process in the secure world, handles page faults and page table updates by leveraging the page fault handler of Linux (with integrity checks for the returned values to prevent malicious modifications), intercepts exceptions & system calls to forward them to Linux and performs random number generation and floating point computations. The values returned by system calls are verified to prevent Iago attacks [97]. Rubinov et al. [120] developed an automated partitioning framework for splitting an existing application into two parts, one for the normal world and the other for the secure world. In their approach a developer is required to annotate sources of confidential data in the application's source code to facilitate taint analysis for identifying *sinks* where confidential data is output. The framework takes the original application's binary and the annotations as input and outputs refactored source code in which all the methods that handle confidential data are isolated as static Java methods. Developers are expected to transform the static

Java methods to C code which in turn is transformed to TEE-specific code by a *Wrapper* that is part of the framework. Lind et al. [108] developed a framework with similar aim of partitioning existing applications but their work targets Intel SGX [60] instead of ARM TrustZone and only requires the developers to annotate sources of confidential data as opposed to the work by Rubinov et al. which required developers to write additional code.

A paper by Tamrakar et al. [125] is closely related to ours. They provided a reference design for enabling the use of TEEs for storing and using the electronic IDs that are issued to European citizens. They implemented their design using two different TEEs - ARM TrustZone and Onboard Credentials [103] available on Nokia Windows 8 Phones. In their design a pre-existing smart card based electronic ID enables remote enrolment of user's smartphone as a substitute for the smart cards, unlike our design in which a user has to physically visit an enrolment centre. The smart card based electronic ID is used to login to an online enrolment portal which sends a registration code in the form of a QR code, which is then scanned by the phone's camera. The EId TA running in the secure world sends back this registration code along with a certificate request (CSR) to the enrolment server which verifies the registration code and forwards the CSR to a CA which generates a certificate for the user that is forwarded to the user's device by the enrolment portal. Marforio et al. [113] have suggested software and hardware changes to the baseband processor on smartphones to enable usage of SMS for enrolment of user's smartphone. Their work suggests that baseband processor should be able to recognize specially crafted SMSs meant for enrolment and they should only forward these SMSs to the secure world unlike. Until the changes suggested by them are introduced in baseband processors, our suggestion of physical visits to enrolment centres can be followed.

5.3 Attacks on TEEs

Since the security of our payment system relies on the security of the TEE implementation in the user's smartphone, a successful attack on the TEE will nullify the security provided by our design. Several researchers have found design and implementation flaws in various TEEs currently deployed in consumer devices.

Machiry et al. [111] showed how the 'semantic gap' between the secure and the

non-secure world can be exploited by a malicious normal world app to perform a *confused deputy attack* that causes the secure world to modify memory not owned by that app. They were able to exploit this vulnerability in commercial TEE implementations from Qualcomm and Trustonic. They have also proposed a novel defence they dubbed *Cooperative Semantic Reconstruction*. In this defence, the normal world's untrusted OS includes the PID of a normal world app in SMC calls to the secure world. When a TA requests the trusted OS to resolve a virtual address received as argument from the normal world, the trusted OS queries the untrusted OS with the address, the buffer length and the PID. The untrusted OS checks whether the buffer belongs to the address space of the process with the given PID and if the check is successful, it returns corresponding physical address. The trusted OS then checks whether the returned physical address is part of untrusted region of memory (which is accessible to the untrusted OS) and if the check is successful, the TA is allowed to access that buffer.

Ning et al. [115] performed a security analysis of debugging features in ARM processors and found security vulnerabilities that can allow a malicious normal world OS to gain complete control over code executing in the secure world. Since ARMv7, it is possible for an on-chip processor to debug another on-chip processor within the same SoC [13]. The debug authentication signals available on the platform only consider the privilege level of the debug target and not the debug host. It is possible for a processor executing code in the non-secure mode to debug another processor on the same SoC executing code in secure mode, thereby breaking the isolation that TrustZone is supposed to provide. Ning et al. found that debug authentication signals were enabled by default on commercial devices like Raspberry Pi, Huawei Mate 7, Motorola E4 Plus, Xiaomi Redmi 6, etc. The authors crafted an attack they have called *Nailgun* that was able to steal AES encryption key from the secure world on an NXP i.MX53 Quick Start Board [77], execute arbitrary payload in EL3 on ARMv8 Juno board r1 [37] and extract the fingerprint from secure storage on Huawei Mate 7 [44].

Fabian et al. [99] analyzed the exploitability of TAs on OP-TEE OS, which is the trusted OS we have used in our implementation. They implemented multiple vulnerable TAs with vulnerabilities inspired by flaws found in commercial TEEs [86, 91, 92, 94, 123]. They have made their vulnerable TAs available at [29]. The

TAs have memory corruption bugs that allow control flow hijacking or arbitrary code execution.

Cerdeira et al. [96] did a systematic study of disclosed vulnerabilities in commercial TrustZone-based TEE implementations. After analyzing 207 TEE bug reports from 5 major vendors - Qualcomm, Huawei, Trustonic, Linaro, NVIDIA - the authors came up with a list of 23 different issues with TEE implementations such as overly bloated TCBs, weak ASLR, failure to prevent TA downgrades by the untrusted OS, cache side channels, etc. and a list of 9 defences proposed by the research community.

Chapter 6

Conclusion

In this work we have shown how TEEs can be leveraged to securely obtain confidential input from a user and send it to a remote server even when the kernel is controlled by an attacker. Our design can be used to protect confidential input for any app, but payment apps were the primary focus for us. Our design is platform-agnostic and can be realized with any TEE implementation that allows exclusive access to a touchscreen and a security indicator. We hope that our PoC implementation using ARM TrustZone and Android will be useful for other researchers working on smartphone app security in general and digital payment security in particular. Because of the absence of a security indicator in the smartphones currently available in the market, our design cannot be deployed commercially with a simple software update. But we believe that our work makes a good case for introducing a security indicator LED in new smartphone devices. Because of the limitations of the hardware, unavailability of documentation and paucity of publicly available drivers for the secure world, we weren't able to create a truly secure implementation. This is one possible direction for future work. The hardware needs to have user programmable secure boot. It also needs a graphics driver and a touchscreen driver for the secure world so that the secure world can display things on the screen and take input from the user. The graphics and touch drivers don't need to have advanced features like multi-touch support, 3D acceleration, etc. Given the required information about the graphics hardware, a simple driver can be developed from scratch. Once we have graphics and touch capability, we

will need to write software for rendering text on the screen. We don't need support for multiple fonts or text formatting because the Trusted UI is not meant to be used for entering large amounts of formatted text. It only needs one font with glyphs for characters from all major languages that need to be supported by a phone. Our current Trusted UI implementation can only support the ASCII subset of the larger UTF-8 character set. With the text rendering facility developed we need to write a Trusted App for an on-screen keyboard. The current implementation is also missing support for biometric authentication using fingerprints or face scans because the Hikey960 board we have used does not have a fingerprint reader or a camera integrated with it. Without biometric authentication we cannot have strong local authentication on the device and therefore cannot limit the harm caused by device theft.

We have reasoned about the security of our design in Chapter 3, but we haven't formally verified the security of the proposed protocols. We plan to use Verifpal [102, 85] to model and test our protocols. We also intend to try and have our API accepted into the official upstream Android project. For that we will have to make sure that our implementation works with the latest version of Android.

Appendix

A Data Types used in Protected I/O API

```
error_code = uint .size 4

; DER-encoded X.509 public key certificate
cert = bstr

;Server's Common Name as recorded in its certificate
common_name = tstr
server_info = {
    common_name: tstr,
    cert_chain: [ + cert]
}

key_params = {
    ; Name of a standard elliptic curve
    curve: tstr,
    ; Whether biometric authentication is required for using key
    bio_auth_reqd: bool
}

attestation_challenge = bstr .size 8

; Input form received from server
input_form = {
    is_confidential: bool,
    title: tstr,
    ; Optional description for the form
```

```
    ? description: tstr,
    ; one or more form fields
    fields: [+ form_field]
}

form_field = text_field // password_field // integer_field //
             currency_field // select_field // checkbox_field

field_types = (
    text: 1, password: 2, integer: 3, currency: 4,
    select: 5, checkbox: 6
)

;Common data in all types of input fields
form_field_base = (
    type = &field_types,
    label: tstr,
    ? description: tstr
)

; Different types of input fields
text_field = {
    form_field_base,
    min_length: uint,
    max_length: uint,
    ? default_val: tstr
}

password_field = {
    form_field_base,
    min_length: uint,
    max_length: uint
}

integer_field = {
    form_field_base,
    min_value: int,
    max_value: int,
    ? default_val: int
```

```
}

currency_field = {
    form_field_base,
    ; optional 3 letter currency code
    ? cur_code: tstr .size 3,
    ; a currency value expressed as two unsigned integers
    min_value: [ 2*2 uint ],
    max_value: [ 2*2 uint ],
    ? default_val: [ 2*2 uint ]
}

select_field = {
    form_field_base,
    choices: [ + tstr ],
    default_choice: uint
}

checkbox_field = {
    form_field_base,
    choices: [ + tstr ],
    ; minimum number of choices that need to be entered
    ; should be 0 for a non mandatory field
    min_choice_count: uint,
    max_choice_count: uint,
    ? default_choice: [ + uint ]
}

filled_form_field = {
    type = &field_types,
    ; for text and password field
    value: tstr //
    ; for integer field
    value: int //
    ; for select field
    value: uint //
    ; for currency field
    values: [ 2*2 uint ] //
    ; for checkbox field
```

```
    values: [ * uint ]
}

filled_form = {
    fields: [ + filled_form_field ]
}

encrypted_message_for_server = {
    message: {
        nonce: bstr .size 8,
        ; 32 bit value for time
        current_time: uint .size 4,
        ; Generated afresh for each new message
        ephemeral_pub_key = bstr,
        ; Encrypted CBOR-formatted byte array
        ; Its contents depend on the API call
        encrypted_data: bstr,
    },
    ; ECDSA signature on message field
    signature: {
        r: bstr,
        s: bstr
    }
}

encrypted_message_from_server = encrypted_message_for_server

signed_message_for_server = {
    message: {
        nonce: bstr .size 8,
        ; 32 bit value for time
        current_time: uint .size 4,
        ; Generated afresh for each new message
        ephemeral_pub_key = bstr,
        ; Encrypted CBOR-formatted byte array
        ; Its contents depend on the API call
        data: bstr,
    },
    ; ECDSA signature on message field
```

```
    signature: {
        r: bstr,
        s: bstr
    }
}

signed_message_from_server = signed_message_for_server
```

References

[1] 3-D Secure, . URL https://www.emvco.com/emv-technologies/3d-secure/. 37

[2] 3-D Secure 2.0 Announcement, . URL https://www.emvco.com/wp-content/uploads/documents/EMV-3DS-2-Spec-Launch-Final-October-2016.pdf. 37, 56

[3] EMV® 3-D Secure Protocol and Core Functions Specification, . URL https://www.emvco.com/wp-content/uploads/documents/EMVCo_3DS_Spec_v220_122018.pdf. vi, 17, 18, 56

[4] 3-D Secure Press Kit, . URL https://www.emvco.com/terms-of-use/?u=wp-content/uploads/documents/EMV-3DS-press-kit-FAQ_FINAL-1.pdf. 37, 56

[5] EMV® 3-D Secure Protocol and Core Functions Specification, . URL https://www.emvco.com/wp-content/uploads/documents/EMVCo_3DS_SDKSpec_220_122018.pdf. 18

[6] AIDL — Android Open Source Project. URL https://source.android.com/devices/architecture/aidl/overview. 51

[7] Dissecting the AMD Platform Security Processor, . URL https://www.youtube.com/watch?v=n9dhHG4tbE0. 8

[8] AMD PRO Security, . URL https://www.amd.com/en/technologies/pro-security. 8

[9] Git repositories for Android, . URL https://android.googlesource.com. 45

[10] Git repositories for Android, . URL https://source.android.com. 45

[11] Using Reference Boards — Android Open Source Project, . URL https://source.android.com/setup/build/devices. 45

[12] ARM1176JZF-S Technical Reference Manual, . URL https://developer.arm.com/documentation/ddi0301/h. 9

[13] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition: Part 3, Debug Architecture, . URL https://developer.arm.com/documentation/ddi0406/c/Debug-Architecture. 63

[14] TrustZone for AArch64, . URL https://developer.arm.com/architectures/learn-the-architecture/trustzone-for-aarch64/single-page. vi, 9

[15] Trusted Firmware A. URL https://www.trustedfirmware.org/projects/tf-a/. 11

[16] Android Verified Boot 2.0. URL https://android.googlesource.com/platform/external/avb/+/master/README.md. 14

[17] Overview of EMVCo. URL https://www.emvco.com/about/overview/. 17, 37

[18] Android 7.0 for Developers, . URL https://developer.android.com/about/versions/nougat/android-7.0?hl=en#key_attestation. 15

[19] Android Security Bulletins, . URL https://source.android.com/security/bulletin. 3

[20] Keymaster 3.0 HIDL Specification, . URL https://android.googlesource.com/platform/hardware/interfaces/+/pie-release/keymaster/3.0. 25, 47

[21] Verifying hardware-backed key pairs with Key Attestation, . URL https://developer.android.com/training/articles/security-key-attestation. 23

[22] Android Keystore system, . URL https://developer.android.com/training/articles/keystore. 15

[23] Security Enhancements — Android Open Source Project, . URL https://source.android.com/security/enhancements. 13

[24] Verified Boot — Android Open Source Project, . URL https://source.android.com/security/verifiedboot. 13

[25] Using Binder IPC — Android Open Source Project. URL https://source.android.com/devices/architecture/hidl/binder-ipc. 51

[26] Concise Binary Object Representation. URL https://cbor.io. 24

[27] Concise Data Definition Language. URL https://tools.ietf.org/html/rfc8610. 24

[28] EMV Payment Tokenisation Specification. URL https://www.emvco.com/emv-technologies/payment-tokenisation/. 2

[29] OP-TEE TAs vulnerable to memory corruption bugs. URL https://github.com/teesec-research/optee_examples. 63

[30] Google Mobile Services. URL https://www.android.com/intl/en_in/gms/. 52

[31] GlobalPlatform. URL https://globalplatform.org/. 5, 11

[32] GlobalPlatform Device Technology TEE System Architecture. URL https://globalplatform.org/wp-content/uploads/2017/01/GPD_TEE_SystemArch_v1.1_Public_Release.pdf. vi, 5, 6

[33] GlobalPlatform Specification Library. URL https://globalplatform.org/specs-library/?filter-committee=tee. 11

[34] HIDL — Android Open Source Project. URL https://source.android.com/devices/architecture/hidl. 49

[35] Intel® Trusted eXecution Technology Software Development Guide. URL https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf. 58

[36] JAB Code sources on Github. URL https://github.com/jabcode/jabcode. 28, 33

[37] ARM Versatile Express Juno r1 Development Platform (V2M-Juno r1) Technical Reference Manual. URL https://developer.arm.com/documentation/100122/0100/Introduction/About-the-Versatile-Express-Juno-r1-Development-Platform. 63

[38] dm-verity — The Linux Kernel documentation . URL https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/verity.html. 13

[39] Key and ID Attestation. URL https://source.android.com/security/keystore/attestation. 15

[40] Using KitKat verified boot . URL https://nelenkov.blogspot.com/2014/05/using-kitkat-verified-boot.html. 13

[41] Linux Plumber's Conference 2016 - OP-TEE. URL http://www.linuxplumbersconf.net/2016/ocw/system/presentations/3675/original/LPC%202016%20-%20OP-TEE.pdf. vi, 13

[42] LibTomCrypt 1.17 sources on Github. URL https://github.com/libtom/libtomcrypt/tree/1.17. 48

[43] TCG Mobile Trusted Module Specification. URL https://trustedcomputinggroup.org/resource/mobile-phone-work-group-mobile-trusted-module-specification/. 60

[44] Huawei Mate 7. URL https://consumer.huawei.com/en/support/phones/mate7/. 63

[45] A Clean Slate Approach to Linux Security RISC-V Enclaves. URL https://hex-five.com/wp-content/uploads/MultiZone-Linux-Enclave-White-Paper.pdf. 8

[46] NanoCBOR sources on Github. URL https://github.com/bergzand/NanoCBOR. 48

[47] OMTP Advanced Trusted Environment v1.1, . URL https://www.gsma.com/newsroom/resources/omtp-documents-1-1-omtp-advanced-trusted-environment-omtp-tr1-v1-1 5

[48] Open Mobile Terminal Platform, . URL http://www.omtp.org/. 5

[49] Open Portable Trusted Execution Environment. URL https://www.op-tee.org/. 12, 45

[50] DoD 5200.28-STD - Trusted Computer System Evaluation Criteria (the Orange Book). URL https://csrc.nist.gov/publications/history/dod85.pdf. 8

[51] Source code for our implementation. URL https://github.com/iisc-cssl/secure_payments. 53

[52] Introduction to PCI DSS. URL https://www.cryptomathic.com/news-events/blog/an-introduction-to-pci-dss. 2, 19

[53] Payment Services Directive. URL https://eur-lex.europa.eu/eli/dir/2015/2366/oj. 32

[54] Android Protected Confirmation: Taking transaction security to the next level. URL https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html. 16

[55] KYC Master Direction. URL https://www.rbi.org.in/Scripts/BS_ViewMasDirections.aspx?id=11566. 32

[56] RFC 4226 - HOTP: An HMAC-Based One-Time Password Algorithm, . URL https://tools.ietf.org/html/rfc4226. 61

[57] RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, . URL https://tools.ietf.org/html/rfc5280. 15, 25

[58] RFC 6238 - TOTP: Time-Based One-Time Password Algorithm, . URL https://tools.ietf.org/html/rfc6238. 61

[59] Single euro payments area. URL https://ec.europa.eu/info/business-economy-euro/banking-and-finance/consumer-finance-and-payments/payment-services/single-euro-payments-area-sepa_en. 32

[60] Intel® Software Guard Extensions Programming Reference, . URL https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. 62

[61] Intel® Software Guard Extensions, . URL https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html. 8

[62] ARM DEN 0028C - SMC Calling Convention. URL https://developer.arm.com/documentation/den0028/c/. 10

[63] Linaro Security Working Group on Github, . URL https://github.com/linaro-swg. 45

[64] Keymaster and GateKeeper HAL implementations from Linaro, . URL https://github.com/linaro-swg/kmgk. 47

[65] Android manifest for building OP-TEE in AOSP, . URL https://github.com/linaro-swg/optee_android_manifest. 45

[66] Security work at Linaro, . URL https://www.linaro.org/engineering/core/security/. 45

[67] Apple Platform Security - Secure Enclave, . URL https://support.apple.com/en-in/guide/security/sec59b0b31ff/web. 44

[68] Android 7.0, N Compatibility Definition, Section 9.11, . URL https://source.android.com/compatibility/7.0/android-7.0-cdd#9_11_keys_and_credentials. 15

[69] Stack Overflow. URL https://stackoverflow.com/. 55

[70] TrustZone Protection Controller. URL https://developer.arm.com/documentation/dto0015/a/. 23

[71] Here comes Treble: A modular base for Android. URL https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html. 14

[72] Unique Identification Authority of India. URL https://uidai.gov.in/. 32

[73] Unified Payments Interface. URL https://www.npci.org.in/product-overview/upi-product-overview. 1

[74] Cashless India. URL http://cashlessindia.gov.in/. 1

[75] FreeCharge. URL https://www.freecharge.in/. 1

[76] Google Pay. URL https://pay.google.com/. 1

[77] i.MX53 Quick Start Board. URL https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/i-mx53-quick-start-board:IMX53QSB. 63

[78] mAadhaar. URL https://play.google.com/store/apps/details?id=in.gov.uidai.mAadhaarPlus. 32

[79] mbedTLS 2.6.1 sources on Github. URL https://github.com/ARMmbed/mbedtls/tree/mbedtls-2.6.1. 48

[80] Paytm. URL https://paytm.com/. 1

[81] PCI Security Standards Council. URL https://www.pcisecuritystandards.org/about_us/. 2, 19

[82] PhonePe. URL https://www.phonepe.com/en/. 1

[83] Samsung Pay. URL https://www.samsung.com/global/galaxy/samsung-pay/. 1

[84] Venmo. URL https://venmo.com/. 1

[85] Verifpal Website. URL https://verifpal.com/. 66

[86] Alexandre Adamski, Joffrey Guilbon, and Maxime Peterlin. A deep dive into samsung's trustzone (part 3), Jul 2020. URL https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-3.html. 63

[87] G. Arfaoui, S. Gharout, and J. Traoré. Trusted execution environments: A look under the hood. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 259–266, 2014. doi: 10.1109/MobileCloud.2014.47. 5

[88] ARM Limited. Building a Secure System using TrustZone Technology. URL http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. 3, 8, 9, 45

[89] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 90–102, New York, NY, USA, 2014. Association

for Computing Machinery. ISBN 9781450329576. doi: 10.1145/2660267.2660350. URL https://doi.org/10.1145/2660267.2660350. 57

[90] Rakesh Rajan Beck, Abhishek Vijeev, and Vinod Ganapathy. Privaros: A framework for privacy-compliant delivery drones. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020. 57

[91] Gal Beniamini. Qsee privilege escalation vulnerability and exploit (cve-2015-6639), May 2016. URL https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html. 63

[92] Gal Beniamini. Trust issues: Exploiting trustzone tees, Jul 2017. URL https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html. 63

[93] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating arm trustzone devices in restricted spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 413–425, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342698. doi: 10.1145/2906388.2906390. URL https://doi.org/10.1145/2906388.2906390. 57

[94] Marcel Busch and Kalle Dirsch. Finding 1-day vulnerabilities in trusted applications using selective symbolic execution. 63

[95] Sam Castle, Fahad Pervaiz, Galen Weld, Franziska Roesner, and Richard Anderson. Let's talk money: Evaluating the security challenges of mobile money in the developing world. In *Proceedings of the 7th Annual Symposium on Computing for Development*, ACM DEV '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450346498. doi: 10.1145/3001913.3001919. URL https://doi.org/10.1145/3001913.3001919. 55

[96] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432, 2020. doi: 10.1109/SP40000.2020.00061. 64

[97] Stephen Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *ASPLOS '13*, 2013. 61

[98] F. Corella and K. Lewison. Fundamental security flaws in the 3-d secure 2 cardholder authentication specification. 2019. 56

[99] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. Memory corruption attacks within android tees: A case study based on op-tee. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388337. doi: 10.1145/3407023.3407072. URL https://doi.org/10.1145/3407023.3407072. 63

[100] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *Proceedings of the 2014 Mobile Security Technologies (MoST) workshop*, 2014. 57

[101] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 488–501, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4928-4. doi: 10.1145/3081333.3081349. URL http://doi.acm.org/10.1145/3081333.3081349. 61

[102] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic protocol analysis for the real world. Cryptology ePrint Archive, Report 2019/971, 2019. https://ia.cr/2019/971. 66

[103] Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. On-board credentials with open provisioning. In *Proceedings of the 4th International*

Symposium on Information, Computer, and Communications Security, ASIACCS '09, page 104–115, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583945. doi: 10.1145/1533057.1533074. URL https://doi.org/10.1145/1533057.1533074. 62

[104] Renuka Kumar, Sreesh Kishore, Hao Lu, and Atul Prakash. Security analysis of unified payments interface and payment apps in india. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity20/presentation/kumar. 56

[105] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, page 1–13, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357203. doi: 10.1145/3210240.3210334. URL https://doi.org/10.1145/3210240.3210334. 57

[106] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330244. doi: 10.1145/2637166.2637225. URL https://doi.org/10.1145/2637166.2637225. 58

[107] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. Vbutton: Practical attestation of user-driven operations in mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 28–40, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5720-3. doi: 10.1145/3210240.3210330. URL http://doi.acm.org/10.1145/3210240.3210330. 59

[108] Joshua Lind, Christian Priebe, D. Muthukumaran, D. O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, T. Reiher, David Goltzsche, D. Eyers, Rüdiger

Kapitza, C. Fetzer, and Peter R. Pietzuch. Glamdring: Automatic application partitioning for intel sgx. In *USENIX Annual Technical Conference*, 2017. 62

[109] Dongtao Liu and Landon P. Cox. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, HotMobile '14, pages 7:1–7:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2742-8. doi: 10.1145/2565585.2565591. URL http://doi.acm.org/10.1145/2565585.2565591. 59

[110] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *MobiSys '12*, 2012. 58

[111] Aravind Machiry, E. Gustafson, Chad Spensky, C. Salls, N. Stephens, Ruoyu Wang, A. Bianchi, Yung Ryn Choe, C. Krügel, and G. Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *NDSS*, 2017. 62

[112] Samin Yaseer Mahmud, Akhil Acharya, Benjamin Andow, William Enck, and Bradley Reaves. Cardpliance: PCI DSS Compliance of Android Applications. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity20/presentation/mahmud. 56

[113] Claudio Marforio, Nikolaos Karapanos, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. Secure enrollment and practical migration for mobile trusted execution environments. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, page 93–98, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324915. doi: 10.1145/2516760.2516764. URL https://doi.org/10.1145/2516760.2516764. 62

[114] Steven J. Murdoch and R. Anderson. Verified by visa and mastercard securecode: Or, how not to design authentication. In *Financial Cryptography*, 2010. 56

[115] Z. Ning and F. Zhang. Understanding the security of arm debugging features. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 602–619, 2019. doi: 10.1109/SP.2019.00061. 63

[116] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), January 2019. ISSN 0360-0300. doi: 10.1145/3291047. URL https://doi.org/10.1145/3291047. 9

[117] Sazzadur Rahaman, Gang Wang, and Danfeng Yao. Security certification in payment card industry: Testbeds, measurements, and recommendations. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 481–498, 2019. 55

[118] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R. B. Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 17–32, USA, 2015. USENIX Association. ISBN 9781931971232. 55

[119] Bradley Reaves, Jasmine Bowers, Nolen Scaife, Adam Bates, Arnav Bhartiya, Patrick Traynor, and Kevin R. B. Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications. *ACM Transactions on Privacy and Security*, 20(3), August 2017. ISSN 2471-2566. doi: 10.1145/3092368. URL https://doi.org/10.1145/3092368. 55

[120] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 923–934, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884817. URL http://doi.acm.org/10.1145/2884781.2884817. 61

[121] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 2015. doi: 10.1109/Trustcom.2015.357. 5

[122] A. A. Sani. Schrodintext: Strong protection of sensitive textual content of mobile applications. In *MobiSys*, 2017. 60

[123] Nick Stephens. BEHIND THE PWN OF THE TRUSTZONE. URL https://www.youtube.com/watch?v=jDTXTLkKUcM. Slides - https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose. 63

[124] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 976–988, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813692. URL http://doi.acm.org/10.1145/2810103.2813692. 61

[125] S. Tamrakar, J. Ekberg, and P. Laitinen. On rehoming the electronic id to tees. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 49–56, Aug 2015. doi: 10.1109/Trustcom.2015.356. 62

[126] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M. McCune. Trustworthy execution on mobile devices: What security properties can my mobile platform give me? In *TRUST*, 2012. 3, 5

[127] Wikipedia contributors. 2016 Indian banknote demonetisation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=2016_Indian_banknote_demonetisation, 2020. 1

[128] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. Trustpay: Trusted mobile payment on security enhanced arm trustzone platforms. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 456–462, 2016. 61

[129] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang 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. 59

[130] Kailiang Ying, Priyank Thavai, and Wenliang 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. 59

[131] X. Zheng, L. Yang, G. Shi, and D. Meng. Secure mobile payment employing trusted computing on trustzone enabled platforms. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1944–1950, 2016. 60