DIPARTIMENTO DI INGEGNERIA INFORMATICA
Corso di Laurea Magistrale in Ingegneria Informatica

TEA: enabling THCE on Android devices

Candidato:
Luca Lorrai
Matricola s3353264

Relatore:
Prof. Alessio Merlo

Correlatore:
Dott. Ing. Luca Verderame

Anno Accademico 2015–2016

Indice

1 Introduzione **5**

2 Background **7**

2.1 Smart Card 7

2.2 Near Field Communication 8

2.3 NFC Card Emulation 9

2.3.1 Problematiche di sicurezza e diffusione 10

2.4 NFC Host-based Card Emulation 11

2.4.1 Problematiche relative alla sicurezza 12

2.5 Trusted Execution Enviroment 12

2.6 GlobalPlatform: linee guida per TEE 14

2.6.1 Rich Execution Enviroment API 17

2.6.2 Trusted Execution Enviroment API 19

2.6.3 TEE: funzioni crittografiche 19

2.6.4 TEE: secure storage 20

2.6.5 TEE: socket API 20

2.6.6 TEE: interfaccia utente 21

2.7 File system per le smart card 22

3 Trusted Host-based Card Emulation **24**

3.1 Struttura generale 25

3.2 THCE Engine 26

3.3 THCE Monitor 26

3.4 THCE API 27

3.5 Protocollo di inizializzazione 27

4 **TEA: Trusted host-based card Emulation for Android** **29**
4.1 Introduzione a TEA 29
4.1.1 TEA Library 30
4.1.2 TEA Engine 31
4.2 Specifiche della struttura interna 31
4.3 Prerequisiti tecnici 33
4.4 TEA library: definizione e specifiche 34
4.4.1 Protocollo di inizializzazione 35
4.4.2 Comunicazione con una smart card 39
4.4.3 Considerazioni sulla sicurezza 39
4.5 TEA Engine: definizione e specifiche 40
4.5.1 Protocollo di inizializzazione 41
4.6 Principali differenze tra TEA e THCE 49

5 **Risultati sperimentali** **51**
5.1 Setup 51
5.2 Protocollo di inizializzazione 54

6 **Caso d'uso: pagamento contactless** **57**
6.1 PostePay e protocollo VISA PayWave 58
6.2 Prova di pagamento 61
6.3 Analisi dei risultati 65

7 **Stato dell'arte** **67**

8 **Conclusioni e sviluppi futuri** **70**

Elenco delle figure

2.1 Emulazione smart card con l'utilizzo di SE 9
2.2 Host-based card emulation 11
2.3 Arm Trustzone TEE 14
2.4 Flusso di esecuzione Globalplatform API 16
2.5 Esempio file system di una smart card 22
2.6 Rappresentazione di un Elementary file 23

3.1 Trusted Host-based Card Emulation 25

4.1 Trusted host-based card Emulation for Android 30
4.2 Esempio: Tag-Length-Value 46
4.3 Command APDU 46

5.1 Test TEA: struttura di riferimento 53

6.1 Protocollo PayWave 59

Elenco delle tabelle

4.1 Struttura DF 32
4.2 Struttura EF 32
4.3 Comandi APDU 48

6.1 PayWave processing DOL 60
6.2 Processing DOL 62

Capitolo 1

Introduzione

Negli ultimi anni vi è stato un forte incremento nella diffusione di tecnologie di comunicazione a corto raggio come il Near Field Communication (NFC). In particolare, l'adozione dell'NFC sui dispositivi mobili, come smartphone e tablet, ha consentito lo sviluppo di una miriade di funzionalità aggiuntive. Controllo degli accessi, pagamenti contactless e ticketing sono solo alcuni dei nuovi servizi offerti grazie a tale tecnologia. L'NFC supporta tre diversi modi operativi, il reader/writer, il peer-to-peer e il card emulation (CE). Quest'ultimo consente ad un dispositivo NFC di apparire all'esterno come una tradizionale smart card.

La tecnologia NFC CE coinvolge l'utilizzo di un componente hardware aggiuntivo, chiamato Secure Element (SE), che ha in carico l'emulazione della carta e che offre un alto grado di sicurezza. Nonostante le grandi potenzialità, il mercato non ha però risposto in maniera favorevole a questa implementazione per via degli elevati costi e della difficoltà da parte di sviluppatori di terze parti di ottenere l'accesso ai SE proprietari.

Per superare tali limitazioni, nel 2012 è stata introdotta una nuova architettura chiamata Host-based Card Emulation (HCE) che permette una emulazione completamente software della smart card, eliminando la necessità di un SE. L'utilizzo di HCE ha portato una maggiore facilità di sviluppo che gli ha permesso in breve tempo di avere una forte presa sul mercato.

Tuttavia, eliminando l'ausilio di un hardware dedicato all'emulazione e

fisicamente separato dal resto del sistema, l'HCE ha introdotto nuove e serie problematiche di sicurezza [6].

Per mitigare tali problematiche, pur mantenendo l'accessibilità al sistema, in [1], è stato introdotto il Trusted Host-based Card Emulation (THCE).

Il framework THCE fornisce allo sviluppatore un sistema di gestione della smart card basato su un Trusted Execution Enviroment (TEE), un ambiente sicuro in cui è possibile eseguire applicazioni in modo isolato rispetto al sistema operativo. Questo permette alle applicazioni di utilizzare la tecnologia NFC tramite smartcard in maniera assolutamente trasparente senza dover per questo rinunciare ai livelli di sicurezza garantiti dal Secure Element.

Pur essendo stato definito e teorizzato in [1], non esistono attualmente implementazioni del framework THCE per dispositivi reali.

L'obiettivo di questa tesi è stato progettare, sviluppare e testare il framework THCE per dispositivi mobili. Tale implementazione, chiamata *Trusted host-based card Emulation for Android* (TEA), sfruttando le potenzialità di THCE, implementa un sistema per inizializzare ed utilizzare una smart card in un dispositivo Android basato su TEE.

A ulteriore conferma dell'applicatibilità del framework THCE in contesti reali, il TEA è stato testato con successo con i pagamenti contactless secondo gli standard attualmente presenti sul mercato (come VISA Paywave [18]). In particolare è stato possibile dimostrare empiricamente che i risultati ottenuti con il TEA sono paragonabili a una carta di credito tradizionale.

Capitolo 2

Background

Nel trattare alcune delle argomentazioni presentate nella tesi saranno necessari riferimenti ad alcune tecnologie presenti attualmente sul mercato. In questo capitolo verranno esposte in modo dettagliato gli strumenti principali, che saranno utili per una corretta comprensione delle pagine successive.
Inizialmente sarà introdotta una breve descrizione di cos'è una smart card, successivamente verrà approfondita la tecnologia Near Field Communication, mostrando più in dettaglio i servizi Card Emulation (CE) e Host Card Emulation (HCE), ponendo l'attenzione su quali siano le principali problematiche. Inoltre verranno introdotti l'architettura Trusted Execution Enviroment e le specifiche imposte da Globalplatform, le tecnologie alla base dei framework THCE e TEA.

2.1 Smart Card

Una smart card è un dispositivo hardware simile per dimensioni a una carta di credito al cui interno è integrato un microchip che possiede potenzialità di elaborazione e memorizzazione di dati ad alta sicurezza.
Esistono due categorie di smart card basate sulla modalità di comunicazione, ovvero contact o contactless. Il microchip all'interno della carte fornisce funzionalità di calcolo e memorizzazione dati e per accedervi sono presenti delle interfacce che dipendono dal tipo di carta. Le smartcard comunicano

attrverso terminali esterni, chiamati reader. Alcuni esempi di smart card sono le SIM, utilizzate in telefonia e le carte di credito con cui è possibile identificare l'utente associato alla carta.

2.2 Near Field Communication

Il Near Field Communication o NFC, è una tecnologia definita da un insieme di protocolli di comunicazione che permettono lo scambio di dati tra due dispositivi vicini. Come altri protocolli di prossimità, NFC, utilizza l'induzione magnetica per stabilire la comunicazione tra i dispositivi, creando un canale che supporta una velocità massima di trasmissione di 424 kbit/s.
Per mantenere una comunicazione stabile i dispositivi non possono essere posizionati a più di quattro centimetri di distanza.
All'interno del canale vengono scambiati piccoli pacchetti di dati chiamati Application Protol Data Unit (APDU).
Al momento la tecnologia NFC offre diversi servizi classificabili in tre modi operativi:

- *Reader/writer*
- *Peer To Peer*
- *Card emulation*

In origine, la tecnologia NFC veniva impiegata per interagire con TAG di tipo Radio-Frequency IDentification (RFID), piccoli dispositivi passivi che permettono l'immagazzinamento di dati.
Con la modalità *Reader/writer* è possibile leggere un TAG semplicemente avvicinando un dispositivo NFC che, tramite induzione elettromagnetica, lo alimenta e ne permette la lettura/scrittura dei dati.
La seconda modalità prevista è *Peer to Peer*, dove i dispositivi possono connettersi in NFC tra di loro per scambiare informazioni. Date le limitate velocità supportate da NFC, in questa modalità, i dispositivi in genere scambiano solo le informazioni necessarie per creare un canale sicuro con altre

tecnologie come Bluetooth o Wi-Fi. Per esempio nel sistema operativo Android questa modalità è gestita da Android Beam.
L'ultima delle modalità descritte è la *Card emulation* che consente a un dispositivo NFC di essere riconosciuto da un reader come se fosse una smart card contactless.
Per lo svolgimento della tesi è stata approfondita la modalità *Card emulation*.

2.3 NFC Card Emulation

L'NFC Card Emulation è l'architettura che definisce l'emulazione di carte contactless e prevede l'implementazione di un *Secure Element* (SE) nel proprio sistema. Un SE è una piattaforma pensata per essere resistente alle manomissioni, definita da una combinazione di hardware e software, che insieme garantiscono un ambiente sicuro su cui è possibile eseguire applicazioni. Con l'utilizzo di un SE viene quindi garantito che il codice delle applicazioni e i dati sensibili siano ben protetti e isolati dall'esterno.

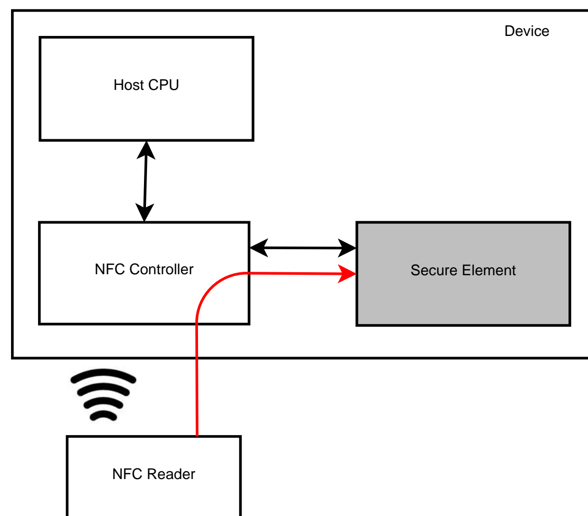


Figura 2.1: Emulazione smart card con l'utilizzo di SE

Come mostrato in figura 2.1, all'interno di un *device*, *NFC Reader* interagisce con *NFC Controller*, l'hardware che contiene l'antenna del dispositivo. *NFC Controller* instrada i pacchetti direttamente al *Secure Element* (SE),

che conterrà la logica necessaria per emulare una smart card. Il SE, essendo fisicamente separato dal dispositivo, permette di resistere a una vasta gamma di attacchi software e manomissioni hardware.
Esistono tre tipologie di implementazione differenti di SE:

- Universal Integrated Circuit Card (UICC)
- Embedded SE
- MicroSD

Le versioni UICC e microSD sono soluzioni che possono essere inserite o rimosse da un dispositivo ospite, viceversa embedded SE è integrato nella scheda madre di un device e non può essere rimosso. Uno scenario tipico per l'uso dell'emulazione di smart card con SE è l'esecuzione di una applicazione sicura, che interagendo con l'esterno attraverso una interfaccia grafica, può richiede l'inserimento di un PIN che gli permette di accedere a una zona protetta, dove verrà quindi eseguita la logica della transazione.

2.3.1 Problematiche di sicurezza e diffusione

Anche se l'emulazione di una smart card attraverso un SE garantisce lo stesso livello di affidabilità e sicurezza offerti da una smart card contactless, nel tempo sono stati tentati alcuni attacchi. Uno dei principali è il relay attacks [11]. Un relay attack è una variante dell'attacco Man-in-the-middle, in cui è possibile per un terzo agente inserirsi in una comunicazione tra due dispositivi. Per risolvere il problema una delle soluzioni utilizzate da ISO/IEC 14443-3 [13] è di imporre un timeout alle transazioni. Tuttavia, come descritto in [14], anche questa soluzione non garantisce una completa immunità a questo tipo di attachi. Recentemente sono apparsi anche altri tipi di attacco basati sull'utilizzo di una combinazione di Wi-Fi, 3G e Bluetooth.
L'utilizzo di un SE è al momento la più affidabile e sicura soluzione per l'emulazione di smart card. Pur garantendo un ottimo livello di sicurezza, attualmente, sono pochi i dispositivi che lo implementano.
Per poter implementare in un device un SE è necessario ottenere accordi

con i principali produttori, che offrono i loro servizi a costi molto elevati. I costi elevati e la frammentazione, hanno portato a una scarsa diffusione sul mercato di questa tecnologia.

2.4 NFC Host-based Card Emulation

Nel 2012 una azienda americana chiamata SimplyTapp, fondata da Doug Yeager e Ted Fifelski, ha definito un nuovo servizio chiamato Host-based Card Emulation (HCE).
HCE definisce una evoluzione del servizio Card Emulation, eliminando la necessità di un SE.

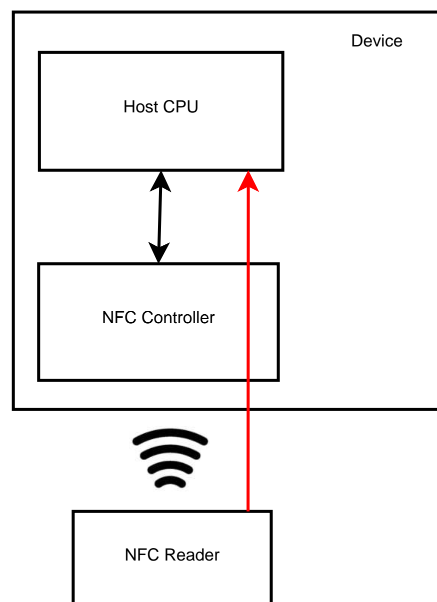


Figura 2.2: Host-based card emulation

Come si vede in figura 2.2, la transazione NFC e l'emulazione della carta sono gestiti direttamente da una logica software, che viene eseguita all'interno di *Host CPU*, il processore centrale del *device*. Con questa soluzione non si ha la necessità di un SE e il *NFC Controller* permette un accesso diretto dei dati al processore.
Risulta quindi una soluzione più versatile che lascia spazio a una maggiore

personalizzazione delle applicazioni, ma la logica della sicurezza deve essere implementata direttamente dallo sviluppatore.
HCE è presente su molti sistemi operativi mobile, in Android è stato introdotto dalla versione KitKat (release 4.4.0) con il nome di *HCE Service*. Attualmente questo approccio ha permesso al servizio di emulazione di smart card di essere integrato all'interno di device compatibili con un hardware NFC grazie a un semplice aggiornamento software.

2.4.1 Problematiche relative alla sicurezza

La facilità di implementazione ha reso questa soluzione molto popolare, e nel giro di poco tempo molte delle piattaforme disponibili sul mercato hanno iniziato a implementare il supporto ad HCE.

Tuttavia, a causa della natura stessa dell'architettura, L'HCE soffre di diverse problematiche di sicurezza.

Se nel caso di Card Emulation, il software era eseguito in modo sicuro all'interno di una ambiente protetto e separato, ora la sicurezza dell'implementazione è dipendente dalle abilità dello sviluppatore. La logica della transazione e le funzioni di sicurezza vengono infatti eseguite su un sistema operativo che potrebbe essere stato compromesso o contenere vulnerabilità di sicurezza. Ad esempio, in [6] viene mostrato come nel sistema operativo Android, in presenza di root del dispositivo o in generale con un malware, i dati di HCE siano accessibili, permettendo il furto delle informazioni sensibili oggetto della transazione (es. il PIN della carta di credito).

A causa di tali considerazioni, l'uso del HCE è stato limitato a scarsa sensibilità di sicurezza.

2.5 Trusted Execution Enviroment

Il Trusted Execution Enviroment (TEE) è una architettura hardware e software che implementa elevate caratteristiche di sicurezza la cui standardizzazione è gestita da Globalplatform.
In un sistema tradizionale, il sistema operativo e le applicazioni, sono ese-

guite in un solo contesto chiamato Rich Execution Enviroment (REE). Con l'introduzione di TEE è stata definito un nuovo contesto di esecuzione chiamato Trusted Execution Enviroment.
All'interno della zona sicura possono essere eseguite solo delle applicazioni che implementano determinate caratteristiche di sicurezza, chiamate Trusted application (TA), con cui è possibile comunicare attraverso delle API specifiche.
A differenza di un SE il TEE non è garantito per essere sicuro contro possibili attacchi hardware, è comunque possibile definire, se il dispositivo ne è provvisto, alcune delle componenti direttamente dentro un SE. Questa possibilità permette di aumentare le capacità di sicurezza per il salvataggio dei dati.
Per come è definito, il TEE permette un alto livello di protezione da tutti gli attacchi che vengono eseguiti nel sistema operativo principale, che sia esso Linux, Android, Windows o altri.
Vasudevan in [25] ha elencato le principali caratteristiche di sicurezza che il TEE è in grado di soddisfare:

- *Isolated Execution*, garantisce che le applicazioni siano eseguite in un contesto isolato e protetto.
- *Secure storage* protegge il salvataggio dei dati sensibili, come ad esempio chiavi di sicurezza.
- *Remote attestation*, assicura a un utente remoto di comunicare con la giusta TA all'interno del device.
- *Secure provisioning* permette una comunicazione in remoto con una specifica TA, mantenendo integrità e confidenzialità dei dati trasmessi.
- *Trusted Path*, il trasferimento dei dati da e verso il TEE avviene attraverso un canale che è protetto da manomissioni e intercettazioni.

Essendo aumentate in modo considerevole le richieste di affidabilità e sicurezza, molti chip manufacturers hanno iniziato a definire le proprie versioni di TEE. Tra le tante proposte presenti sul mercato, una delle più diffuse è

l'interpretazione di ARM chiamata Trustzone. Trustzone è attualmente implementata in tutte le versioni di processori ARM a partire dalla versione v7 in avanti.
Vediamo ora uno schema che rappresenta l'implementazione di ARM:

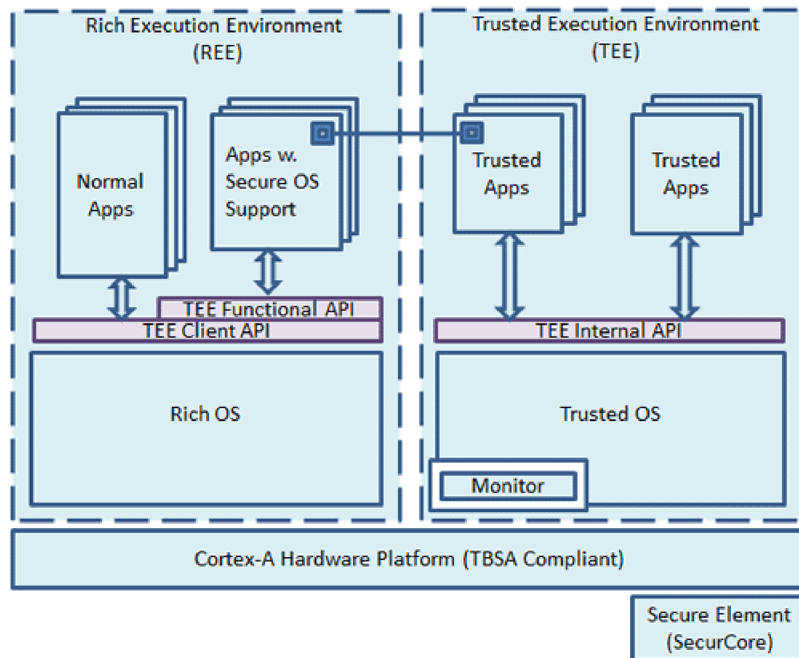


Figura 2.3: Arm Trustzone TEE

Come si può notare nella figura 2.3, il sistema è rappresentato con una netta separazione tra i due mondi definiti prima. Sul mercato, oltre a Trustzone, sono presenti molte implementazioni di TEE, esistono infatti le versioni per processori intel, samsung e Texas Instrument. Nello svolgimento della tesi sarà comunque presa in considerazione la sola proposta di ARM, che risulta la più diffusa sul mercato.

2.6 GlobalPlatform: linee guida per TEE

GlobalPlatform (GP) è una associazione no-profit che dedica il proprio lavoro alla definizione e lo sviluppo di standard nel mondo delle applicazioni sviluppate su chip. Tra le specifiche rilasciate da GP è possibile trovare delle linee

guida che suggeriscono una definizione per le applicazioni nel mondo delle smart cart. La naturale evoluzione di questi servizi è stata la definizione di standard per la gestione del TEE.
Un sistema compatibile con tali specifiche avrà una struttura tale da permettere ad una applicazione nel mondo non sicuro di interagire con una TA all'interno del TEE.
La divisione dei contesti di esecuzione ha reso necessario l'utilizzo di API specifiche, supportate da un driver di sistema dedicato.
Nella gestione del flusso dei dati tra il sistema e la zona sicura, GP ha identificato una struttura che prevede la creazione di un contesto, al cui interno possono essere create delle sessioni. Successivamente alla creazione di una sessione è possibile iniziare la comunicazione con una TA.
Con questa particolare gestione viene garantito un canale diretto e sicuro tra i due contesti dentro il quale è possibile richiamare i comandi definiti all'interno di una TA.
Per la creazione del contesto e di una sessione sono previste due funzioni separate chiamate *InitializeContext* e *OpenSession*.
L'effettiva esecuzione di un comando è possibile solo attraverso una funzione chiamata *InvokeCommand*, eseguibile solo all'interno di una sessione attiva.
Terminata la comunicazione, è necessaria la chiusura del canale utilizzando le funzioni *CloseSession* e *CloseContext*.
Per avere accesso alle funzioni di una TA è necessario rispettare l'ordine di esecuzione dei comandi come mostrato in figura 2.4.

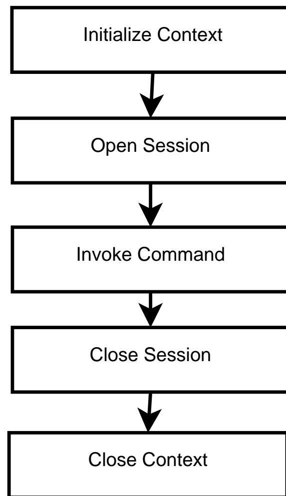


Figura 2.4: Flusso di esecuzione Globalplatform API

Nella definizione delle proprie funzioni GP ha definito una struttura proprietaria, dove le funzioni interagiscono solo attraverso i parametri passati alla funzione.
Il valore di ritorno sarà sempre un valore di tipo *TEEC_Result* o *TEE_Result* che identificano lo stato di esecuzione della chiamata.
Per verificare la corretta esecuzione di un comando sarà quindi necessario analizzare il valore di ritorno, che nel caso di corretta esecuzione sarà *TEEC_SUCCESS* o *TEE_SUCCESS*.
Se l'esecuzione di un comando non è andata a buon fine il valore di ritorno identificherà il particolare tipo di errore.
Per comprendere al meglio il funzionamento della architettura introduciamo un esempio. Supponiamo di avere una TA all'interno del TEE che definisce due funzioni, `funzione_a` e `funzione_b`. Se una funzione nel REE vuole eseguire uno dei comandi, deve prima utilizzare *InitializeContext* e *OpenSession*. Dopo l'esecuzione delle due chiamate è possibile eseguire la funzione *InvokeCommand* utilizzando come parametro il riferimento a `funzione_a` o `funzione_b`. In questo modo la TA riceve il comando per eseguire una delle due funzioni previste. Quando ha terminato è possibile richiamare *CloseSession* e *CloseContext* per chiudere la comunicazione.
Vediamo più in dettaglio nelle sezioni successive le funzioni per il REE e il

TEE.

2.6.1 Rich Execution Enviroment API

Iniziamo ad analizzare le API dedicate al REE.

```
1 TEEC_Result TEEC_InitializeContext(
2       const char*   name,
3       TEEC_Context* context)
```

Listing 2.1: Prototipo di TEEC_InitializeContext

La funzione in questione permette di inizializzare un contesto, all'interno del quale è possibile definire le sessioni.
La variabile *context*, di tipo *TEEC_ Context* identifica idealmente l'apertura di un canale tra il REE e il TEE.

```
1 TEEC_Result TEEC_OpenSession (
2       TEEC_Context*     context,
3       TEEC_Session*     session,
4       const TEEC_UUID* destination,
5       uint32_t          connectionMethod,
6       const void*       connectionData,
7       TEEC_Operation*   operation,
8       uint32_t*         returnOrigin)
```

Listing 2.2: Prototipo di TEEC_OpenSession

La funzione *TEEC_ OpenSession* riceve in ingresso la variabile *context* istanziata nella funzione precedente, come accennato è necessario seguire l'ordine di esecuzione delle funzioni per evitare errori.
Un altro parametro importante è *session*, dopo l'esecuzione della funzione conterrà un valore che ne identifica la sessione in corso.
L'ultimo parametro da osservare è *destination*, sarà infatti la stringa contenuta in questa variabile a identificare una TA specifica.

```
1 TEEC_Result TEEC_InvokeCommand(
2 TEEC_Session*     session,
```

```
3 uint32_t           commandID,
4 TEEC_Operation*    operation,
5 uint32_t*          returnOrigin)
```

Listing 2.3: Prototipo di TEEC_InvokeCommand

Iniziata una sessione si è ora in grado di comunicare con una applicazione nel mondo sicuro. La gestione delle chiamate avviene attraverso la funzione *TEEC_InvokeCommand*, che permette di richiamare il comando identificato da *commandID*, che sarà una funzione definita all'interno della TA.
Per la comunicazione tra le applicazioni Globalplatform prevede l'utilizzo di quattro parametri. I parametri possono appartenere a solo uno dei formati predefiniti e devono rispettare il formato richiesto dalla funzione all'interno della TA.
Vediamo i possibili valori definiti per i parametri di *InvokeCommand*:

- TEEC_NONE
- TEEC_VALUE_INPUT /OUTPUT
- TEEC_VALUE_INOUT
- TEEC_MEMREF_TEMP_INPUT /OUTPUT
- TEEC_MEMREF_TEMP_INOUT
- TEEC_MEMREF_WHOLE
- TEEC_MEMREF_PARTIAL_INPUT /OUTPUT
- TEEC_MEMREF_PARTIAL_INOUT

Se l'applicazione all'interno del TEE riceve dei parametri non compatibili con il comando richiesto, verrà bloccata subito l'esecuzione della funzione. Si può notare come la struttura ideata da Globalplatform sia molto vincolata ma allo stesso tempo molto completa.

2.6.2 Trusted Execution Enviroment API

Vediamo ora all'interno del mondo sicuro quali sono le chiamate fondamentali da implementare. Per mantenere una struttura coerente, le funzioni ideate da Globalplatform sono esattamente analoghe a quelle viste nel mondo non sicuro. Dovranno essere presenti per un corretto funzionamento le funzioni viste in precedenza.

- TEE_InitializeContext
- TEE_OpenSession
- TEE_InvokeCommand
- TEE_CloseSession
- TEE_CloseContext

.

TEE_OpenSession è legata alla funzione *TEEC_OpenSession*, quando viene chiamata la funzione nel REE, viene aperta una sessione in riferimento alla TA selezionata nel TEE.
Utilizzando *TEE_InvokeCommand* è possibile richiamare la logica associata al comando invocato nel REE.
Per chiudere la connessione sono definite *TEE_CloseSession* e successivamente *TEE_CloseContext*.

2.6.3 TEE: funzioni crittografiche

Uno dei scenari d'uso maggiore del TEE è per l'esecuzione di funzioni crittografiche definite in una libreria presente nel mondo sicuro. Queste funzioni sono utilizzabili da una TA e comprendono le API basilari per poter eseguire gli algoritmi crittografici più comuni. Le modalità di utilizzo e la lista degli algoritmi supportati è presente in [2].
Le specifiche definiscono le funzioni necessarie per inizializzare un *TEE_ObjectHandle*, un gestore che permette di eseguire le funzioni crittografiche. Le chiamate

principali per completare l'esecuzione di un algoritmo crittografico sono simili e quelle definite in altri linguaggi.
Per prima cosa viene chiamata la funzione *TEE_ CipherInit*, questa inizializza l'handle con la chiave di crittografia e le informazioni sull'algoritmo da eseguire nelle operazioni seguenti. Successivamente è necessario chiamare *TEE_ CipherUpdate* che assegna il vettore corrispondente al testo in chiaro e al blocco cifrato in uscita. Per completare la crittografia si esegue *TEE_ CipherDoFinal*, che esegue l'algoritmo crittografico e restituisce il contenuto cifrato.

2.6.4 TEE: secure storage

Un altro elemento chiave che viene definito da GlobalPlatform è lo storage di informazioni all'interno di TEE. Il salvataggio di dati permanenti è gestita dall'utilizzo di Persistent Object(PO), che vengono salvati nello spazio in memoria dedicato alla TA. Quando viene istanziata una TA gli viene assegnato uno spazio di indirizzamento dedicato, non è infatti possibile per altre TA accedere a questa area della memoria.
Un PO è una struttura che può contenere qualunque tipo di dato poiché viene salvato in memoria come uno stream di dati. É possibile salvare per esempio chiavi crittografiche, certificati e altre strutture dati.
Interagire con il contenuto di un PO è possibile solo grazie a un *Object Handle*, simile a quello utilizzato per le funzioni di crittografia. Nell'apertura di un PO si possono scegliere i parametri con cui si può interagire con i dati, si può per esempio bloccare la gestione a un solo handle, oppure condividere con più handle o ancora, definire le modalità di acceso ai dati.

2.6.5 TEE: socket API

Molte delle implementazioni che utilizzano il TEE hanno necessità di connettersi a un server remoto. Normalmente la comunicazione avviene utilizzando le API definite in precedenza per poter instradare i pacchetti all'esterno della TA. In questo modo sarà una applicazione nel REE a gestire la comunicazio-

ne con il server, utilizzando un protocollo a scelta tra quelli supportati del sistema operativo.
Per l'utilizzo dei protocolli di rete più comuni, GP ha comunque definito un insieme di API basilari raccolte in **TEE_Socket_API** [22]. Le API permettono di utilizzare i protocolli di rete UDP e TCP e sono definite con una struttura POSIX. Le chiamate supportate sono le stesse offerte dai socket ovvero: *open, close, send, recv.*
Per permettere una definizione più personale delle chiamate, GP ha previsto una struttura a stack delle API. É possibile definire i socket per più protocolli di rete, che possono essere impilati, per aggiungere funzionalità differenti. Questo comportamento è analogo alla pila ISO/OSI con cui sono definiti i livelli di rete.
Per permettere questa implementazione, oltre alle chiamate mostrate prima vi è un ulteriore comando, chiamato *ioctl.* Grazie a ioctl è possibile selezionare un livello di rete e definire un buffer che può essere usato per trasferire i dati tra i livelli. L'ultima delle funzionalità offerte da TEE_Socket_API è la funzione *error*,in cui vengono riportati tutti gli errori definiti per i protocolli utilizzati.

2.6.6 TEE: interfaccia utente

Nella definizione di una applicazione può essere necessario immettere delle credenziali o un semplice PIN. L'esecuzione di una interfaccia grafica nel REE potrebbe esporre il sistema a possibili attacchi. Per ridurre al minimo questa eventualità, per TEE, è prevista una interfaccia utente completa, che viene eseguita all'interno della zona sicura [23]. Le API prevedono tre modalità predefinite, che comprendono: l'inserimento di un PIN, l'inserimento di credenziali (User e Password) e una pagina per eventuali avvisi, come ad esempio la conferma del corretto inserimento.
Per ogni modalità è possibile utilizzare sei differenti bottoni: CORRECTION, OK, CANCEL, VALIDATE, PREVIOUS, e NEXT.
Per personalizzare le schermate è possibile impostare una immagine di sfondo in formato PNG. L'immagine può essere in bianco e nero con una profondità

di 8 bit, e in formato RGB con una profondità di 24 bit.
Rispettando le specifiche appena descritte è possibile definire una interfaccia completa e semplice da usare con cui è possibile interagire con un utente in modo sicuro.
Per ulteriori dettagli è possibile consultare [23], dove è presente un esempio completo che mostra i passaggi per richiamare l'interfaccia da TEE.

2.7 File system per le smart card

Le specifiche definite per le smart card contactless sono racchiuse all'interno dello standard ISO/IEC 14443 [16], dove sono specificate le caratteristiche del livello fisico. Le funzionalità interne, come i comandi e la definizione del file system richiamano le funzionalità delle smart card su chip che sono definite in ISO 7816-4 [17].
In questa sezione vediamo in dettaglio le specifiche per il file system.
I dati in una smart card presentano una struttura ad albero del tutto simile a un file system unix tradizionale, nella quale sono presenti file e cartelle. All'interno è possibile identificare 2 tipologie di file, chiamati *dedicated file* (DF) e *elementary file* (EF). Per ricondurci a una struttura unix-like possiamo identificare i DF come le cartelle e gli EF come file veri e propri.
Il file radice presente in una smart card è un particolare tipo di DF che viene chiamato *master file* (MF) ed è il punto di partenza di ogni accensione o reset di una carta.

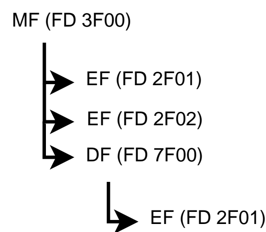


Figura 2.5: Esempio file system di una smart card

Nella figura 2.5 è rappresentato un esempio di file system, dove è visibile la struttura ad albero.

Per navigare nei file di una carta è necessario identificare un percorso selezionando un DF, al cui interno possono essere memorizzati altri DF o EF. Sempre in riferimento allo standard, è possibile la selezione di un file in due modalità:

- Dedicated File Name (DF_Name), composto da 16 byte.
- File Identifier (FID), composto da 2 byte.

Nella struttura delle carte gli EF sono gli unici elementi che possono contenere dei dati come ad esempio chiavi, certificati, dati singoli, oppure stringhe.
In generale sono previste quattro strutture di riferimento per un EF:

- *Transparent file*
- *Record a dimensione fissa*
- *Record a dimensione variabile*
- *Record circolari*

I *transparent file* sono generalmente i contenitori per le chiavi e i certificati, ovviamente è possibile comunque salvare al loro interno ogni tipo di dato. I *record a dimensione fissa*, *record a dimensione variabile* e i *record circolari* sono invece dedicati ad altre operazioni.

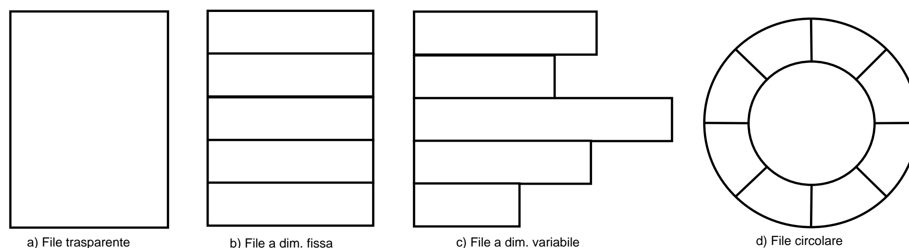


Figura 2.6: Rappresentazione di un Elementary file

Capitolo 3

Trusted Host-based Card Emulation

Il principale obiettivo di Trusted Host-based Card Emulation (THCE) è garantire l'utilizzo della tecnologia NFC Card Emulation in modo completamente trasparente senza dover per questo rinunciare ai livelli di sicurezza garantiti dal Secure Element.

Il framework THCE, utilizza l'architettura TEE, che viene nativamente implementata nella maggior parte dei dispositivi.
I principali punti di forza del framework secondo [1] sono:

1. Sistema aperto: garantisce l'utilizzo del servizio di emulazione di carte senza dover interpellare device manufatcurer e operatori di rete.

2. Minimal invasiveness: minimizza la modifica e la personalizzazione di un device sfruttando le tecnologie già presenti.

3. Secure management and data provisioning: gestione dedicata e sicura dei dati di ogni applicazione che sfrutta THCE.

4. Client authentication: supporto ad autenticazione e certificazione di un device.

5. API di alto livello: ricco set di API per permettere l'utilizzo delle funzioni offerte da THCE.

3.1 Struttura generale

Per la definizione di THCE è necessario individuare i contesti di esecuzione delle applicazioni, proprio come fatto per la definizione di TEE.
Il primo contesto è chiamato Rich Execution Enviroment (REE), dove viene eseguito il sistema operativo non sicuro che viene opportunamente separato dal secondo contesto, Trusted Execution Enviroment (TEE) che implementa le caratteristiche di sicurezza.
La struttura completa viene definita come in figura 3.1.

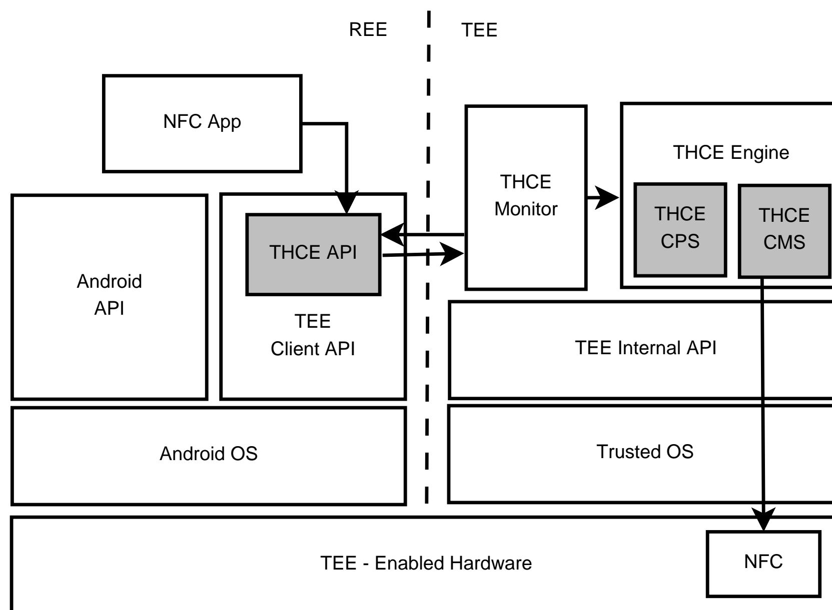


Figura 3.1: Trusted Host-based Card Emulation

All'interno del TEE sono presenti *THCE engine*, la componente fondamentale del framework e il *THCE monitor* che gestisce le comunicazioni con il rich execution environment.
Nel mondo non sicuro è presente *THCE API*, un modulo che comprende tutte le API necessarie alle applicazioni per interfacciarsi con THCE.
Il seguito di questo capitolo descriverà brevemente ognuno di questi moduli.

3.2 THCE Engine

La componente fondamentale del framework risiede nel *THCE Engine*. Essa si occupa interamente delle comunicazioni con l'hardware NFC e con un server esterno attraverso la rete. L'Engine ha il compito di gestire l'emulazione di carte e offre delle funzionalità simili a quelle offerte da HCE in Android. All'interno del THCE Engine sono presenti due moduli che si occupano di elaborare le APDU in arrivo da NFC e della gestione delle credenziali e dei dati sensibili.

- **Credential Manager Service**
  Il primo modulo è il Credential Manager Service (CMS), esso gestisce il salvataggio delle credenziali delle carte e lo storage dei dati sensibili. Le informazioni sensibili saranno utilizzate solo dal CMS e non lasceranno mai la zona sicura. Questo approccio è analogo a quello di un SE, dove le informazioni sensibili non sono mai esposte all'esterno. All'interno del CMS possono essere gestite contemporaneamente molte smartcard, garantendo allo stesso tempo un isolamento delle informazioni sensibili tra le varie applicazioni.

- **Credential Provisioning Service**
  Il Credential Provisioning Service (CPS) è in grado di offrire le funzionalità OTA (Over the air) per inserire e aggiornare i dati delle smart card. Per garantire queste funzionalità è in grado di instaurare un canale sicuro tra il TEE e il Server con cui inizia la comunicazione.
  Dovrà quindi gestire l'intera procedura per l'inizializzazione di una carta e il salvataggio dei dati all'interno del TEE mantenendo la confidenzialità della comunicazione.

3.3 THCE Monitor

Un'altra componente di grande importanza è il *THCE monitor*. In generale tutte le applicazioni provenienti dal REE con cui si inizia una comunicazione

sono da considerarsi non sicure. Per avere certezza della genuinità dell'applicazione sono definite delle policy che devono essere rispettate. Le policy verranno organizzate come una insieme di regole e devono essere verificate a ogni comunicazione.

3.4 THCE API

Per poter raggiungere una TA all'interno del TEE non è possibile accedervi direttamente, è necessario utilizzare delle chiamate dedicate definite in un driver nel REE. Il driver per il THCE contiene tutte le API che permettono di gestire le situazioni più comuni nella comunicazione con le smart card e per l'inizializzazione del protocollo. Le chiamate sono definite in modo che lo scambio di informazioni non contenga mai dati sensibili.
Nella definizione del framework l'unico che avrà accesso ai dati sensibili è il THCE Engine.

3.5 Protocollo di inizializzazione

Nell'idea generale del THCE una smart card prima di poter essere utilizzata deve essere inizializzata e verificata. Nella fase di inizializzazione viene identificato il dispositivo in modo univoco e solo successivamente verranno inviate le informazioni necessarie alla definizione della smart card. Per questa fase viene utilizzato una variante del protocollo per la mutua autenticazione StS(Station-to-Station), basato sul noto protocollo Diffie-Hellman per lo scambio di chiavi [21].
In questa sezione saranno presentate le fasi per il protocollo di inizializzazione, i due enti in gioco sono un dispositivo con implementato THCE e un server dedicato all'inizializzazione dei servizi della smart card.
Definiamo D come il TEE presente in un dispositivo e S il Server. S avrà a disposizione le informazioni necessarie (Payload) per uno specifico utente che chiameremo U. Il server S è in grado di autenticare U grazie all'hash della sua password personale H(x). D ha il suo certificato a chiave pubblica $\mathrm{Cert_d}$

e S avrà l'analogo Cert_s.
Vediamo ora le fasi del protocollo pensato per l'inizializzazione:

$$(1)D \rightarrow S : D, \alpha^{rd} \tag{3.1}$$

Sia G un insieme in cui è computazionalmente difficile risolvere il problema di Diffie-Helmann, sia α un multiplo dello stesso ordine di G, D sceglie un valore r_d e calcola α^{r_d}.

$$(2)S \rightarrow D : \alpha^{rs}, Enc_\text{k}\{Cert_\text{s}, Sgn_\text{s}\{S, D, \alpha^{r_\text{s}}, \alpha^{r_\text{d}}\}\} \tag{3.2}$$

S in modo analogo sceglie un valore r_s, calcola α^{r_s} e deriva la chiave simmetrica $\text{k} = (\alpha^{r_\text{d}})^{r_\text{s}}$. La chiave k appena trovata servirà per crittare i dati che verranno scambiati successivamente. S invia quindi α^{r_s} e un pacchetto crittato contenente il certificato Cert_s e la firma del pacchetto composto dalle identità dei due enti insieme ad α^{r_s} e α^{r_d}. Il Cert_s viene passato crittato con la chiave k, il protocollo è resistente quindi agli attacchi di sostituzione della chiave pubblica.

$$(3)D \rightarrow S : Enc_\text{k}\{Cert_\text{d}, Sgn_\text{d}\{A, U, h(x), D, S, \alpha^{r_\text{d}}, \alpha^{r_\text{s}}\}\} \tag{3.3}$$

Ricevuto il messaggio D calcola k, decritta il messaggio, e invia al server S un nuovo messaggio cifrato contenente il proprio certificato Cert_d e la firma di un pacchetto contenente le informazioni per l'inizializzazione (A), l'identità dell'utente (U), l'hash della password dell'utente (h(x)) e le informazioni di identità del server e del TEE con le chiavi pubbliche. In questa fase del protocollo il server S ha tutte le informazioni per identificare l'utente U.

$$(4)S \rightarrow D : Enc_\text{k}\{Sgn_\text{s}\{S, D, Payload\} \tag{3.4}$$

Nell'ultima fase il server S invia a D un pacchetto cifrato contenente le informazioni di identità e un Payload firmati. Nel Payload saranno inserite le informazioni utili per inizializzare la carta sul dispositivo.

Capitolo 4

TEA: Trusted host-based card Emulation for Android

L'obiettivo principale di questa tesi è stato studiare e implementare un framework per dispositivi Android dotati di TEE, che fosse compatibile con le specifiche THCE discusse nei capitoli precedenti. Da questo lavoro è nato **Trusted host-based card Emulation for Android** (TEA).
Nel prosieguo di questo capitolo verranno descritti TEA e le principali scelte implementative che hanno portato alla definizione del framework.

4.1 Introduzione a TEA

In questo capitolo sarà espresso più in dettaglio il framework TEA e di come è stata pensata la struttura interna.
Alcune delle scelte intraprese nella definizione del framework sono dovute alla particolare struttura prevista da Globalplatform.
Tutte le API fornite dal TEE, pur garantendo un elevato standard di sicurezza, offrono severe limitazioni nella loro personalizzazione. Per questo motivo si è deciso di rimanere legati allo standard GP e di limitare lo sviluppo di componenti legati all'hardware in uso.
La figura 4.1 schematizza l'architettura TEA.

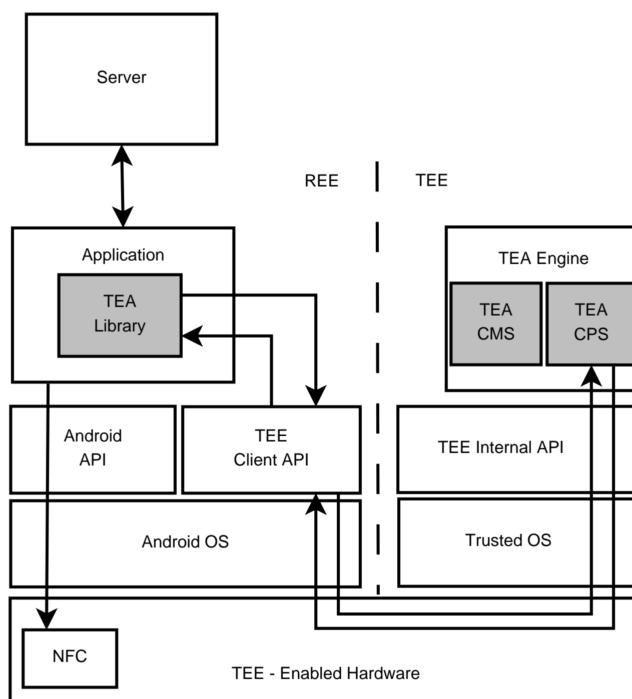


Figura 4.1: Trusted host-based card Emulation for Android

Le componenti necessarie a definire il framework in un dispositivo Android sono principalmente due, TEA Library e TEA Engine, che insieme definiscono un legame tra il REE e il TEE.

4.1.1 TEA Library

TEA Library è la libreria che mette in contatto una applicazione Android con il mondo sicuro.
Dentro TEA Library saranno integrate le funzionalità per la creazione di una comunicazione con la TA all'interno del TEE.
Per comunicare con l'engine del framework, TEA Library rende disponibili le funzioni con cui è possibile inizializzare e interagire una smart card emulata.
Per definizione, l'unico modo con cui è possibile comunicare con l'engine è attraverso l'utilizzo di questa libreria.

4.1.2 TEA Engine

Il cuore del framework è rappresentato da TEA Engine, la TA all'interno del TEE, che raccoglie le chiamate in arrivo dalla libreria TEA Library ed elabora i comandi per eseguire le funzionalità richieste.
All'interno dell'engine è possibile identificare due moduli separati:

- **TEA Credential Provisioning Service (CPS)**
  TEA CPS garantisce le funzionalità OTA (over-the-air) del framework. Lavorando in comunione con il protocollo di inizializzazione permette di creare un canale sicuro con un server, per inizializzare le componenti necessarie di una smart card.

- **TEA Credential Manager Service (TEA CMS)**
  TEA CMS ha il compito di garantire un salvataggio sicuro delle informazioni all'interno del TEE. Utilizzando le funzionalità offerte da Globalplatform gestisce l'interazione della TA con il REE, permettendo scambio di dati per ogni singola applicazione presente nella zona sicura.

4.2 Specifiche della struttura interna

La gestione dei file nella Trusted Application è stata pensata il più possibile conforme alle specifiche definite per le smart card. Intrapresa la struttura del file system come mostrato nei capitoli introduttivi, vediamo come è gestito l'immagazzinamento di dati per il framework, dove è i file sono chiamati Persistent Object (PO).
I PO sono particolari contenitori di file che vengono automaticamente gestiti dai servizi nel TEE e permettono il salvataggio di array di dati. Per interagire con essi sono necessari degli handle, strutture particolari che permettono una lettura e/o scrittura del contenuto di un PO in accordo con delle policy definite da un flag.
Per TEA si è pensato di memorizzare ogni elemento presente nella carta, che sia EF o DF, all'interno di un PO. Questa scelta è stata pensata poiché, i

file in una smart card, possono essere richiamati direttamente conoscendo il *DF_ Name*, caratteristica analoga ai PO, che sono richiamabili direttamente conoscendo il nome con cui sono stati salvati.
Per identificare il DF attualmente selezionato si è pensato di istanziare una variabile globale alla sessione chiamata *object_ selected*. Con *object_ selected* è possibile per una funzione fare riferimento all'EF corretto. La corretta identificazione di un EF è fondamentale, perché per lo standard corrente, più EF possono avere lo stesso *FID* se sono definiti in un DF differente. DF e EF presentano strutture interne differenti ma vengono salvati come un array di byte per via della particolare struttura di un PO, è quindi importante poter discriminare tra i due.
Nelle tabelle 4.1 e 4.2 viene mostrata la struttura di riferimento con cui verranno salvati i dati all'interno di un PO.

valore	# byte
F_ID	2
DF_name	16
Flag	1
Length	4
Value	Length

Tabella 4.1: Struttura DF

valore	# byte
F_ID	2
DF_name	16
Type	1
Flag	1
Length	4
Pos	4
Record_size	1
Rec_num	4
Value	Record * Rec_num

Tabella 4.2: Struttura EF

Il campo *Flag* permette di identificare i diritti di lettura e scrittura del file. Al momento dell'esecuzione di un comando per la lettura o la modifica dei file sarà passato un parametro alla funzione relativa che identifica il permesso attualmente disponibile. Se il parametro non rispetta i permessi definiti in *Flag*, la funzione stoppa la sua esecuzione.
La struttura per un DF è pensata per permettere il salvataggio dei riferimenti ad altri DF o EF.
Gli EF contengono in aggiunta il parametro *Type* che permettere di identificare il tipo di struttura interna, vi è poi la variabile *Pos* che identifica all'interno del file a quale record si fa riferimento.

4.3 Prerequisiti tecnici

Nella definizione del codice si è reso necessario utilizzare degli algoritmi per la crittografia e la firma di dati. Il TEE implementa la libreria crypto all'interno della zona sicura e supporta una grande varietà di algoritmi crittografici. Nel rimanere conformi al protocollo definito in THCE, per TEA sono stati utilizzati algoritmi per la cifratura e la mutua autenticazione. In particolare per la mutua autenticazione è stato utilizzato l'algoritmo Diffie-Hellman, nativamente supportato da TEE.
Nelle specifiche rilasciate da GP per l'algoritmo Diffie-Hellman è supportata una lunghezza della chiave compresa tra 256 bit e 1024 bit. Essendo la chiave derivata utile per la successiva cifratura dei dati con AES CTR si è scelto di utilizzare una chiave da 256 bit, che è la massima supportata per AES CTR.
La firma dei dati è identificata dall'algoritmo RSA SSA con SHA1 e una lunghezza della chiave a 512 bit.
Per la scrittura del codice necessario allo sviluppo del framework è stato utilizzato come linguaggio il C. L'utilizzo di C nella scelta del linguaggio è imperativa, questo perché le chiamate alle funzioni nel TEE hanno bisogno di lavorare in comunione con le funzioni del kernel, che nei sistemi unix-based è scritto in C/C++. La scelta di tale linguaggio non limita le possibilità di

utilizzo del framework nei sistemi Android, Google ha definito una componente del suo sistema operativo chiamato Android Native Development Kit (NDK) che sfruttando il supporto di Java al codice nativo attraverso JNI, permette di utilizzare le funzioni scritte in linguaggio C o C++ direttamente nel codice di una applicazione Android.

4.4 TEA library: definizione e specifiche

L'inizializzazione di una smart card all'interno del TEE è possibile grazie alla comunicazione con un server esterno. Il server avrà il compito di identificare la TA presente nel dispositivo e il proprietario della carta. Rimanendo fedeli alle specifiche del protocollo definito in [1] il TEE e il server completano l'identificazione seguendo quattro fasi. Per gestire i vari step della comunicazione sono state identificate tre funzioni:

- tea_init_protocol_command
- tea_second_phase_command
- tea_complete_process_command

Nei sottocapitoli verranno mostrate più in dettaglio le funzioni delle procedure appena definite, verranno inoltre mostrate le porzioni di codice più significative.
Passando ora alla comunicazione con una smart card è stata prevista una sola funzione:

- tea_process_apdu_command.

La funzione è pensata per ricevere in ingresso le APDU in arrivo dall'hardware NFC. Il flusso di dati in ingresso e in uscita è conforme alle specifiche ISO/IEC 7816, una chiamata a questa funzione è l'equivalente di una comunicazione con una carta fisica.

4.4.1 Protocollo di inizializzazione

In questa sezione verrà mostrata la struttura interna pensata per TEA Library e saranno mostrate in dettaglio le funzioni necessarie per il protocollo di inizializzazione. La funzione `tea_init_protocol_command` rappresenta la fase (1) del protocollo.
Vediamo il prototipo della funzione:

```
1 uint8_t *tea_init_protocol_command(size_t *
    responseApduSize );
```

Listing 4.1: Prototipo tea_init_protocol_command

In questa fase iniziale il protocollo definisce che la TA presente nel TEE inizi una comunicazione con il server di riferimento. La funzione riceve in ingresso un solo valore, un puntatore a una variabile che definisce la lunghezza della risposta. In caso di errore la funzione ritornerà un puntatore nullo e in *responseApduSize* un valore pari a 0. All'interno della funzione verranno inizializzate le variabili che identificano il contesto di esecuzione e la sessione. Questi parametri sono necessari a identificare la comunicazione attualmente in uso e permetteranno di avere una sola comunicazione attiva.

```
1
2 res = TEEC_InitializeContext(ctxName, &ctx);
3 if (res != TEEC_SUCCESS)
4   errx(1, "TEEC_InitializeContext, code 0x%x", res);
5 res = TEEC_OpenSession(&ctx, &sess, &uuid,
    TEEC_LOGIN_PUBLIC, NULL, NULL, &err_origin);
6 if (res != TEEC_SUCCESS)
7   errx(1, "TEEC_Opensession, code 0x%x origin 0x%x", res,
      err_origin);
8 res = TEEC_RegisterSharedMemory(&ctx, &outputTHCE);
9 if (res != TEEC_SUCCESS)
10   errx(1, "TEEC_RegisterSharedMemory, code 0x%x", res);
11
12 memset(&op, 0, sizeof(op));
```

```
13 op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_OUTPUT,
       TEEC_NONE, TEEC_NONE, TEEC_NONE);
14 op.params[0].memref.parent = &outputTEA;
15 op.params[0].memref.offset = 0;
16 op.params[0].memref.size = 256;
17
18 res = TEEC_InvokeCommand(&sess, TA_INIT_PROTOCOL_COMMAND,
       &op, &err_origin);
```

Listing 4.2: Sezione di codice tea_init_protocol_command

Nella sezione di codice appena mostrata si possono notare le fasi di generazione del contesto, l'apertura di una sessione e la registrazione delle memoria condivisa. La memoria condivisa è l'unico modo permesso da TEE per scambiare dati con una TA, si è definita quindi una zona di memoria chiamata *outputTEA* che conterrà il pacchetto di ritorno per il server. Se la funzione richiamata nella TA ritorna con successo, all'interno di *op.params[0].memref.parent* sarà presente il pacchetto generato dalla TA e in *responseApduSize* la dimensione della risposta.
Vediamo ora il prototipo di `tea_second_phase_command` e come viene gestita in TEA.

```
1 uint8_t *tea_second_phase_command(size_t *
     responseApduSize, uint8_t *serverData, size_t
     serverData_len);
```

Listing 4.3: Prototipo tea_second_phase_command

La funzione riceve in ingresso tre parametri che rappresentano rispettivamente, la dimensione del pacchetto di risposta, la stringa ricevuta in ingresso dal server e la lunghezza relativa alla stringa. All'interno della funzione non è più necessario aprire una sessione perchè deve essere utilizzata la sessione inizializzata nella chiamata precedente. In caso contrario viene generato un errore. Il controllo per verificare se la sessione è attiva e unica verrà poi eseguito nella TA. Se una sessione risulta nulla è il sintomo che non è stata eseguita correttamente la funzione di inizializzazione o che l'ordine di esecuzione delle chiamate non risulta corretto. Nel caso in cui la sessione dovesse

risultare diversa dalla sessione attualmente in uso viene bloccata l'esecuzione. Per un qualunque tipo di errore è stato previsto che la funzione elimini la sessione e il contesto correnti, sarà quindi necessario eseguire il protocollo dall'inizio. Vediamo ora il codice relativo alla chiamata della TA nella zona sicura e al passaggio dei parametri necessari.

```
1 op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_INPUT,
     TEEC_MEMREF_TEMP_OUTPUT, TEEC_NONE, TEEC_NONE);
2 op.params[0].memref.parent = &inputTEA;
3 op.params[0].memref.offset = 0;
4 op.params[0].memref.size = serverData_len;
5 op.params[1].memref.parent = &outputTEA;
6 op.params[1].memref.offset = 0;
7 op.params[1].memref.size = 1024;
8
9 res = TEEC_InvokeCommand(&sess, TA_SECOND_PHASE_CMD, &op,
      &err_origin);
```

Listing 4.4: Sezione di codice tea_second_phase_command

Come si può notare, per la funzione, sono stati definiti due vettori, uno in ingresso e uno in uscita. Il vettore in ingresso *inputTEA* è stato precedentemente inizializzato nella funzione con il dato *serverData* ricevuto in ingresso. Questa fase è la più delicata di tutto il processo di inizializzazione, poichè nella TA sarà eseguita la derivazione della chiave simmetrica. Se la derivazione non dovesse andare a buon fine non è possibile cifrare e decifrare i pacchetti delle chiamate successive. Nel caso in cui la chiamata alla TA sia eseguita con successo, `tea_second_phase_command` può ritornare la stringa da inviare al server.
Successivamente al ritorno della chiamata precedente, che comprende le fasi (2) e (3) del protocollo, si è giunti alla parte conclusiva del protocollo di inizializzazione (4).
Vediamo il prototipo della funzione:

```
1 bool tea_complete_process_command(uint8_t *serverData,
     size_t serverData_len);
```

Listing 4.5: Prototipo tea_complete_process_command

I parametri della funzione comprendono in questo caso, solo i dati in ingresso ricevuti dal server.

```
1  res = TEEC_InvokeCommand(&sess, TA_COMPLETE_PROCESS_CMD,
      &op, &err_origin);
2  if (res != TEEC_SUCCESS){
3          errx(1, "TEEC_InvokeCommand␣failed␣with␣code␣0x%x
              ␣origin␣0x%x", res, err_origin);
4          goto error;
5  }
6
7  return false;
8  error:
9  TEEC_CloseSession(&sess);
10 TEEC_FinalizeContext(&ctx);
11 return true;
```

Listing 4.6: Sezione di codice tea_complete_process_command

Nel codice appena mostrato è presente la chiamata alla funzione nella TA che definisce l'ultimo passaggio del protocollo. La funzione riceve in ingresso il contenuto della variabile *serverData* contenente le informazioni in arrivo dal server. Passando ora alle informazioni ritornate da complete process all'interno del TEE, sarà presente il pacchetto con i dati necessari per la definizione della smart card, è quindi importante che la verifica della firma sia corretta. Si può inoltre osservare che nella parte finale del codice sono presenti le funzioni per la chiusura di una sessione e del contesto. Questa scelta identifica la chiusura della comunicazione con il TEE, nel caso si voglia utilizzare nuovamente l'accesso alla zona sicura sarà necessario creare una nuova sessione.

4.4.2 Comunicazione con una smart card

Per comunicare con una smart card nella zona sicura si è pensato di identificare una sola funzione chiamata `tea_precess_apdu_command`. La funzione ha il solo compito di controllare la sessione e richiamare la funzione nel mondo sicuro inviando i dati in arrivo da NFC. Per il controllo della sessione viene solo controllato se esiste già una sessione attiva, altrimenti viene creata. Dopo la verifica della sessione è possibile chiamare la funzione `TA_PROCESS_APDU_CMD` presente nel TEE.

```
1 op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_INPUT,
     TEEC_MEMREF_TEMP_OUTPUT, TEEC_NONE, TEEC_NONE);
2         op.params[0].memref.parent = &inputTEA;
3         op.params[0].memref.offset = 0;
4         op.params[0].memref.size = MAX_APDU_SIZE;
5         op.params[1].memref.parent = &outputTEA;
6         op.params[1].memref.offset = 0;
7         op.params[1].memref.size = MAX_APDU_SIZE;
8 
9         res = TEEC_InvokeCommand(&sess,
             TA_PROCESS_APDU_CMD, &op, &err_origin);
```

Listing 4.7: Sezione di codice tea_complete_process_command

4.4.3 Considerazioni sulla sicurezza

Definita la componente nel REE possiamo iniziare a fare delle considerazioni sulla sicurezza. Rispetto al framework THCE le più grandi differenze sono dovute a una gestione esterna della rete e della comunicazione con NFC. La scelta di spostare la gestione della rete non ha portato a una perdita di sicurezza nel protocollo di inizializzazione. Come già accennato infatti il protocollo previsto in THCE è stato controllato con un tool per la verifica formale chiamato *Schyther* [19]. I risultati del tool hanno permesso di verificare come il protocollo permetta di effettuare uno scambio corretto dei certificati anche in presenza di un attaccante, procedura chiave per un pro-

tocollo a chiave pubblica. Il tool prende in considerazione un attaccante che ha pieno accesso ai pacchetti di rete ed è in grado di cifrare e decifrare i pacchetti se a conoscenza della chiave. L'esito positivo del test garantisce che i due enti in gioco nella comunicazione possano comunicare con un canale sicuro e mutuamente verificato. Possiamo quindi affermare che il caso in questione risulta compatibile con i risultati del tool, la comunicazione sicura è garantita quindi anche con questa variante nella gestione della rete.

Prendendo ora in considerazione la comunicazione via NFC dobbiamo prima considerare il contesto in cui vengono eseguiti gli scambi di informazione. Il massimo della sicurezza garantita attualmente è quella relativa alle smart card contactless fisiche e non emulate. Lo scambio di informazioni avviene in un canale condiviso e pubblico, è quindi sempre possibile leggere le informazioni che vengono trasferite tra la carta e un reader.

In uno scambio di dati via NFC risulta difficile un attacco man-in-the-middle per via della vicinanza necessaria per poter comunicare. In generale quindi le carte contactless cercano di utilizzare e definire protocolli che garantiscano principalmente robustezza ad attacchi di reply. Nell'emulazione con TEA è però possibile avere una lettura e una modifica dei dati in transito da TEE e reader. TEA è stato pensato per evitare l'accesso indesiderato alle informazioni contenute in una smart card emulata, cosa che al momento non è garantita a livello software da altri protocolli. La robustezza del protocollo per la comunicazione via NFC non viene preso in considerazione, sarà compito di un possibile utilizzatore definire un canale sicuro per lo scambio di dati sensibili con il reader.

4.5 TEA Engine: definizione e specifiche

Nella sezione precedente sono state mostrate le componenti presenti nel Rich Execution Enviroment. Vedremo ora, come le chiamate in arrivo dal mondo non sicuro sono gestite nel TEE.
Quando viene fatta la richiesta di creazione di una sessione viene richiamata da TEE la funzione `TA_OpenSessionEntryPoint`.

```
1 TEE_Result TA_OpenSessionEntryPoint(uint32_t param_types,
     TEE_Param  params[4], void **sess_ctx);
```

Listing 4.8: Prototipo TA_OpenSessionEntryPoint

Il processo di inizializzazione della sessione viene definito in questa funzione. La sessione è definita come una semplice variabile, che viene generata casualmente ad ogni nuova apertura a cui però ha accesso solo il TEE. Quando l'inizializzazione di una sessione è avvenuta correttamente dal mondo non sicuro è possibile eseguire la chiamata di un comando.
La chiamata che invoca un comando è rediretta alla funzione `TA_InvokeCommandEntryPoint`.

```
1 TEE_Result TA_InvokeCommandEntryPoint(void *sess_ctx,
    uint32_t cmd_id, uint32_t param_types, TEE_Param
    params[4]);
```

Listing 4.9: Prototipo TA_InvokeCommandEntryPoint

Dal prototipo è possibile notare come la funzione, riceve in ingresso la sessione e il *cmd_id*. Con questi due parametri è in grado di verificare se è presente più di una sessione attiva e a quale comando si vuole fare riferimento. Se la sessione è valida è possibile quindi richiamare la funzione relativa al comando specificato.

4.5.1 Protocollo di inizializzazione

Nel capitolo precedente è stata mostrata l'esecuzione di un comando dall'esterno del TEE, vediamo ora come sono definite internamente le funzioni relative al protocollo di inizializzazione. La prima delle funzioni è *ta_init_protocol*, vediamo ora la sezione di codice relativa agli elementi più importanti.

```
1
2   ret = TEE_AllocateTransientObject(TEE_TYPE_DH_KEYPAIR,
       256, dhKey);
3   if(ret != TEE_SUCCESS)
4     goto free_to;
5
```

```
6   TEE_MemMove(p, keygen_dh256_p, len);
7   TEE_MemMove(g, keygen_dh256_g, 1);
8
9   TEE_InitRefAttribute(&attrs[0], TEE_ATTR_DH_PRIME, &p,
       len);
10  TEE_InitRefAttribute(&attrs[1], TEE_ATTR_DH_BASE, &g,
       1);
11  TEE_InitValueAttribute(&attrs[2], TEE_ATTR_DH_X_BITS,
       xBits, 0);
12
13  ret = TEE_GenerateKey(*dhKey, 256, attrs, sizeof(attrs)
       /sizeof(TEE_Attribute));
14  if(ret != TEE_SUCCESS)
15    goto free;
16
17  ret = TEE_GetObjectBufferAttribute(*dhKey ,TEE_ATTR_DH
       _PUBLIC_VALUE ,PuK , &PuKSize );
18  if(ret != TEE_SUCCESS)
19    goto free;
20
21  params[0].memref.size = PuKSize;
22  memcpy(params[0].memref.buffer, PuK, PuKSize);
```

Listing 4.10: Sezione di codice ta_init_protocol

Nella sezione di codice 4.10 è mostrata la generazione delle chiavi per il protocollo DIffie-Hellman. Per la derivazione della chiave si è utilizzato *TEE_TYPE_DH_KEYPAIR* come gestore per la chiave, dentro cui sono stati caricati i parametri *TEE_ATTR_DH_PRIME* e *TEE_ATTR_DH_BASE*. La scelta per la lunghezza della chiave è ricaduta su 256 bit che come accennato è la massima dimensione supportata da AES, utilizzato successivamente per le operazioni di cifratura. Vediamo ora `ta_second_phase_command`, la funzione che implementa i passaggi (2) e (3) del protocollo.

```
1 TEE_Result ta_second_phase_command(TEE_ObjectHandle *
    dhKey, TEE_ObjectHandle *rsa_keypair, uint32_t
```

```
    param_types, TEE_Param params[4]){
```

Come parametri della funzione sono passati due puntatori a *TEE_ ObjectHandle*, uno gestisce la chiave di tipo Diffie Hellman e il secondo la chiave di tipo RSA per la firma dei pacchetti. Vediamo ora come viene derivata la chiave simmetrica necessaria per lo scambio confidenziale dei pacchetti.

```
1   ret = TEE_SetOperationKey(dhHandle, *dhKey);
2   if(ret != TEE_SUCCESS)
3     goto free;
4
5   TEE_InitRefAttribute(&attr, TEE_ATTR_DH_PUBLIC_VALUE,
       params[0].memref.buffer, params[0].memref.size);
6   TEE_DeriveKey( dhHandle, &attr, 1, derivedKey );
7
8   ret = TEE_RestrictObjectUsage1( derivedKey,
       TEE_USAGE_EXTRACTABLE );
9   if(ret != TEE_SUCCESS)
10        goto free;
11  ret = TEE_GetObjectBufferAttribute(derivedKey ,
       TEE_ATTR_SECRET_VALUE ,sharedKey , &sharedKeySize );
```

Listing 4.11: Sezione di codice ta_second_phase_command

Per la derivazione di una chiave simmetrica sono necessari due parametri, *TEE_ATTR_DH_PRIME* e *TEE_ATTR_DH_BASE*, che saranno passati grazie al gestore della chiave usato inizialmente. Il secondo parametro importante è la chiave pubblica del server che viene impostata nelle funzioni con il buffer *params[0].memref.buffer*. Successivamente viene poi generata la chiave grazie alla funzione *TEE_DeriveKey*. La chiave generata rimane all'interno dell'handle, verrà poi estratta da *derivedKey*. Grazie alla chiave ottenuta è ora possibile decifrare il pacchetto ricevuto dal server, all'interno del quale è presente il pacchetto firmato. Decifrato il pacchetto, viene poi recuperato il certificato del server con cui è possibile verificare i dati contenuti nel pacchetto.
La seconda parte della funzione è dedicata alla fase (3) del protocollo di ini-

zializzazione. In questa fase viene generato il pacchetto di risposta da inviare al server. Il pacchetto deve essere prima firmato e successivamente cifrato.

```
1
2   TEE_MemMove(msg, auhdsadas,auhdsadas_len );
3
4   sha1_digest(msg, msg_len, dig, &dig_len);
5
6   ret = TEE_AllocateTransientObject(TEE_TYPE_RSA_KEYPAIR,
        rsa_key_size, rsa_keypair);
7   if (ret != TEE_SUCCESS) {
8     printf("Fail:␣transient␣alloc\n");
9     goto err;
10  }
11
12  ret = TEE_GenerateKey(*rsa_keypair, rsa_key_size, NULL,
        0);
13  if (ret != TEE_SUCCESS)
14    goto err;
15
16  if (!warp_RSA_sig(*rsa_keypair,
      TEE_ALG_RSASSA_PKCS1_V1_5_SHA1, NULL, 0, dig,
      dig_len, sig, &sig_len))
17    goto err;
```

Listing 4.12: Sezione di codice ta_second_phase_command

Il codice mostrato in 4.12 permette di firmare il pacchetto chiamato *auhdsadas* che contiene le informazioni da inviare al server. La prima operazione per firmare con RSA è creare l'hash del pacchetto con l'algoritmo SHA1, che viene eseguito grazie alla funzione sha1_digest. Il risultato sarà dato in pasto a warp_RSA_sig che completerà l'esecuzione dell'algoritmo per la firma dei dati. Se tutto viene eseguito correttamente in `sig` sarà presente la firma del pacchetto originale.

Come ultima operazione per la funzione deve essere eseguita la cifratura del pacchetto.

```
1
2 if (!warp_sym_enc(key, IV, IVlen, alg, plain, plain_len,
     cipher, &write_to_cipher))
3   goto err;
```

Listing 4.13: Sezione di codice ta_second_phase_command

Per la cifratura con gli algoritmi simmetrici è stata pensata una funzione chiamata `warp_sym_enc` a cui vengono passati il tipo di algoritmo e il pacchetto da cifrare.
Per verificare la firma di un pacchetto, gli algoritmi per la firma digitale ricevono in ingresso il pacchetto originale e la firma. Ricevuti i pacchetti ricreano la firma, e se la firma originale e la firma appena ottenuta risultano uguali, allora il pacchetto non è stato alterato e risulta correttamente verificato.
Per inviare al server tutte le informazioni necessarie, *plain* prima di essere cifrato, viene accodato alla firma del pacchetto appena originata.

Vediamo ora la terza e ultima delle chiamate dedicate alla inizializzazione, dedicata alla finalizzazione della smart card. La funzione utilizza le stesse chiamate mostrate in precedenza per decrittare l'ultimo pacchetto in arrivo dal server e la chiave utilizzata sarà la stessa derivata in precedenza.
Nelle fasi precedenti del protocollo,il server ha inviato all'interno di un pacchetto il proprio certificato. Siamo quindi in grado di verificare la firma del pacchetto contenuto all'interno. In caso di conferma di autenticità viene successivamente elaborato il contenuto. Il server, per inizializzare una carta nel TEE, potrà inviare all'interno del *payload* la struttura interna della carta in questione. La struttura dovrà specificare i DF e gli EF che dovranno essere immagazzinati nella carta emulata e i dati saranno ricevuti con una struttura chiamata TLV, ovvero Tag - Length - Value. Il formato TLV è anch'esso presente nelle specifiche ISO/IEC 7816, viene generalmente utilizzato per lo scambio di informazioni all'interno delle apdu. Un pacchetto in formato TLV ha una struttura composta da un TAG di 2 Byte, una length di 2 byte e un corpo composto da *length* byte che è il vero contenitore di dati. Questo tipo di struttura risulta più che adeguata per definire il file system di una smart card, poichè all'interno del corpo possono essere integrati altri pacchetti TLV,

creando così una struttura ad albero. Nella figura 4.2 è presente un esempio di file system di una smart card all'interno di un pacchetto TLV.

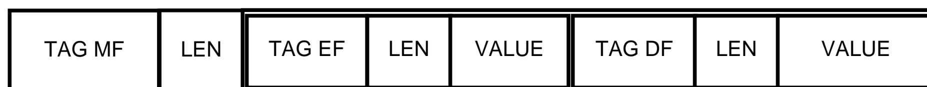


Figura 4.2: Esempio: Tag-Length-Value

L'ultima componente da definire pe il TEA Engine è la funzione `ta_process_apdu_command` che gestisce la apdu in arrivo da NFC. Seguendo la struttura di ISO/IEC 7816-4, per inviare dei comandi a una smart card è possibile utilizzare delle apdu particolari, la cui struttura è definita come segue:

- CLA : Class instruction, identifica la classe, ha dimensione di 1 byte.
- INS : Instruction, serve a identificare il tipo di istruzione selezionata, anche INS ha dimensione 1 byte.
- P1 - P2 : Sono due parametri da un byte ciascuno, il loro contenuto è dipendente dai due parametri precedenti e servono a fornire informazioni aggiuntive alla istruzione selezionata.
- Lc field : Contiene la lunghezza del *field* in byte.
- Data field : Il campo dati, qui viene inserito il contenuto da passare alle istruzioni.
- Le field : Lunghezza della risposta in byte.

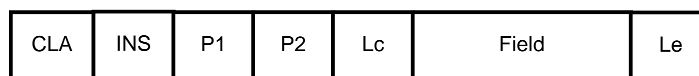


Figura 4.3: Command APDU

Per ridurre al minimo la possibilità di interagire in modo scorretto con una smart card nel TEE è stata definita solo la funzione *ta_ complete_process_ command.* Essa rappresenta la componente definita in precedenza come CPS e ha la

funzione di leggere il contenuto di una APDU per ricavare il comando corrispondente. Non è possibile accedere alla funzionalità della carta se non attraverso questa funzione.
Per ogni comando supportato è presente una funzione dedicata, vedremo in seguito le funzioni definite dallo standard per la gestione di una smart card. Se il comando non è supportato viene prodotta una APDU in risposta, contenente un codice di errore. La funzione *ta_complete_process_command* è definita come un parser, ricevuto in ingresso una APDU separa il byte CLA e il byte INS con cui identifica un comando, chiamerà così la funzione dedicata.

```
adpu_len = params[0].memref.size;
adpu = TEE_Malloc(adpu_len, 0);
TEE_MemMove(adpu,params[0].memref.buffer, adpu_len);

if(adpu[0] != 0x00 || adpu[0] != 0x80)
  return TEE_ERROR_BAD_PARAMETERS;

switch(adpu[1]){
case 0xA4: //select command
  if(adpu[2] == 0x04 && adpu[3] == 0x00)
    ret = select_command(params);
break;

case 0xA8://generate application cryptogram
if(adpu[2] == 0x00 && adpu[3] == 0x00)
  ret = generate_ac_command(params);
break;
```

Listing 4.14: Sezione di codice ta_complete_process_command

In 4.14 è possibile osservare come vengano presi in considerazione i primi 2 byte della apdu in ingresso, che permettono di selezionare il comando corrispondente. Come accennato in precedenza il contenuto della carta verrà sempre gestito dal TEE, non vi è mai una possibilità di accedere ai dati contenuti internamente se non con le funzioni dedicate. Un errore nella composizione della chiamata all'interno della APDU arresta il funzionamento della stessa

e la funzione genera una APDU di errore. I comandi base supportati sono i seguenti [5]:

CLA (0x00)	INS	P1	P2	DATA
ERASE BINARY	0E	hi-offset	low-offset	vuoto
VERIFY	20	00	id Sec Object	DATA
MANAGE CHANNEL	70	00/80	00/01/02/03	vuoto
EXTERNAL AUTENTICATE	82	algoritmo	id Sec Object	DATA
GET CHALLENGE	84	00	00	vuoto
INTERNAL AUTHENTICATE	88	algoritmo	Id sec object	DATA
SELECT FILE	A4	mode	selection option	id file
READ BINARY	B0	hi-offset	low-offset	vuoto
READ RECORD	B2	record number	reference control	vuoto
GET RESPONSE	C0	00	00	vuoto
ENVELOPE	C2	00	00	DATA
GET DATA	CA	custom	custom	vuoto
WRITE BINARY	D0	hi-offset	low-offset	DATA
WRITE RECORD	D2	record number	reference control	DATA
UPDATE BINARY	D6	hi-offset	low-offset	DATA
PUT DATA	DA	custom	custom	DATA
UPDATE RECORD	DC	record number	reference control	DATA
APPEND RECORD	E2	00	reference control	DATA

Tabella 4.3: Comandi APDU

Le funzioni appena elencate sono le funzionalità base che deve supportare una smart card compatibile con ISO 7816. Nello standard sono previsti alcuni comandi che permettono un modifica dei contenuti. Queste funzioni servono in genere per l'inizializzazione di una carta aggiornando il contenuto dei record. L'inizializzazione della carta avviene in TEA attraverso la verifica online, non è quindi necessario utilizzare tali funzionalità. Si è previsto quindi di non supportare queste funzioni, come ad esempio WRITE BINARY e UPDATE BINARY.

4.6 Principali differenze tra TEA e THCE

Con l'introduzione di TEA sono state apportate delle modifiche all'architettura pensata in THCE.
La prima caratteristica che andremo ad osservare è la mancanza della logica per il monitoring delle applicazioni, che nel protocollo di riferimento è definito come *THCE Monitor*.
Le capacità di monitoring sono ancora presenti nel framework ma come mostrato nelle sezioni precedenti sono distribuite all'interno di TEA Engine. Globalplatform per interfacciare TEE e REE definisce un canale, identificato da un contesto e una sessione. Iniziata la comunicazione è possibile iniziare a inviare comandi al TEE. Per TEA si è deciso di utilizzare la sessione attiva come elemento distintivo per filtrare le chiamate in arrivo alla zona sicura senza introdurre un modulo dedicato al monitoring.
Una seconda variazione rispetto al framework originale è la gestione delle comunicazioni esterne del TEE. Inizialmente, la struttura, era pensata per gestire internamente alla zona sicura le funzionalità di rete e la comunicazione con l'hardware NFC. Globalplatform, nella definizione del TEE ha previsto delle chiamate a funzioni capaci di gestire i socket attraverso i protocolli TCP e UDP. La scelta che è stata intrapresa è però di lasciare la gestione della rete alle applicazioni esterne al TEE, nel nostro caso ad Android. La struttura così implementata può permettere una completa personalizzazione sulla scelta del trasporto dei pacchetti sulla rete, che nel caso di una definizione interna non sarebbe stata possibile.
La gestione del protocollo di inizializzazione rimane comunque implementata all'interno di TEE, non è quindi possibile alterare le specifiche del protocollo. Come mostrato in precedenza il sistema così definito mantiene le caratteristiche di sicurezza pensate idealmente nel framework THCE, questo perché nella definizione del protocollo sono state fatte delle considerazioni e dei test di sicurezza. Se una applicazione Android malevola dovesse tentare un attacco, come per esempio man-in-the-middle, il protocollo è in grado di riconoscerlo e di bloccarne l'esecuzione.

Una ulteriore differenza va evidenziata nella gestione dell'hardware NFC. I pacchetti in arrivo dall'hardware in TEA sono ricevuti dal sistema Android. Questa scelta è principalmente dovuta a una mancanza del driver NFC nel kernel dedicato a TEE. Non è stato infatti possibile nei nostri test raggiungere NFC dall'interno della zona sicura, si è scelto quindi di delegare Android per il trasferimento dei pacchetti NFC.
La comunicazione con una Trusted Application secondo le direttive di Globalplatform, avviene solo mediante delle API già definite per TEE. Per rimanere fedeli alle specifiche di Globalplatform si è quindi preferito mantenere la struttura attuale per la gestione delle API.
La comunicazione con un TA è delegata a una nuova libreria, che sarà richiamabile nelle applicazioni Android, che chiameremo TEA Library.

Capitolo 5

Risultati sperimentali

Questo capitolo presta il framework TEA per una analisi sperimentale, con l'obiettivo di verificare l'affidabilità e l'efficacia in contesti reali.
A ulteriore supporto dell'analisi sperimentale, il TEA viene applicato in un caso d'uso reale, il pagamento contactless tramite NFC, identificato nel capitolo successivo. Per testare il sistema sono stati utilizzati diversi dispositivi, ognuno dei quali servirà per verificare le diverse funzionalità implementate. La prima parte sarà dedicata alla definizione del sistema di test e seguiranno le analisi dei risultati.

5.1 Setup

Inserire una Trusted Application in un dispositivo compatibile con ARM Trustzone è possibile solo all'atto di inizializzazione del sistema. La gestione della zona sicura è supportata da un kernel ausiliario che viene eseguito in contemporanea con il kernel di base. ARM ha implementato TrustZone in tutte le nuove linee di CPU a partire dalla versione ARM v7, il suo utilizzo può però essere bloccato via software a livello del bootloader. Non avendo a disposizione un dispositivo con bootloader sbloccato si è deciso di procedere emulando un hardware TEE.
Attualmente una delle soluzioni più complete per emulare un dispositivo ARM è di utilizzare un software sviluppato da linaro chiamato OpenTEE.

OpenTEE è un progetto che implementa un "TEE virtuale" che rispetta le più recenti specifiche di Globalplatform. L'idea principale è di poter testare le Trusted Application in un ambiente virtuale ma perfettamente compatibile con ARM Trustzone. Per l'emulazione di hardware ARM, OpenTEE rilascia un ambiente che si basa sull'emulatore ufficiale di ARM chiamato *Fixed Virtual Platforms* (FVP). L'emulatore permette di sviluppare le proprie applicazioni in un ambiente con base Linux.

La scelta di sviluppare in un ambiente Linux non influenza i risultati dei test, si ricorda che un dispositivo Android ha un Kernel Linux e nel caso di ARM Trustzone le librerie sono implementate in C. TEA in modo analogo è implementata interamente in C, ed è stata pensato per essere integrato in un dispositivo Android utilizzando JNI, che permette a una applicazione di utilizzare codice nativo in linguaggio C e C++.

Per simulare un dispositivo Android si è scelto di usare un LG G2 con Android 6.0.1 Marshmallow, attualmente il mio dispositivo personale. Nel dispositivo è stata sviluppata una applicazione Android che ha i permessi per l'accesso alla rete e all'hardware NFC che sono fondamentali per i test.
Per ovviare all'impossibilità di avere accesso al TEE, si è scelto di implementare un emulatore di TEA. In questo modo l'applicazione di test avrà la possibilità di utilizzare la TEA Library come se si stesse interfacciando con il vero TEA.
Per la fase di inizializzazione, è stato creato un server con le funzionalità del protocollo websocket, che potesse riceve le informazioni da TEA per inizializzare una carta. Il server è gestito come una servlet caricata su un server virtuale.
Per testare le funzionalità di emulazione della carta è stata implementata una seconda applicazione Android all'interno di un tablet Nexus 7, che simula il comportamento di un Reader NFC/PoS. Il reader utilizzerà delle chiamate per testare i comandi di una carta NFC.

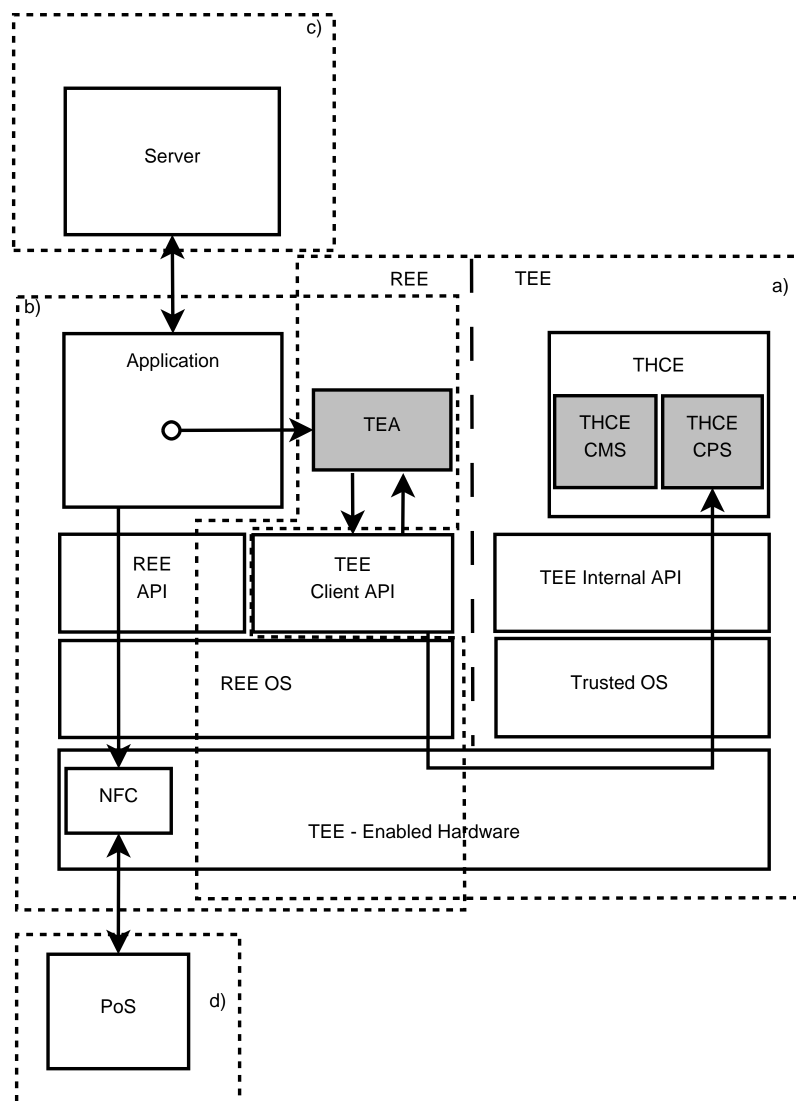


Figura 5.1: Test TEA: struttura di riferimento

In figura 5.1 vengono presentati i componenti per testare il framework TEA. La componente evidenziata in *a* è la parte definita in FVP, l'emulatore ARM che ha permesso la verifica del funzionamento dell'implementazione in TEE. La parte relativa a *b* è rappresentata da LG G2, lo smartphone, che gestisce le componenti di rete e la comunicazione NFC. *C* rappresenta invece il server della banca, implementato in un server virtuale, che permette la gestione delle fasi di inizializzazione del protocollo. L'ultima componente è

d, l'applicazione Android sviluppata su Nexus 7, che emula il comportamento di un reader NFC.

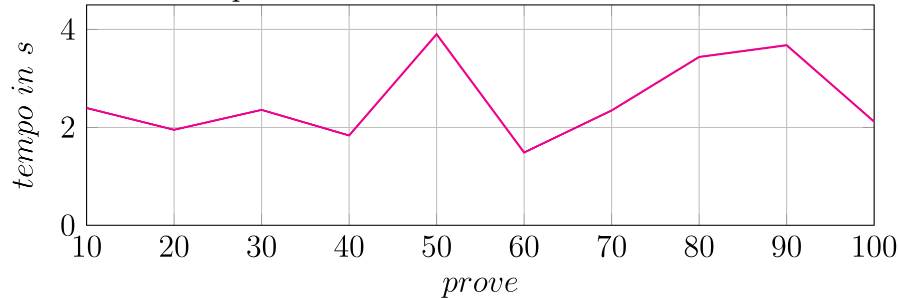
5.2 Protocollo di inizializzazione

Vediamo ora come è stata testata la prima delle fasi necessarie per poter eseguire il framework, ovvero il protocollo di inizializzazione della carta effettuato tra un server e il dispositivo mobile.
Per poter comunicare utilizzando internet sono presenti molti protocolli, quello scelto per il test è stato WebSocket. É stato possibile implementare il supporto a WebSocket per Android e Java grazie a una libreria chiamata Autobahn prodotta da Tavendo [24] che supporta molti linguaggi di programmazione. Questa libreria, compatibile sia con Android che con Java standard, integra le seguenti funzioni:

- OnOpen() : permette di gestire l'evento di apertura del canale instaurato tra il Server e TEA.
- OnBinaryMessage() : in caso di ricezione di un pacchetto durante la sessione, qui viene gestita la logica del protocollo.
- sendBinaryMessage() : funzione con cui sono stati inviati nel socket i messaggi in formato binario.
- disconnect() : per chiudere la connessione attiva alla fine del protocollo.

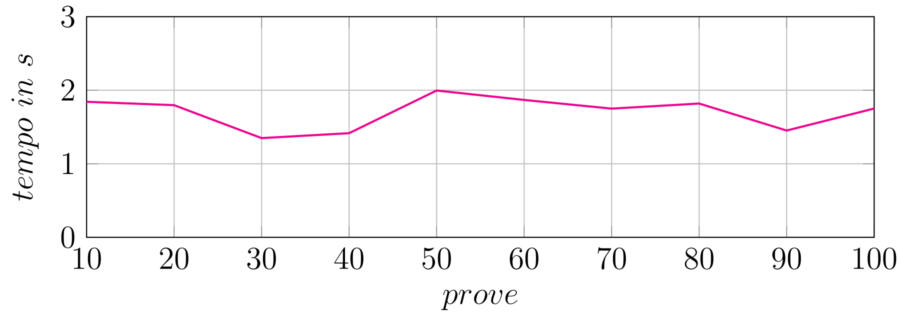
Il server implementa una applicazione Java enterprise, dotata di una servlet che definisce la logica del protocollo lato server.
Le prove sono state divise in due fasi, una prima fase dove il protocollo è stato eseguito tra il dispositivo Android con TEA emulata e il server, e una seconda dove il protocollo è stato testato all'interno dell'emulatore FVP. La divisione dei test ha permesso di provare il framework nei due ambienti di riferimento, la zona sicura e il mondo Android. Per la simulazione sono state eseguite 100 prove consecutive per ognuna delle due fasi appena descritte.

L'esecuzione in Android rispecchia il tempo impiegato dal sistema nelle fasi di trasferimento dei pacchetti attraverso la rete:



Nel grafico appena mostrato sono contenuti i tempi necessari a completare il protocollo di inizializzazione tra il dispositivo e il server. Dalle informazioni del grafico si può vedere che i tempi risultano molto variabili. La variabilità può essere attribuita a molti fattori, per esempio a come Android gestisce le priorità delle applicazioni e il traffico della rete all'istante del test. In questa fase, il framework, non prevede un limite di tempo, bisogna comunque ricordare che in caso di diffusione del protocollo i tempi di inizializzazione di una carta devono essere brevi, nell'ordine dei secondi. Tutte le esecuzioni dei test sono state completate con successo e il tempo massimo ottenuto è stato di 4 secondi.
Passando ora per la seconda fase di test, sono state eseguite nuovamente le chiamate previste dal protocollo di inizializzazione all'interno dell'emulatore ARM.

Per eseguire i test, TEA Library, è stata utilizzata per chiamare le funzionalità all'interno della TA. Sono state create delle funzioni aggiuntive in TEA Library che hanno permesso di emulare il comportamento del server, in modo da ricevere e inviare i pacchetti per completare il protocollo di inizializzazione. Per questa fase è stato considerato il tempo di esecuzione delle funzioni interne al TEE verificando al contempo il corretto funzionamento del protocollo.



Paragonando i risultati si può notare che il TEE ha completato il protocollo con un tempo massimo di 2 secondi. Ricordiamo che con questo test non viene preso in considerazione il tempo necessario all'intera procedura, ma solo il tempo impiegato da TEA implementato nel TEE a completare il protocollo. Come accennato prima, il protocollo non ha un limite di tempo massimo ma una esecuzione con un tempo superiore alle decine di secondi potrebbe portare un utilizzatore a pensare a un blocco dell'applicazione.
Prendendo come riferimento il tempo massimo delle due prove il risultato è di poco inferiore ai 6 secondi.

Capitolo 6

Caso d'uso: pagamento contactless

Nel mondo delle smart card contacless i servizi disponibili sono sempre in espansione. Il caso d'uso con il maggior numero di adozione è nel campo dei pagamenti contactless.
Un pagamento contactless permette a un utente di effettuare una transazione semplicemente avvicinando la propria carta a un PoS (Point of Sale), ovvero a un reader NFC dedicato, che comunicando con una banca identifica il cliente. All'interno di una carta, sono inserite prima del rilascio al cliente le applicazioni e i certificati necessari. Questi, oltre a identificare univocamente una tessera, e di conseguenza un cliente, identificano uno dei possibili protocolli presenti per le smart card permettendo al PoS di iniziare una transazione.
Lo scenario ideato per il test prevede l'utilizzo del framework TEA per eseguire un pagamento contactless. Il test rappresenta uno scenario dove un utente, dopo aver aperto un nuovo conto, decide di voler usufruire del servizio di pagamento attraverso l'applicazione proprietaria della banca. La banca suggerisce di inizializzare la carta sul proprio dispositivo, che verrà poi abilitata ad eseguire pagamenti.
Seguendo lo schema previsto nella fase di testing, utilizziamo quindi un server apache che simula il comportamento del server della banca, un tablet che simula il PoS e per ultimo uno smartphone che contiene la libreria per

l'emulazione della carta.
Per rendere lo scenario il più reale possibile si è pensato di scegliere come riferimento il protocollo adottato da VISA per le sue carte. Per verificare la corretta esecuzione del protocollo sono state pensate due prove separate.
La prima vede il software TEA implementato in FVP, dove grazie alla libreria TEA Library è stato possibile simulare l'arrivo di pacchetti da un PoS NFC.
La seconda prova prevede di richiamare i comandi direttamente da una carta VISA, una PostePay, dove sono stati eseguiti gli stessi comandi della prova precedente. Ottenuti i risultati delle due prove sono state fatte delle considerazioni sui risultati ottenuti per confrontare le due differenti modalità.

6.1 PostePay e protocollo VISA PayWave

Il protocollo preso come riferimento per questa fase di testing è stato PayWave [18], che definisce le procedure adottate dalle carte contactless del circuito VISA.
PayWave è definito seguendo le linee guida rilasciate da EMV, Europass Mastercard Visa, l'ente europeo che definisce le specifiche per le carte dedicate ai pagamenti. Nelle specifiche EMV vengono definiti i materiali utilizzati, i protocolli a livello fisico e le api di una smart card, e sono previste delle specifiche sia per le carte con chip che per le carte contactless.
Come accennato nei capitoli precedenti, le carte contact e contactless utilizzano specifiche a livello software che risultano una espansione dello standard ISO 7816-4. É stato quindi possibile estendere le funzionalità di TEA con le funzioni necessarie. Vediamo ora quali sono i comandi utili per il completamento di PayWave.

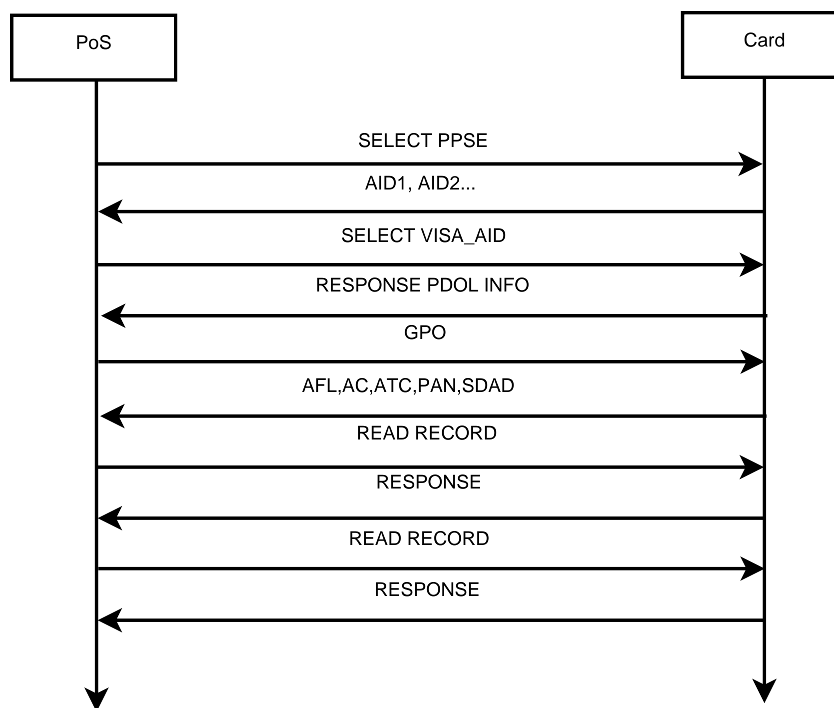


Figura 6.1: Protocollo PayWave

Nel protocollo, ma più in generale nella lettura di una carta vi è sempre una fase iniziale di selezione delle applicazioni presenti. La prima selezione viene eseguita su PPSE ovvero *paypass payment system environment*. PPSE è una sigla che identifica un EF all'interno della carta che contiene le informazioni necessarie all'identificazione delle applicazioni dedicate ai pagamenti del circuito PayPass. Le informazioni di ritorno dalla *select* serviranno per identificare il *DF_ Name* del DF interno alla carta, contenente le informazioni della applicazione PayWave.
Grazie a *VISA_ AID* ricevuto in risposta, è possibile eseguire una seconda *select*, che restituisce i dati necessari per completare la transazione.
Le informazioni ritornate dalla *select* sono inserite all'interno di un pacchetto chiamato Processing DOL (PDOL). Il PDOL è uno dei possibili utilizzi dei pacchetti DOL, ideati per limitare al massimo lo scambio di informazioni tra la carta e un reader. Un pacchetto DOL non segue il formato predefinito TLV ma è composto da una sequenza di tag comando e lunghezza in Byte

che serviranno nelle chiamate successive.
Esistono vari tipi di DOL che dipendono dalla funzione specifica e possono essere PDOL, CDOL1/2 e TDOL. Per il protocollo PayWave è necessario il solo pacchetto PDOL.

TAG	Bytes	Valore
9F66	04	terminal transaction authorized
9F02	06	amount authorized
9F03	06	amount other
9F1A	02	terminal country code
95	05	terminal verification results
5F2A	02	transaction country code
9A	03	transaction date
9C	01	transaction type
9F37	04	unpredictable number

Tabella 6.1: PayWave processing DOL

Nella tabella 6.1 sono presenti i parametri contenuti nel PDOL ritornato dalla carta, e si può notare come il contenuto dei campi permetta di verificare la corretta esecuzione della transazione.
É anche bene notare come nelle richieste della carta vi sia un valore chiamato *unpredictable number*, utilizzato da VISA per eliminare i possibili attacchi di reply.
Ottenute le informazioni, il PoS, ha ora i dati necessari per procedere con la chiamata *Generate Processing Option* (GPO). La funzione GPO sostituisce il vecchio comando per la generazione di un Application Cryptogram (AC). Un AC è un pacchetto che viene generato dalla carta, ed è un comando interpretabile solo dalla banca. Il protocollo PayWave permette le modalità di pagamento solo online, non è necessario l'utilizzo di un PIN ma richiede l'autorizzazione del server.
PayWave definisce i seguenti passaggi per la generazione di GPO. Il primo passaggio consiste nella creazione di una variabile di sessione chiamata K_s. Per la creazione è necessario eseguire un algoritmo per la cifratura della variabile application transaction counter (ATC) utilizzando come chiave master

key (MK). MK è una chiave che viene inserita all'interno della carta all'atto dell'inizializzazione ed è nota solo alla banca e alla carta. Per creare una chiave di sessione efficace bisogna utilizzare un valore da cifrare differente a ogni chiamata. All'interno del pacchetto viene usato ATC, aggiornato ad ogni esecuzione della funzione GPO. Viene quindi garantito un valore sempre diverso per la chiave di sessione.

$$K_{\mathrm{s}} = Enc_{\mathrm{MK}}(ATC) \tag{6.1}$$

Vediamo ora la generazione vera e propria di AC. Per derivare AC viene eseguito l'algoritmo MAC utilizzando come chiave il valore $\mathrm{K_s}$ appena generato, e come pacchetto da cifrare i dati ricevuti dal PoS relativi alla transazione.

$$AC = MAC_{\mathrm{Ks}}(amount, ATC, UN, ...) \tag{6.2}$$

Questo approccio garantisce che i dati che la carta convalida siano effettivamente quelli inviati dal PoS.

$$SDAD = Sign_{\mathrm{privC}}(AC, UN, amount, currency, ATC, nC, ...) \tag{6.3}$$

Nell'ultima parte del protocollo viene generato il pacchetto SDAD che una volta ricevuto dal PoS può essere verificato online dal server della banca. Successivamente alla chiamata GPO, vengono lette le informazioni pubbliche della carta con le funzioni read record. PayWave nella documentazione prevede una durata massima della transizione di 500ms, per il tempo della transazione viene considerata la sola esecuzione di GPO.

6.2 Prova di pagamento

Prendiamo ora in considerazione l'esecuzione dei test. Per eseguire i test abbiamo inizializzato una carta all'interno dell'emulatore FVP con i parametri necessari per eseguire il protocollo appena descritto. TEA supporta al momento le funzioni standard previste da 7816-4, per rendere il sistema

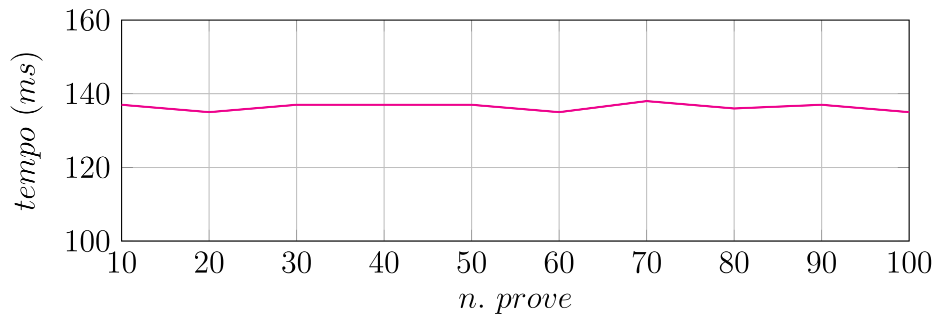
pienamente compatibile con il protocollo PayWave si è aggiunto il supporto a Generate Processing Option che permetterà la generazione del pacchetto *SDAD*. In questa sezione non verranno mostrati i dati relativi al protocollo di inizializzazione, poiché i risultati ottenibili sarebbero analoghi a quelli ottenuti in precedenza.
Il protocollo di inizializzazione è stato eseguito una sola volta, con il solo scopo di inizializzare la carta all'interno di TEA.
Per simulare l'esecuzione della transazione, è stata eseguita varie volte la sequenza di operazioni definita da PayWave, dove per ogni prova, sono stati misurati i tempi necessari al completamento della chiamata GPO. Si può notare infatti, che l'unica chiamata utile per la transazione sia effettivamente la chiamata GPO, le select iniziali e le read record finali servono solo a selezionare una applicazione valida per il pagamento e ottenere le informazioni della carta. Per richiamare le funzionalità di TEA viene utilizzata la libreria TEA Library in FVP. La libreria eseguirà in sequenza le chiamate necessarie al protocollo PayWave. Per la funzione Generating Processing Options si sono utilizzati i seguenti parametri:

TAG	Valore
Terminal transaction authorized	36 E0 40 00
Amount authorized	00 00 00 00 10 00
Amount other	00 00 00 00 00 00
Terminal country code	03 80
Terminal verification results	00 00 00 00 00
Transaction country code	03 80
Transaction date	16 01 13
Transaction type	00
Unpredictable number	8E 23 13 76

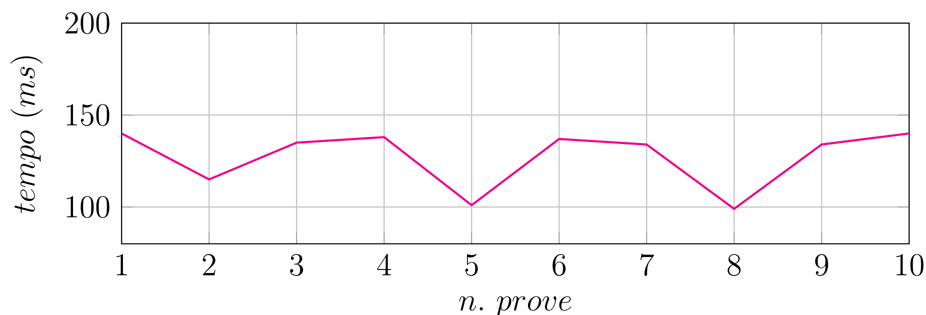
Tabella 6.2: Processing DOL

Il valore terminal transaction authorized identifica la modalità di pagamento che il PoS è in grado di risolvere. Se la carta non può rispettare queste modalità il pagamento viene rifiutato. I parametri successivi identificano: la cifra del pagamento (Amount authorized), il codice di riferimento italiano

(Terminal country code e Transaction country code), la data della transazione (Transaction date) e un numero casuale (Unpredictable number).
Per il primo test sono state eseguite 100 prove consecutive del protocollo utilizzando i parametri appena mostrati, ed è stato preso come riferimento il tempo della sola chiamata GPO.

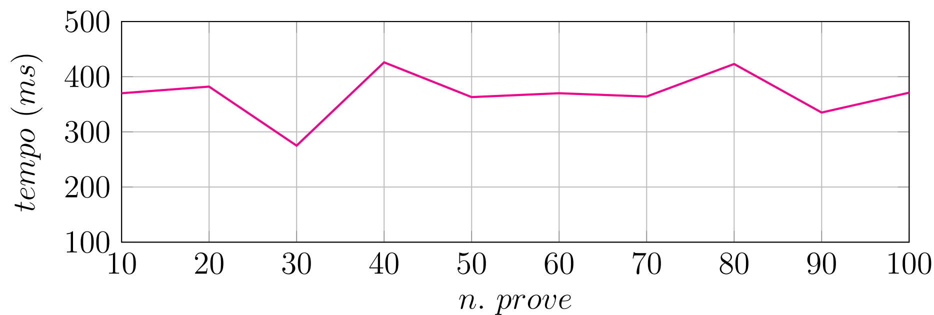


In questa fase del test si è preso in considerazione il solo emulatore FVP che non ha accesso all'hardware NFC, i risultati registrati mostrano solo il tempo necessario al completamento di GPO e non comprendono il tempo di trasferimento necessario ad Android per inviare e ricevere i dati via NFC.
Il comando GPO richiede una sola comunicazione in ingresso e una in uscita, per ricavare il tempo di trasferimento impiegato da Android a trasferire i dati via NFC si è pensato di utilizzare una carta NFC fisica, dove verrà eseguita una funzione di prova, per esempio una select. Il tempo misurato nell'esecuzione della select rappresenta un valore prossimo al solo tempo necessario al trasferimento dei dati. Questa assunzione non si discosta molto dalla realtà, poiché la select è una operazione che richiede un tempo minimo.

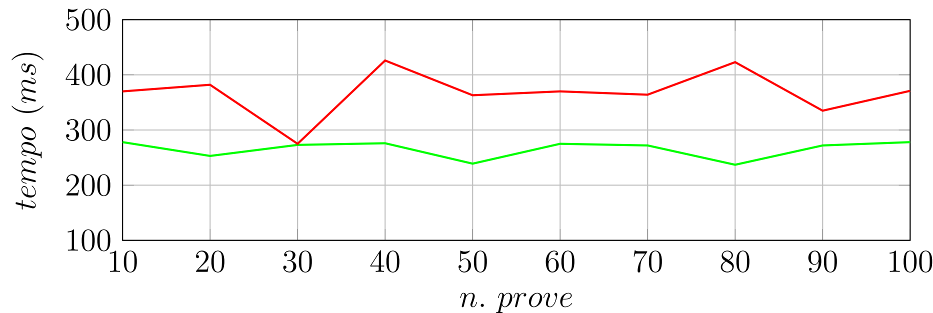


I risultati nel grafico identificano il tempo misurato dalla applicazione An-

droid tra l'esecuzione del comando select e la ricezione di una risposta da parte della carta fisica PostePay.
Per questo test non è stato necessario eseguire un numero elevato di prove poichè il risultato è solo un upperbound che indica il tempo necessario ad Anroid ad eseguire il trasferimento dei dati via NFC.
Prima di effettuare una analisi dei risultati vediamo il tempo impiegato da PostePay con il medesimo protocollo per il pagamento, in questa fase è stato utilizzato il tablet Nexus 7 come un PoS per scambiare i pacchetti con la carta.



I tempi risultano più alti di quelli ottenuti nell'esecuzione del comando diretto a TEA, ma ricordiamo che i risultati per TEA non tengono conto del tempo necessario per il trasferimento dei pacchetti via NFC.
Vediamo ora un grafico che mostra contemporaneamente le prove eseguite con TEA e con PostePay.
Nel grafico i risultati in verde mostrano il tempo impiegato da TEA a rispondere a GPO a cui è stato sommato il tempo massimo ottenuto nell'esecuzione del comando select. Questo dato rappresenta un valore sovrastimato del tempo di esecuzione necessario al framework TEA all'interno di un device per rispondere a una chiamata GPO.
In rosso sono riportati i risultati ottenuti dall'esecuzione del comando su PostePay.



VISA ha previsto per PayWave un tempo massimo per la transazione di 500 ms. Il limite di tempo è stato introdotto per evitare possibili attacchi di reply durante l'esecuzione di una transazione. I risultati ottenuti mettono in risalto le bontà del framework TEA, che ha restituito risultati analoghi a quelli ottenuti da una carta PostePay rispettando il limite dei 500 ms, risultando inoltre più veloce della carta presa come riferimento.

6.3 Analisi dei risultati

In riferimento ai risultati ottenuti in questa fase di testing, possiamo trarre delle considerazioni sul framework TEA.
Nel caso d'uso di un pagamento contactless TEA ha permesso di emulare correttamente il comportamento di una PostePay, risultando più rapido nel completamento della transazione.
Questo dato è importante perché ci identifica che TEA è una tecnologia reale e applicabile in un ambito delicato come il pagamento contactless.
In riferimento alla sicurezza possiamo sovrapporre il comportamento della carta PostePay con quello di TEA. I dati sensibili contenuti nella carta PostePay, ma in generale nelle carte contactless, sono accessibili solo dalle funzioni interne, questo garantisce che i certificati e le informazioni sensibili siano mantenute separate dall'esterno. Gli unici dati in uscita dalla carta sono i dati previsti dal protocollo utilizzato, in questo caso PayWave.
Per come è definito TEA l'unico modo per accedere alle funzioni e i dati di una smart card è grazie ai comandi definiti in TEA Library.
Questo approccio è identico a quello ideato per una smart card o per l'emula-

zione di carte con SE, i sistemi con la più alta sicurezza presenti attualmente sul mercato.
Possiamo quindi concludere che se il protocollo utilizzato per una particolare applicazione non prevede il passaggio in chiaro di dati sensibili via NFC, TEA permetta un isolamento completo dei contenuti in una zona isolata dal sistema operativo Android, emulando al contempo il comportamento di una smart card tradizionale.

Capitolo 7

Stato dell'arte

Il problema delle sicurezza nell'emulazione di smart card è presente ormai da molti anni e il mercato ha proposto nel tempo molte soluzioni. Non tutte le applicazioni presentate si sono rivelate una scelta solida e con una buona presa sul mercato. Il livello massimo di sicurezza garantito attualmente dalle smart card risiede nell'adozione di Secure Element (SE), piccoli chip con funzioni crittografiche e una sicurezza hardware intrinseca dell'architettura. Questo è risultato un ottimo approccio per molti anni. La difficoltà nell'implementazione dovuta alla presenza di un hardware dedicato e proprietario ne ha reso difficile la diffusione su larga scala. Inoltre l'emulazione con SE risulta vulnerabile ad alcuni tipologie di attacchi, i reply attack, come viene descritto in [11]. Le difficoltà avute con SE hanno portato il mercato ad approcciare un nuovo sistema emulando le smart card direttamente a livello software. HCE è la più recente alternativa software per l'emulazione di carte, ma è risultata una soluzione non ottimale, seppur venga ancora utilizzata. In [6] e [7] è stato approfondito il servizio HCE e vengono messe in risalto le vulnerabilità di questa soluzione. Il principale problema in questo sistema è l'implementazione software di HCE, essendo una applicazione integrata nel sistema viene eseguita insieme al resto delle applicazioni del device.
Recentemente sono state ideate soluzioni dove viene evidenziato il vantaggio dell'uso di Trusted Execution Enviroment (TEE), una zona sicura presente nelle CPU dei dispositivi mobili. Una di queste è stata proposta da

Koinstiaen [7] che introduce On-board Credentials (ObC), un framework che garantisce una gestione sicura delle credenziali sfruttando il TEE di un dispositivo. Una seconda revisione di ObC è stata proposta in [8]. Questo nuovo approccio è poi divenuto più concreto quando Nokia ha adottato ObC nei suoi dispositivi. L'approccio scelto da Nokia è una ottima soluzione per il salvataggio di chiavi all'interno di un device compatibile con TEE. La soluzione però concentra lo sviluppo sulla gestione sicura delle chiavi ed è confinata all'uso sui soli dispositivi di Nokia. Recentemente è stato presentato [15] dove viene effettuata ancora volta una analisi sulla sicurezza nell'emulazione di carte NFC focalizzandosi sui vantaggi offerti da TEE e le principali differenze con i SE.

Per aumentare la sicurezza di HCE i servizi come ad esempio Google Wallet hanno iniziato a supportare una gestione cloud del SE. In [16] Underwriters Laboratories ha però evidenziato come questa soluzione non sia compatibile con l'emulazione di carte per i pagamenti. Nell'articolo viene fatta una considerazione sui tempi necessari per la comunicazione con il cloud. La latenza elevata rende non applicabile questa soluzione per l'uso nel protocolli attualmente previsti per i pagamenti come MasterCard PayPass (400 ms) e PayWave (500 ms). Si può notare comunque come il mercato abbia intuito le potenzialità offerte dal TEE e di come questa tecnologia possa portare notevoli benefici nella sicurezza. Recentemente altre soluzioni proprietarie sono state introdotte sul mercato, Apple Pay e Samsung Pay sono i due esempi più importanti. Apple e Samsung Pay introducono un nuovo concetto di emulazione di carte e grazie a una combinazione di SE, cloud e TEE le due soluzioni propongono un alto livello di sicurezza. Le due proposte sono però proprietarie e utilizzano un hardware e un software dedicato, inoltre l'emulazione è dedicata solo ai pagamenti via NFC e non si estende a tutte le smart card.

Tutte le soluzioni presenti attualmente approcciano il problema sfruttando solo marginalmente le potenzialità di TEE. Nessuna delle proposte garantisce attualmente un supporto completo ai pagamenti via NFC all'interno di TEE, approccio simile a quello presente in un SE. In questa tesi si è scelto di intraprendere la strada introdotta dal framework THCE [1] che introduce

una struttura capace di gestire l'emulazione completa di una smart card in un ambiente sicuro e isolato come quello di TEE. Questo approccio è l'unico che permette una sicurezza comparabile a quella garantita da un SE ma utilizzando un hardware presente nella maggior parte dei dispositivi potrebbe avere una grande diffusione.

Capitolo 8

Conclusioni e sviluppi futuri

In questa tesi è stato progettato, sviluppato e testato un framework per NFC Card Emulation per dispositivi Android sulle specifiche del *Trusted Host-based Card Emulation*definite in [1]. L'implementazione, chiamata *Trusted host-based card Emulation for Android* (TEA), ha portato THCE nei dispositivi mobili con sistema operativo Android compatibili con TEE.

Lo sviluppo di TEA è stato possibile con l'utilizzo di OpenTEE, un ambiente simulato prodotto da Linaro, che ha permesso di implementare le varie componenti del sistema presenti nel TEE.

Per testare TEA si è scelto di utilizzare il sistema dei pagamenti contactless adottando come standard di riferimento il protocollo PayWave previsto da VISA. I risultati hanno dimostrato l'efficacia del sistema TEA che ha prodotto risultati paragonabili a quelli ottenuti da una tradizionale carta di credito del circuito VISA (PostePay).

La struttura modulare del framework TEA lascia lo spazio a possibili sviluppi futuri. Utilizzando le funzioni attualmente implementate sarebbe possibile interagire con Trusted Applications aggiuntive, definendo ulteriori utilizzi per il framework.

Ad esempio, attualmente in TEA la gestione dell'NFC viene implementata nel REE, per slegare il framework dall'hardware. Con una ulteriore TA sarebbe possibile per un produttore, definire le funzionalità per interagire con l'NFC, rendendo il sistema ancora più simile al funzionamento di un SE,

che riceve direttamente i pacchetti NFC.

Attualmente il framework implementa le funzionalità standard definite in ISO 7816. Una ulteriore estensione di TEA potrebbe includere altri standard per le carte contactless rendendo il sistema una piattaforma completa per l'emulazione di smart card.

Glossario

ADPU Application Protocol Data Unit

API Application programming interface

DF Dedicated file

EF Elementary file

NDK Native Development Kit

NFC Near Field COmmunication

PO Persistent Object

PoS Point of sale

REE Rich Execution Enviroment

TA Trusted Application

TEA Trusted Execution Enviroment

TEE Trusted Execution Enviroment

THCE Trusted Host-based Card Emulation

Bibliografia

[1] Alessandro Armando, Alessio Merlo e Luca Verderame. *Trusted Host-based Card Emulation*

[2] GlobalPlatform Device Technology. *TEE Internal Core API Specification v1.1*

[3] GlobalPlatform Device Technology. *TEE Internal API Specification v1.0*

[4] GlobalPlatform Device Technology. *TEE Client API Specification v1.0*

[5] Ugo Chirico. *Programmazione delle smart card*

[6] T. Janssen, M. Zandstra. *White paper - HCE security implications, analyzing the security aspects of HCE*

[7] K. Kostiainen, J.-E. Ekberg, N. Asokan, A. Rantala. *On-board credentials with open provisioning*

[8] J.-E. Ekberg, K. Kostiainen, N. Asokan. *The Untapped Potential of TEEs on Mobile Devices*

[9] Smart Card Alliance. *The Three Approaches to NFC Security - SE, TEE & HCE*

[10] Security Task Force. *A Security Analysis of NFC Implementation in the Mobile Proximity Payments Environment*

[11] S. Drimer and S. J. Murdoch. *Keep Your Enemies Close: Distance Bounding Against Smartcard Relay Attacks*

[12] GlobalPlatform Device Technology. *Trusted User Interface API*

[13] ISO/IEC 14443:2011 *Identification cards – Contactless integrated circuit cards – Proximity cards – Part 3: Initialization and anticollision*

[14] GP Hancke, KE Mayes, K Markantonakis. *Confidence in smart token proximity: Relay attacks revisited*

[15] Borko Lepojevic, Bojan Pavlovice Aleksandar Radulovic. *Implementing NFC service security – SE VS TEE VS HCE*

[16] Underwriters Laboratories.*Transaction security*

[17] ISO/IEC 7816 *Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*

[18] Visa Public. *Transaction acceptance device guide (TADG)*

[19] Schyter Tool. *https://www.cs.ox.ac.uk/people/cas.cremers/scyther/*

[20] GlobalPlatform Device Technology. *TEE Sockets API Specification v1.0, GPD_SPE_100*

[21] Whitfield Diffie, Paul C. Van Oorschot, Michael J. Wiener. *Authentication and authenticated key exchanges*

[22] GlobalPlatform Device Technology. *TEE Sockets API Specification v1.0*

[23] GlobalPlatform Device Technology. *TEE Trusted User Interface API Specification v1.0*

[24] Tavendo. *Autobahn - http://tavendo.com/*

[25] A. Vasudevan, E. Owusu, Z. Zhou J. Newsome, and J. McCune. *Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?*