



Verifica dinamica dei permessi di accesso in un'estensione del modello di sicurezza di Android

Giulio Puri

Tesi presentata per il conseguimento del titolo di

DOTTORE MAGISTRALE
IN
INGEGNERIA INFORMATICA

Relatore: Chiar.^{mo} Prof. Alessandro Armando

Correlatori: Dott. Gabriele Costa

Ing. Alessio Merlo



Dipartimento di Ingegneria, Bioinformatica, Robotica e Ingegneria dei Sistemi
Università degli studi di Genova

19 Settembre 2014

Indice

Elenco delle Figure	iii
Lista delle Tabelle	iv
Introduzione	v
1 Preliminari Tecnici	1
1.1 Android	1
1.1.1 Application Framework	2
1.1.2 Security Framework	3
1.1.3 Limiti del Security Framework di Android	5
1.2 Alcuni linguaggi per la specifica di politiche di sicurezza	5
1.2.1 Role-Based Access Control	5
1.2.2 Logiche temporali	6
2 Caso di Studio	8
2.1 Servizi di MaplePay.com	8
2.2 Funzionalità dell'applicazione	10
2.3 Struttura di MaplePay	10
2.4 Requisiti di sicurezza	13
3 Estensione del Modello di Sicurezza di Android	16
3.1 Modellazione delle componenti e delle configurazioni	16
3.2 Definizione e validazione delle politiche	17
3.3 Semantica operazionale	20
3.4 Validazione come problema di soddisfabilità	26
4 Secure Component Provider	27
4.1 Architettura e funzionamento del prototipo	27
4.2 Organizzazione di SCPApp	30
4.2.1 Request Receiver	30
4.2.2 Component Stacks Manager	32
4.2.3 SAT-Solver	35
4.2.4 Application Installer	35
4.2.5 Security Database	36
4.3 Struttura di SCPLib	37
4.4 Problematiche riscontrate e scelte implementative	39

5 Risultati e Test	43
5.1 Obiettivo dei test	43
5.2 Struttura degli esperimenti	43
6 Conclusioni e Sviluppi Futuri	48
Bibliografia	50

Elenco delle Figure

1.1	Software Stack di Android	2
1.2	Permessi di applicazione	4
1.3	Esempio di formula CTL	7
1.4	Esempio di formula LTL	7
2.1	Processo di pagamento di MaplePay.com	9
2.2	Pagamento verso contatti	10
2.3	Pagamento verso e-commerce	11
2.4	Architettura di MaplePay	11
3.1	Valutazione degli scope	18
3.2	Struttura di MaplePay con politiche di sicurezza	20
3.3	Evoluzione della configurazione di sistema 1-4	23
3.4	Evoluzione della configurazione di sistema 5-8	24
3.5	Tassonomia del linguaggio di specifica delle politiche	25
4.1	Struttura del prototipo	30
5.1	Test componenti individuate	46
5.2	Test ritardi misurati	47

Lista delle Tabelle

2.1 Azioni di MaplePay	13
4.1 Gerarchia dei processi	41
5.1 Componenti e permessi delle applicazioni più diffuse	44

Introduzione

I dispositivi mobili fanno oramai parte della vita quotidiana di milioni di persone. La loro diffusione aumenta di giorno in giorno e spesso ad essi vengono affidati dati sensibili, rendendoli delle vere e proprie chiavi di accesso verso molti servizi. Mentre da un lato tutto ciò rende più semplice l'accesso a molte informazioni, dall'altro costituisce un singolo punto critico che necessita di adeguati sistemi di sicurezza.

Tra tutti, Android risulta essere il sistema operativo mobile più diffuso al mondo [1], pertanto, la ricerca e lo studio dei suoi meccanismi di sicurezza ricopre un ruolo molto importante nel miglioramento di queste tecnologie mobili sempre più diffuse.

Android prevede principalmente due meccanismi di sicurezza posti a differenti livelli della sua architettura: il primo, a livello Kernel, sfrutta il concetto di multi-utenza per creare un meccanismo di accesso ai dati basato su permessi utente, mentre il secondo, a livello applicativo, consiste in un sistema di permessi per la protezione dei dati e delle applicazioni. Quest'ultimo sfrutta delle *label* che gli sviluppatori devono includere nel proprio software all'interno di un file XML, detto *Manifest*, per poter accedere alle risorse sensibili del dispositivo, come specifici moduli hardware e collezioni di dati personali.

Durante la fase d'installazione delle applicazioni, i permessi richiesti vengono presentati all'utente, che decide se annullare o procedere con l'installazione concedendo i permessi richiesti. Questo è il primo e unico messaggio di avviso che, nella maggior parte dei casi, viene mostrato all'utente: successivamente, l'applicazione non dovrà più richiedere alcun permesso, in quanto già concessi in fase di installazione e mantenuti fino alla sua disinstallazione.

Questo approccio, sebbene cerchi di semplificare la comprensione dell'utente sui permessi richiesti dalle applicazioni e non interrompa la loro fase di esecuzione con messaggi di avviso che potrebbero intaccare l'esperienza d'uso, presenta diversi aspetti negativi, tra cui:

- il modello “tutto o niente”, che obbliga gli utenti a concedere in blocco tutti i permessi richiesti dalle applicazioni (pena la cancellazione dell'installazione);
- l'associazione dei permessi alle applicazioni come un singolo agente, piuttosto che ai singoli elementi che le compongono, che potrebbe comportare la concessione di permessi per funzionalità non richieste dall'utente;

- la mancata garanzia che l'utente capisca correttamente quali permessi stia acquisendo l'applicazione e di come essi possano essere utilizzati nei confronti, sia della sicurezza della stessa, sia in quella delle altre applicazioni installate;
- l'impossibilità da parte dello sviluppatore di poter stabilire regole per poter controllare l'operato dell'utente;

In questa tesi verrà presentata un'estensione del supporto di sicurezza di Android, fornendo agli sviluppatori la possibilità di specificare delle politiche di sicurezza e di poterle assegnare, assieme ai permessi, ai singoli elementi costitutivi delle applicazioni. In questo modo, sarà possibile includere all'interno delle stesse applicazioni delle regole utili a governare sia la loro esecuzione sia l'interazione tra le loro componenti.

Ciò sarà permesso sfruttando le specifiche strutturali del file manifest di Android e per mezzo di un applicativo centrale, la *Secure Component Provider Application* (SCPApp), che, durante i processi d'invocazione delle componenti delle applicazioni, si occuperà di trasformare le politiche e i permessi specificati in problemi di soddisfacibilità proposizionale (SAT), che verranno processati mediante un SAT-Solver.

L'approccio sviluppato in questa tesi offre alcuni vantaggi che ne rendono immediata l'applicazione e la portabilità sui dispositivi mobili.

Nessuna personalizzazione del sistema operativo.

L'approccio presentato è costruito sul piano applicativo del software stack di Android, pertanto, può essere applicato a dispositivi e configurazioni già esistenti.

Nessuna richiesta di conoscenze aggiuntive per gli sviluppatori.

Il pacchetto di librerie fornito (SCPLib), che mette a disposizione un meccanismo per l'interazione tra applicazioni e SCPApp, e il meccanismo utilizzato per la specifica dei permessi sulle componenti, sono del tutto simili a quelli già forniti dal sistema operativo Android.

Alta granularità per la specifica di permessi e di politiche.

Tali informazioni di sicurezza vengono associate alle singole componenti che costituiscono l'applicazione.

Ampio raggio di applicazione delle politiche.

Le politiche di sicurezza specificabili possono essere di tipo *Dirette*, *Locali* e *Globali*, in modo da poter imporre differenti tipologie di restrizioni sul contesto d'esecuzione della componente. Inoltre possono assumere la proprietà *Sticky*.

Garanzia di nessuna configurazione illegale a run-time.

Le interazioni tra le componenti vengono verificate dinamicamente durante la loro fase d'invocazione. Nel caso in cui si verificassero delle violazioni delle proprietà di sicurezza, tale fase d'invocazione verrebbe interrotta prima del suo termine, impedendo il verificarsi di configurazioni illegali.

Struttura della Tesi

Nel Capitolo 1 verranno presentati brevemente i preliminari tecnici e il background necessario per comprendere correttamente la trattazione. Il Capitolo 2, introdurrà un caso d'uso che fornirà un contesto realistico a cui sarà fatto riferimento nei capitoli successivi e permetterà di mostrare i limiti del framework di sicurezza di Android. Nel Capitolo 3, verrà presentata l'estensione del modello di sicurezza considerata mentre, nel Capitolo 4, si descriverà dettagliatamente il prototipo implementato. Nel Capitolo 5 si mostreranno i risultati e le prestazioni ottenute col prototipo e, infine, all'interno del Capitolo 6, saranno presentate le conclusioni e alcuni possibili sviluppi futuri.

Capitolo 1

Preliminari Tecnici

All'interno di questo capitolo verranno introdotti gli aspetti tecnici legati a questa tesi. Nella Sezione 1.1 verrà fornita una panoramica del sistema operativo Android, presentando il funzionamento degli Application e Security Framework di cui dispone. In Sezione 1.2, invece, verranno presentati alcuni linguaggi per la specifica di politiche di sicurezza.

1.1 Android

Android è un sistema operativo per dispositivi mobili nato da un progetto open source [2]. Esso è basato su kernel Linux e ha riscosso nel corso degli anni un largo successo proprio per la sua natura open che ne permette una facile personalizzazione. Con 1,5 milioni di attivazioni giornaliere [1], risulta essere il principale sistema operativo per smartphone, largamente utilizzato anche in contesti aziendali ed enterprise.

Strutturalmente, come mostrato in Fig. 1.1, Android è costituito da una serie di *layer* che forniscono differenti funzionalità al sistema. Alla base vi è il Kernel di Linux, che si occupa della gestione dei moduli hardware, della memoria, delle risorse computazionali e di alcuni aspetti fondamentali relativi alla sicurezza che verranno trattati più avanti. Su di esso si poggia lo strato nativo, costituito dalle librerie di sistema per la gestione di differenti tipologie di dati, come file multimediali, database, grafica 2D e 3D, e per il supporto a protocolli di sicurezza come SSL. Fanno parte di questo layer anche il modulo Android Runtime, che fornisce il supporto per le applicazioni Java mediante le Core Libraries e la Dalvik Virtual Machine (DVM). Le prime sono una versione adattata delle librerie base di Java per il sistema Android, mentre, la DVM, è una Virtual Machine (VM) che a differenza di quella tradizionale opera in file *.dex* anzichè *.class*, e che permette, come comportamento standard, l'esecuzione di più sue istanze contemporaneamente. Infine si ha l'Application Framework, che fornisce le funzionalità di base del dispositivo alle applicazioni che interagiscono con lui.

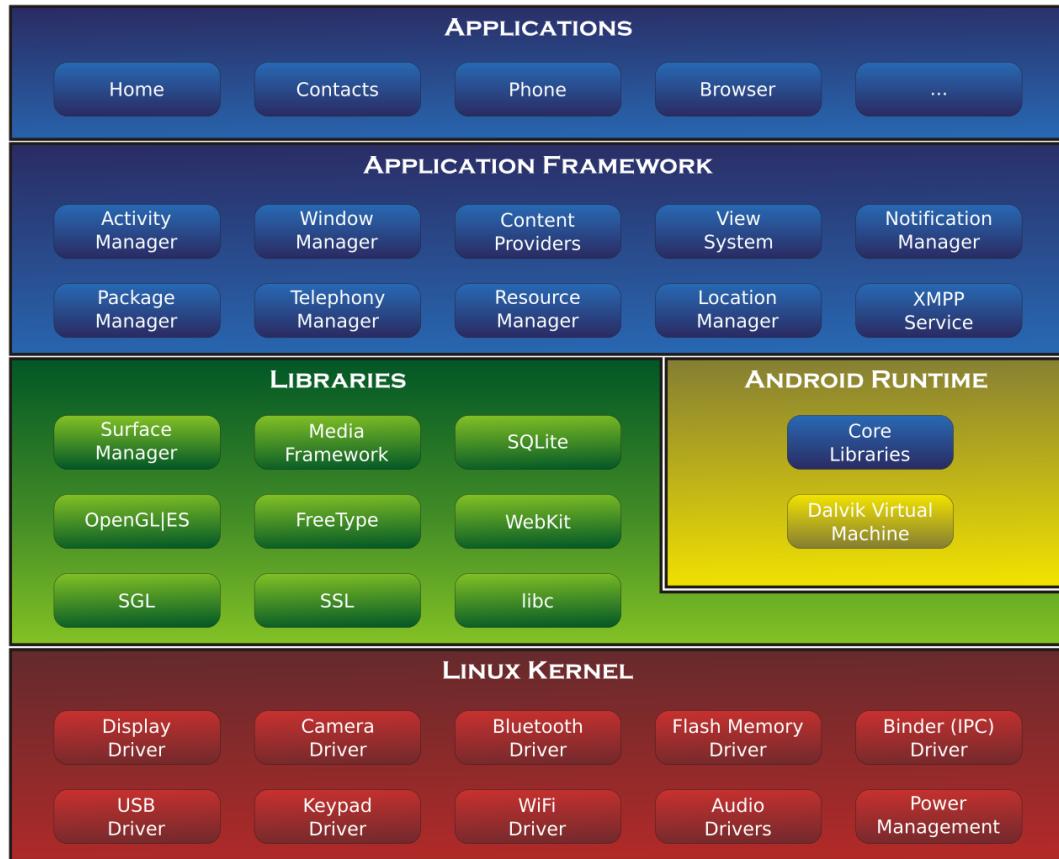


FIGURA 1.1: Software Stack di Android.

1.1.1 Application Framework

Le applicazioni destinate ai dispositivi Android sono principalmente scritte mediante codice Java, per il quale sono fornite tutte le librerie necessarie per interagire col sistema operativo. Inoltre, anche se sconsigliato se non strettamente necessario [3], le applicazioni possono includere porzioni di codice nativo, per esempio scritto in C o C++.

In entrambi i casi, le applicazioni sono costituite da una o più componenti, ossia dei blocchi strutturali che forniscono dei punti di accesso verso l'applicazione sia al sistema operativo sia all'interazione dell'utente. Esistono quattro tipologie di componenti: ognuna possiede un suo ruolo specifico e un suo ciclo di vita, e definisce determinati comportamenti dell'applicazione.

- *Activity*: forniscono una schermata che permette all'utente di interagire con l'applicazione;
- *Broadcast Receiver*: rispondono alle richieste inviate in broadcast dal sistema operativo o da altre applicazioni;
- *Content Provider*: gestiscono l'accesso a insiemi strutturati di dati;
- *Service*: permettono l'esecuzione di processi in background che non richiedono un'interazione con l'utente;

Per una corretta esecuzione dell'applicazione, il sistema operativo deve conoscere alcune sue caratteristiche e informazioni fondamentali. Queste vengono elencate all'interno del file Manifest, analizzato da Android durante la fase di installazione di ogni applicazione. Esso contiene, tra le altre cose, il *package* dell'applicazione, che fungerà da identificativo univoco, l'elenco delle librerie che utilizza e quello delle componenti che la costituiscono.

Il comportamento standard di Android prevede che ogni componente sia invocabile da qualunque applicazione. Ciò può avvenire sia per mezzo di un opportuno messaggio, detto *intent*, che dichiara l'intento di una data applicazione di voler avviare l'esecuzione di un'activity, di un service o di un broadcast receiver, sia per mezzo di un particolare oggetto, detto *contentResolver*, che fornisce un'interfaccia di accesso ai content provider registrati all'interno del sistema.

Gli intent possono appartenere a due tipologie.

- *Intent Esplicativi*, che specificano la componente da invocare indicandone il nome della classe comprensivo di package;
- *Intent Impliciti*, che dichiarano una generica azione da eseguire lasciando al sistema operativo il compito di decidere quale componente invocare;

1.1.2 Security Framework

Alla base del security framework si ha la sicurezza a livello di Kernel Linux, dove un modello multi-utente basato sui permessi garantisce il mutuo isolamento delle risorse ed evita il verificarsi di interazioni malevole tra utenti, che potrebbero comportare il collasso delle risorse computazionali. Ad ogni applicazione viene infatti associato un *Linux User Id* (UID) univoco, ad essa sconosciuto e calcolato sulla base della firma che ogni sviluppatore deve porre su ogni applicazione che intende distribuire. Ogni applicazione esegue poi su un processo dedicato: in questo modo si crea una *SandBox* di esecuzione a livello Kernel, che si applica a tutti gli elementi software superiori, come librerie di sistema, applicazioni native e ambiente di esecuzione run-time per le applicazioni Java.

Il kernel fornisce anche dei meccanismi di comunicazione tra processi estesi: oltre che ai classici meccanismi Unix come *Socket*, memorie condivise e *Pipe*, sono resi disponibili anche Binder, Intent e Content Provider, ossia dei meccanismi ottimizzati che garantiscono alte prestazioni e facilitano la comunicazione interna ed esterna dei processi.

A livello applicazione, la sicurezza viene garantita per mezzo di un meccanismo basato su permessi. In Android le risorse sensibili di sistema sono accessibili esclusivamente tramite il sistema operativo e, le applicazioni, per poter accedere alle API di interesse, devono richiedere determinati permessi. Questi devono essere dichiarati all'interno del Manifest, e possono riguardare sia componenti hardware, come i moduli Bluetooth, GPS e NFC, sia risorse software, come i Content Provider di

accesso ai dati utente. Durante il processo di installazione, l'utente viene informato sui permessi richiesti dall'applicazione e può decidere se completarla conferendo all'applicazione i permessi richiesti o se annullarla (Fig. 1.2). Una volta ottenuti i permessi, l'applicazione li mantiene fino a quando resta installata sul dispositivo.

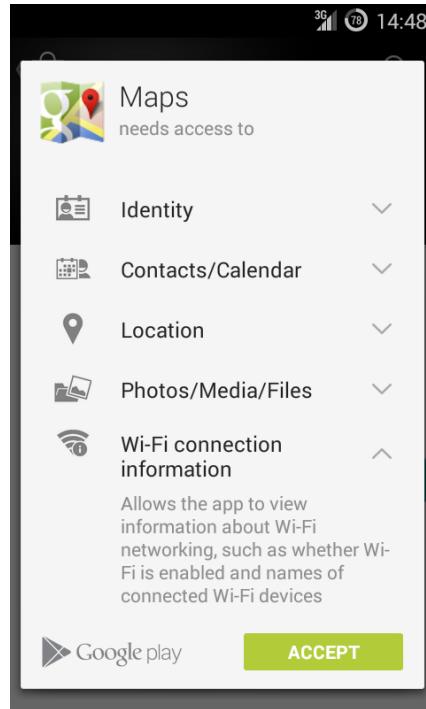


FIGURA 1.2: Permessi richiesti dall'applicazione Maps durante la fase di installazione.

I permessi permettono anche la definizione di alcune regole: le componenti Service, Activity e Broadcast Receiver possono infatti dichiarare i permessi che le applicazioni chiamanti devono possedere per poterle invocare. Invece, gli Intent inviati in broadcast, possono specificare dei permessi che i Broadcast Receiver devono possedere per poter ricevere il messaggio. Infine, i Content Provider possono richiedere dei permessi per la lettura e/o scrittura dei dati che gestiscono.

Oltre ai permessi standard di sistema, gli sviluppatori possono definirne di propri per stabilire ulteriori restrizioni. I permessi di questo tipo possiedono un attributo indicante il livello di protezione, il cui scopo è identificare il livello di rischio del permesso e stabilirne, sia le regole di accesso, basate sulla firma delle applicazioni coinvolte, sia come e se il sistema debba informare l'utente durante la fase di conferimento del permesso. I livelli di protezione associabili sono i seguenti.

- *Normal*: dedicato ad applicazioni a basso rischio. Il permesso viene acquisito in fase d'installazione, senza la necessità di un'approvazione esplicita dell'utente;
- *Dangerous*: dedicato ad applicazioni a rischio elevato, ossia che richiedono accesso a dati utente o che potrebbero acquisire il controllo del dispositivo. Richiedono un'approvazione esplicita dell'utente prima di ogni esecuzione.

- *Signature*: dedicato a permessi acquisibili solo da applicazioni firmate dallo stesso sviluppatore dell'applicazione in cui è definito il permesso.
- *SignatureOrSystem*: dedicato a permessi acquisibili solo da applicazioni firmate dallo stesso sviluppatore dell'applicazione in cui è definito il permesso o da applicazioni di sistema.

1.1.3 Limiti del Security Framework di Android

Il framework di sicurezza di Android è stato oggetto di molti studi che ne hanno dimostrato limiti e vulnerabilità [4–6]. Per esempio, il meccanismo di comunicazione degli intent, se non utilizzato in maniera adeguata, fornisce ad applicazioni malevole una via di accesso per l'estrapolazione di dati e dell'informazione utente [7].

Altri errori di progettazione di applicazioni e di componenti di sistema, come mostrato in [8, 9], possono invece fornire la possibilità di eseguire attacchi di tipo privilege escalation, che hanno come obiettivo quello di conferire alle applicazioni permessi d'accesso a informazioni a loro normalmente precluse.

Di conseguenza, molti autori [10–13] hanno proposto modifiche o addirittura ridefinito il security framework.

1.2 Alcuni linguaggi per la specifica di politiche di sicurezza

In letteratura sono stati presentati differenti linguaggi per la specifica di politiche. Tipicamente, ogni formalismo affronta specifici aspetti della sicurezza informatica, come l'accesso al sistema o l'utilizzo delle sue risorse. Nel seguito, riportiamo alcuni esempi utili per caratterizzare il potere espressivo del linguaggio di politiche utilizzato in questa tesi.

1.2.1 Role-Based Access Control

RBAC (Role-Based Access Control) [14] è un modello per la definizione e applicazione di politiche di controllo degli accessi. Brevemente, esso prevede di assegnare uno o più *ruoli* ai soggetti che interagiscono con le risorse di un dato sistema. Ogni ruolo è associato a uno o più *permessi di accesso* alle risorse del sistema.

Esistono vari linguaggi per specificare politiche RBAC. Consideriamo ad esempio l'approccio logico proposto in [15]. Esso definisce il concetto di dominio D di un sistema come una quadrupla $\langle S, A, O, R \rangle$, dove S è un insieme di soggetti, A un insieme di azioni, O un insieme di oggetti e R un insieme di ruoli. Uno stato del sistema è definito come un sottoinsieme $\Pi \subseteq S \times A \times O \times R$. Ad esempio, $(s, a, o, r) \in \Pi$ significa che il soggetto s ha nello stato Π , il permesso di eseguire l'azione a sull'oggetto o attraverso il ruolo r .

Ogni stato del sistema deve essere validato rispetto a una condizione ψ definita dalla seguente sintassi.

$$\psi, \psi' ::= P(s, a, o, r) \mid \top \mid \perp \mid \psi \wedge \psi' \mid \psi \vee \psi'$$

Dove $P(s, a, o, r)$ è un predicato quaternario che definisce le regole di accesso.¹

Quindi, la soddisficiabilità di uno stato Π rispetto una data condizione ψ , in simboli $\Pi \models \psi$, è definita come segue.

$$\begin{aligned} \Pi \not\models \perp \\ \Pi \models \top \\ \Pi \models P(s, a, o, r) &\iff (s, a, o, r) \in \Pi \\ \Pi \models \psi \wedge \psi' &\iff \Pi \models \psi \text{ e } \Pi \models \psi' \\ \Pi \models \psi \vee \psi' &\iff \Pi \models \psi \text{ o } \Pi \models \psi' \end{aligned}$$

Esempio 1.1. Consideriamo un sistema caratterizzato dai seguenti insiemi di azioni, oggetti, soggetti e ruoli:

$$\begin{aligned} A &= \{a_1, a_2\} & O &= \{o_1, o_2\} \\ S &= \{s_1, s_2\} & R &= \{r_1, r_2\} \end{aligned}$$

La seguente matrice Π rappresenta gli stati del sistema.

$$\Pi \triangleq \begin{array}{|c|c|c|c|} \hline & s_1 & a_1 & o_1 & r_1 \\ \hline s_1 & & a_2 & o_2 & r_2 \\ \hline \end{array}$$

Data la condizione $\psi = P(s_1, a_1, o_1, r_1) \wedge P(s_1, a_2, o_1, r_1)$, si ha che $\Pi \models P(s_1, a_1, o_1, r_1)$ e che $\Pi \not\models P(s_1, a_2, o_1, r_1)$, pertanto, $\Pi \not\models \psi$. ■

1.2.2 Logiche temporali

Le logiche temporali sono logiche modali le cui modalità vengono valutate contro le configurazioni di un modello nel tempo.

CTL

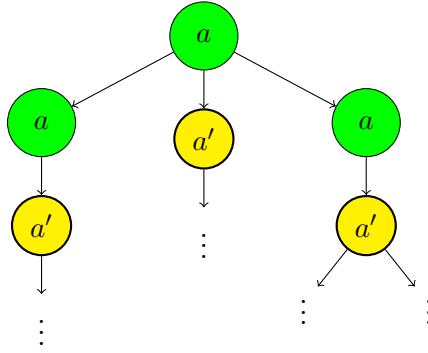
La Computation Tree Logic [16] è una logica temporale che permette di esprimere proprietà delle strutture di modelli computazionali rappresentati come alberi d'esecuzione.

Dato un insieme di proposizioni atomiche AP , con $a \in AP$, l'insieme di formule CTL definibili su AP è dato da tutti e soli i termini generabili dalla seguente grammatica.

$$\psi, \psi' ::= a \mid \neg \psi \mid \psi \vee \psi' \mid \psi \wedge \psi' \mid AX \psi \mid AG \psi \mid A[\psi U \psi']$$

Dato uno stato S , $AX\psi$ è vera sse ψ è vera per ogni immediato successore di S , $AG\psi$ è vera sse ψ è vera su tutti gli stati dell'albero radicato in S , $A[\psi U \psi']$ è vera sse su tutti i percorsi radicati in S , ψ è vera finché non vale ψ' .

¹Questa presentazione semplificata trascura il fatto che P possa essere definito anche su un dominio di variabili. Si veda [15] per una dettagliata trattazione.

FIGURA 1.3: Rappresentazione di un albero che soddisfa la formula CTL $A[a U a']$.

Esempio 1.2. Consideriamo la seguente formula CTL:

$$A[a U a']$$

L'albero mostrato in Fig. 1.3 soddisfa la politica appena definita. ■

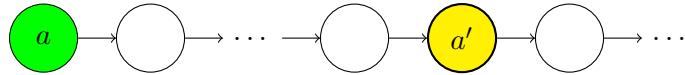
LTL

La Linear Temporal Logic [17] è una logica temporale che, a differenza di CTL, si applica a sequenze lineari di stati che rappresentano la computazione come una serie di cambi di stato nel tempo.

Dato un insieme di proposizioni atomiche AP con $a \in AP$. L'insieme di formule LTL definibili su AP è dato da tutti e soli i termini generabili dalla seguente sintassi:

$$\psi, \psi' ::= a \mid \neg\psi \mid \psi \vee \psi' \mid \psi \wedge \psi' \mid X\psi \mid G\psi \mid F\psi \mid \psi U \psi'$$

Dato uno stato S , $X\psi$ è vera sse ψ è vera nell'immediato successore di S , $G\psi$ è vera sse ψ è vera in S e in tutti i suoi successori, $F\psi$ è vera sse ψ è vera in un qualche stato successivo a S , $\psi U \psi'$ è vera sse ψ è vera fino a che non viene raggiunto uno stato in cui vale ψ' .

FIGURA 1.4: Rappresentazione di una sequenza che soddisfa la formula LTL $a \rightarrow Fa'$.

Esempio 1.3. Consideriamo la seguente formula LTL:

$$a \rightarrow Fa'$$

La sequenza mostrata in Fig. 1.4 soddisfa la politica appena definita. ■

Capitolo 2

Caso di Studio

In questo capitolo presentiamo un caso di studio che useremo nel corso della trattazione della tesi come contesto realistico per illustrare i contributi proposti. In particolare, verrà simulato il processo di progettazione di un'applicazione per il sistema Android da parte di un'azienda che offre servizi di pagamento e di invio di denaro online, MaplePay.com.

In Sezione 2.1, si descriveranno i servizi offerti dalla società MaplePay.com. Nella seconda, Sezione 2.2, si presenteranno i requisiti funzionali che l'applicazione dovrà soddisfare, mentre, nella Sezione 2.3, ne verrà descritta la struttura e i meccanismi di interazione con l'ambiente Android.

Infine, nella Sezione 2.4, verranno presentati i requisiti di sicurezza dell'applicazione, mostrando come il framework di sicurezza di Android limiti la loro implementazione.

2.1 Servizi di MaplePay.com

MaplePay.com è una società online che offre servizi di pagamento e di trasferimento di denaro tramite Internet. Il loro funzionamento prevede una fase iniziale di registrazione, in cui i clienti creano un account personale inserendo i propri dati e le proprie carte di credito. Successivamente, gli utenti hanno la possibilità di inviare denaro ad altri account MaplePay.com, specificando semplicemente l'indirizzo email che il destinatario ha utilizzato durante la fase di registrazione. L'accesso al proprio account, sia per visualizzarne le informazioni sia per eseguire qualsiasi transazione, richiede una fase di autenticazione mediante *username* e *password*. Inoltre, ad ogni operazione il servizio registra il movimento eseguito all'interno di uno storico personale, in modo da poter generare un estratto conto.

Le transazioni possono essere eseguite sia tramite web page, collegandosi al portale di MaplePay.com, sia per mezzo di un web service, interrogabile da web application esterne. Nel secondo caso MaplePay.com mette a disposizione delle API per poter interagire coi i propri servizi per avviare pagamenti direttamente dai siti web delle società di e-commerce.

Nel caso di pagamento attraverso il sito di MaplePay.com, l'utente deve semplicemente eseguire il login. Fatto ciò, egli avrà accesso ai propri dati, al proprio storico delle transazioni e alla schermata per l'invio di denaro. Invece, nel caso di pagamento da sito e-commerce, il processo di avvio della transazione prevede il protocollo descritto in seguito e mostrato in Fig. 2.1.

1. Il sito di e-commerce effettua una richiesta di pagamento tramite API di MaplePay.com per poter avviare il processo di pagamento;
2. MaplePay.com risponde alla richiesta mediante una chiave di autorizzazione al pagamento;
3. Per mezzo della chiave ricevuta, il sito di e-commerce re-indirizza l'utente sul portale di MaplePay.com in cui gli verrà richiesto di autenticarsi per mezzo delle proprie credenziali;
4. Eseguita l'autenticazione, l'utente visualizza un resoconto della transazione e sceglie se confermare o annullare il pagamento;

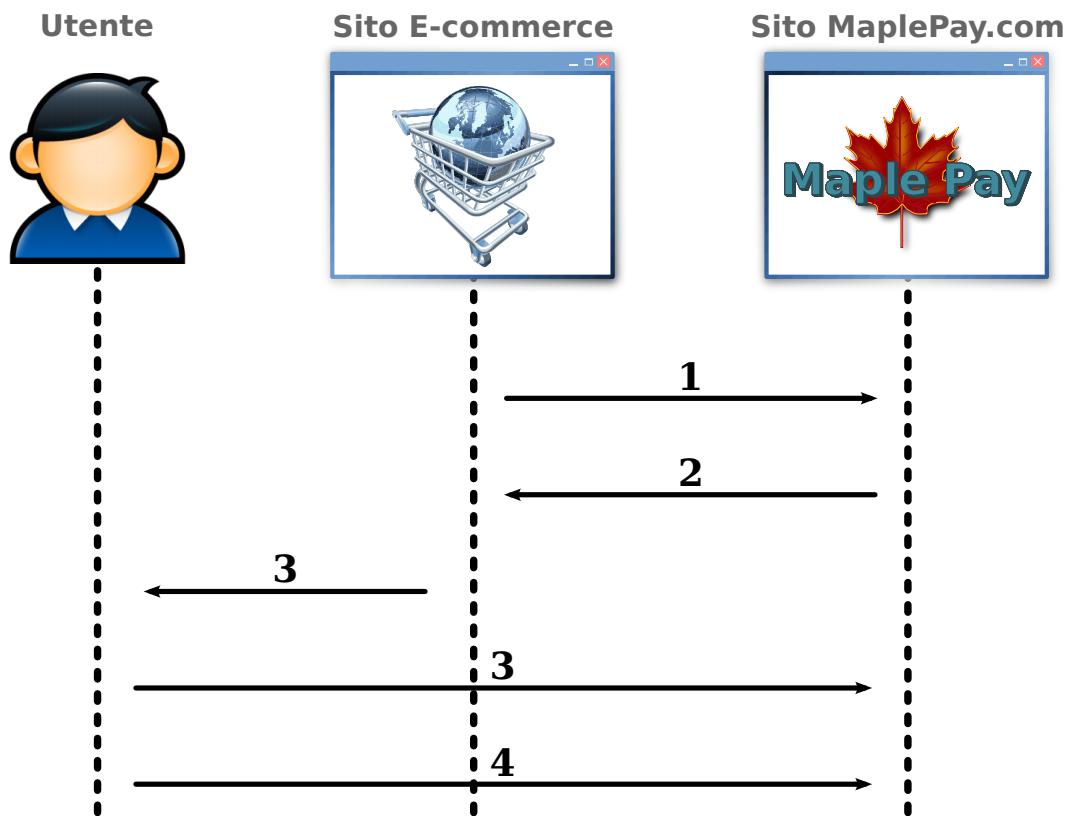


FIGURA 2.1: Flusso di esecuzione del processo di pagamento da sito e-commerce mediante API di MaplePay.com.

Per rendere il servizio sempre più utile e completo, gli sviluppatori di MaplePay.com vogliono progettare ed implementare un'applicazione mobile Android. Essa deve fornire le stesse funzionalità già disponibili dal servizio via web, garantendo inoltre alcuni requisiti funzionali e di sicurezza specifici per l'ambiente mobile.

2.2 Funzionalità dell'applicazione

L'obiettivo principale dell'applicazione MaplePay è permettere agli utenti di eseguire pagamenti dai propri dispositivi mobili. In particolare, dovranno essere fornite le seguenti modalità di pagamento.

Pagamenti verso Contatti.

L'utente deve poter inviare denaro a contatti aventi un account registrato a MaplePay.com. Un'interfaccia utente dovrà permettere l'inserimento manuale dell'importo, del destinatario prima di poter procedere con la transazione (Fig. 2.2);

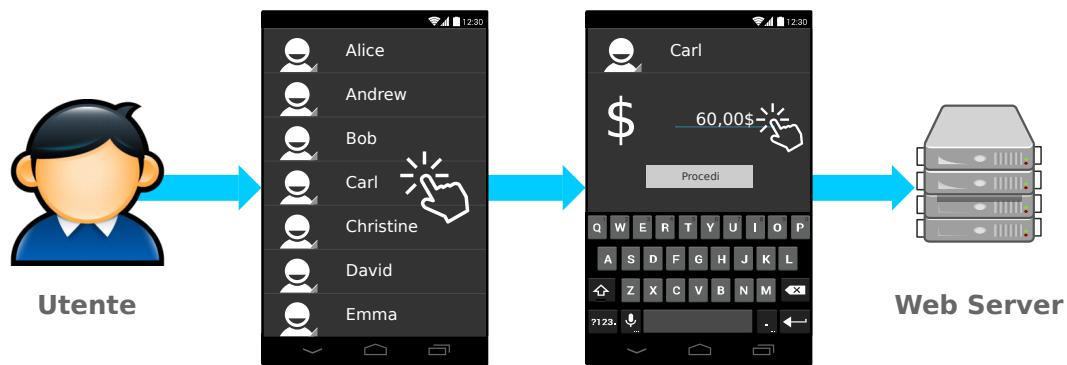


FIGURA 2.2: Esempio di flusso di esecuzione nel caso di pagamenti verso contatti.

Pagamenti verso E-Commerce.

Applicazioni esterne devono poter avviare procedure di pagamento, verso i propri account MaplePay.com, per l'acquisto di beni o di servizi da parte dell'utente.

Distinguiamo due tipologie: *micro pagamenti*, d'importo inferiore a 25\$, che possono eseguiti in maniera automatica dall'applicazione senza la necessità di autorizzazioni da parte dell'utente (Fig. 2.3), e *pagamenti normali*, di qualsiasi importo, che richiedono invece un'autorizzazione esplicita da parte dell'utente (Fig. 2.3);

Oltre a queste funzionalità principali, l'utente deve avere la possibilità di accedere alle informazioni del proprio account MaplePay.com. In particolare, deve poter visualizzare il proprio conto spese e scaricare sul dispositivo lo storico delle transazioni in modo da poterlo esaminare per mezzo di applicazioni di visualizzazione di documenti, fornite dall'ambiente esterno o dal sistema operativo.

2.3 Struttura di MaplePay

Come spiegato nella Sezione 1.1.1, l'applicazione è costituita da componenti, in particolare nove, ognuna delle quali fornisce una specifica funzionalità. La Fig. 2.4 le



FIGURA 2.3: Esempio di flusso di esecuzione nel caso di pagamenti verso E-Commerce.

riassume, mostrandone i principali flussi di interazione e la loro collocazione all'interno della struttura dell'applicazione. Vengono inoltre mostrati i punti di ingresso (○) e di uscita (●), che rappresentano i canali d'interazione dell'applicazione con l'ambiente esterno per, rispettivamente, avviare le procedure di invio di denaro e di apertura degli estratti conto.

MainActivity

Fornisce all'utente il punto di accesso tramite il processo standard d'invocazione delle applicazioni Android. Si occupa di controllare che l'utente esegua la fase di autenticazione e, successivamente, mostra le funzionalità rese disponibili dall'applicazione, come la possibilità di inviare un pagamento a un contatto conosciuto o visualizzare lo storico dei movimenti;

LoginActivity

Mostra una UI per l'autenticazione dell'utente per mezzo di Userid e di Password; tali dati, oltre che a fornire un'autenticazione locale a livello di dispositivo, verranno inoltre utilizzati per la generazione del token necessario per

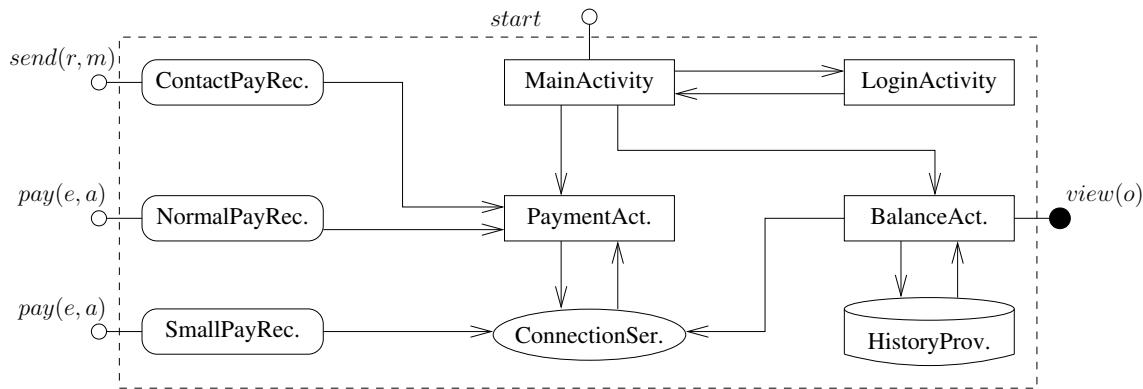


FIGURA 2.4: La architettura dell'applicazione MaplePay.

l’interazione col web service di MaplePay.com all’interno di ConnectionService;

PaymentActivity

Interfaccia utente multimodale che riassume o permette l’inserimento dei dati dei pagamenti da inviare, come destinatario, importo e causale, a seconda del flusso di esecuzione in cui viene invocata. In qualunque caso, per avviare la transazione tramite ConnectionService, richiede che l’utente abbia eseguito la procedura di login; pertanto, se l’applicazione non è stata precedentemente avviata tramite la mainActivity, prima di invocare il service viene invocata la loginActivity;

BalanceActivity

Componente che si occupa sia di contattare tramite ConnectionService il server di MaplePay.com per scaricare le nuove informazioni sull’account, come il saldo aggiornato o nuovi estratti conto, sia di presentare, per mezzo di una lista, tutti i documenti passati già memorizzati sul dispositivo. Ogni elemento listato può inoltre essere aperto e visualizzato in applicazioni esterne di visualizzazioni di documenti;

ContactPaymentReceiver

Fornisce alle applicazioni di gestione dei contatti esterne un’interfaccia per l’invio di denaro verso un contatto specifico;

NormalPaymentReceiver

Fornisce alle applicazioni di e-commerce esterne un’interfaccia di avvio di pagamenti solo per importi maggiori di 25\$;

MicroPaymentReceiver

Fornisce alle applicazioni di e-commerce esterne un’interfaccia di avvio di micro pagamenti di importo minore o pari a 25\$;

ConnectionService

Componente che media l’accesso al web service di MaplePay.com. Si occupa dell’autenticazione dell’utente, dell’avvio delle transazioni e del download delle informazioni dell’account;

HistoryProvider

Content Provider che si occupa di gestire i dati locali dell’applicazione, ossia gli estratti conto precedentemente memorizzati sul dispositivo;

Come descritto nella Sezione 2.2, l’applicazione fornisce opportune interfacce per l’interazione con l’ambiente Android, per garantire sia la possibilità di avviare da applicazioni esterne le procedure di pagamento, sia quella di aprire gli estratti conto tramite applicazioni di visualizzazione di documenti. Tali interazioni avvengono per mezzo del meccanismo di intent. La Tabella 2.1 mostra le azioni che vengono specificate per interagire con MaplePay, indicando anche la presenza di eventuali payload e se si tratta di azioni definite dal sistema Android o introdotte da MaplePay. Nel dettaglio:

Start, avvia l’activity principale dell’applicazione. Viene utilizzata da Android per avviare l’applicazione alla pressione della sua icona da parte dell’utente.

Send, invia il messaggio m al destinatario r . Viene utilizzata dalle applicazioni esterne per avviare la procedura di invio di denaro verso un contatto. Il messaggio m contiene al suo interno i dettagli del pagamento, come nome del destinatario e importo.

View, mostra all’utente un dato o . Viene inviato da MaplePay per invocare applicazioni di visualizzazione di documenti per poter esaminare gli estratti conto.

Pay, paga il totale a all’entità e . Viene inviato dalle applicazioni di e-commerce esterne per avviare le procedure di pagamento tramite MaplePay.

Azione	Abbrev.	Payload	Sistema
ACTION_MAIN	<i>start</i>	—	✓
ACTION_SEND	<i>send</i>	r, m	✓
ACTION_VIEW	<i>view</i>	o	✓
MP_PAYMENT	<i>pay</i>	e, a	

TABELLA 2.1: Le azioni di interfaccia gestite da MaplePay.

2.4 Requisiti di sicurezza

Oltre ai requisiti funzionali, l’implementazione di MaplePay deve soddisfare alcuni requisiti di sicurezza.

1. Devono poter accedere tutte e sole le applicazioni aventi un’autorizzazione o rilasciata da MaplePay.com per le aziende partner (*AP*), o dall’utente stesso per le applicazioni ritenute da lui affidabili (*UA*). Inoltre, mentre per i micro-pagamenti può valere sia l’una che l’altra, per i pagamenti normali deve esserci sempre quella esplicita dell’utente.
2. Le richieste di pagamento verso contatti possono provenire solo da applicazioni che godono effettivamente dei privilegi di accesso alle liste di contatti (*RCP*) associati a uno o più account utente (*GAP*).
3. L’inserimento delle credenziali di autenticazione deve avvenire senza il rischio che sensori del dispositivo possano estrapolare informazioni riservate [18, 19]. Ad esempio, microfoni e fotocamere frontali potrebbero catturare l’interazione dell’utente con la tastiera, permettendo l’individuazione delle credenziali di accesso.
4. Non deve essere consentito il rilascio di informazioni relative allo storico dell’utente da applicazioni di visualizzazione di documenti che utilizzano canali di comunicazione verso l’esterno del dispositivo, come Internet, Bluetooth e memoria SD.

Esempio 2.1. Tramite le politiche RBAC, è possibile definire dei ruoli che le applicazioni che vogliono interagire con MaplePay per inviare denaro devono ricoprire, in modo da modellare il requisito 1 appena descritto. In particolare, definiamo:

$$\begin{aligned} A &= \{send, pay\} & O &= \{ContactPay_{Rec}, NormalPay_{Rec}, MicroPay_{Rec}\} \\ S &= \{App_1, App_2\} & R &= \{UA, AP, NP, MP\} \end{aligned}$$

e la seguente matrice Π .

$\Pi \triangleq$	App_1	pay	$NormalPay_{Rec}$	UA
	App_1	pay	$NormalPay_{Rec}$	NP
	App_1	pay	$MicroPay_{Rec}$	UA
	App_1	pay	$MicroPay_{Rec}$	MP
	App_2	$send$	$ContactPay_{Rec}$	AP
	App_2	$send$	$ContactPay_{Rec}$	RCP
	App_2	$send$	$ContactPay_{Rec}$	GAP

È possibile eseguire il controllo degli accessi sugli stati Π mediante le seguenti condizioni.

$$\psi = P(s, pay, NormalPay_{Rec}, UA) \wedge P(s, pay, NormalPay_{Rec}, NP)$$

$$\psi' = P(s, pay, MicroPay_{Rec}, MP) \wedge$$

$$(P(s, pay, MicroPay_{Rec}, UA) \vee P(s, pay, MicroPay_{Rec}, AP))$$

$$\begin{aligned} \psi''' &= (P(s, send, ContactPay_{Rec}, AP) \vee P(s, send, ContactPay_{Rec}, UA)) \wedge \\ &(P(s, send, ContactPay_{Rec}, RCP) \wedge P(s, send, ContactPay_{Rec}, GAP)) \end{aligned}$$

Pertanto, si ha che per $s = App_1$, $\Pi \models \psi$ mentre $\Pi \not\models \psi'''$. ■

Esempio 2.2. Tramite formule CTL è possibile modellare il requisito 3. In particolare, definendo l'insieme delle proposizioni atomiche AP come $AP = \{c, m, l\}$, con c , accesso alla camera frontale, m , accesso al microfono e, l , schermata di login visibile, è possibile specificare la seguente formula ψ :

$$\psi = A [\neg(c \vee m) U \neg l]$$
■

Esempio 2.3. Tramite politiche LTL è invece possibile modellare il requisito 4, evitando che flussi non autorizzati di informazioni, diretti e indiretti, possano fuoriuscire dal contesto di esecuzione dell'applicazione.

Definendo l'insieme delle proposizioni atomiche $AP = \{r, n, w, b\}$, dove r indica l'avvenuto rilascio dei dati sulla memoria condivisa, n accesso a internet, w accesso in scrittura alla memoria condivisa e, b , accesso al trasmettitore bluetooth, è possibile specificare la seguente formula ψ' :

$$\psi' = G [r \rightarrow \neg F (n \vee w \vee b)]$$
■

Discussione

RBAC

Anche assumendo di poter individuare un meccanismo per associare i permessi di Android ai ruoli, non è intuitivo, sebbene esista qualche proposta preliminare [20], come caratterizzare Π a partire dalle applicazioni e dal sistema operativo.

CTL, LTL

Ammettendo di poter associare i permessi di Android in un insieme di proposizioni atomiche AP , la costruzione di un eventuale albero (per CTL) o sequenza (per LTL) per rappresentare la computazione del modello non sarebbe comunque banale. Per esempio, un tentativo in questa direzione è stato presentato in [21]. Tuttavia tale proposta richiede una sostanziale ridefinizione del framework applicativo e di sicurezza di Android.

Capitolo 3

Estensione del Modello di Sicurezza di Android

In questo capitolo verrà presentata l'estensione del modello di sicurezza di Android presa in considerazione [22]. Nella prima Sezione, 3.1, si descriveranno le metodologie utilizzate per modellare le componenti e i contesti esecutivi di Android. Nella seconda, 3.2, si tratterà del linguaggio delle politiche utilizzato e delle tecniche di verifica delle formule rispetto alla configurazione run-time del sistema. In Sezione 3.3, verrà spiegato il funzionamento dinamico dello stack delle componenti mentre, in Sezione 3.4, si mostrerà come le politiche di sicurezza possano essere verificate per mezzo di un SAT-Solver.

3.1 Modellazione delle componenti e delle configurazioni

L'application framework di Android utilizza uno *stack delle activity* per tenere traccia della gerarchia d'invocazione delle interfacce utenti. All'interno dello stack, la componente sulla cima costituisce sempre l'elemento visibile all'utente. Questo concetto viene esteso dal modello di sicurezza preso in considerazione [22], dove viene adattato per rappresentare la configurazione run-time dell'intero sistema.

Una configurazione run-time del sistema Android, Σ , viene rappresentata per mezzo di un insieme di stack delle componenti S , contenenti zero o più frame F .

Definizione 3.1. Un *frame* è una tripla $F = \langle C, P, \Phi \rangle$ ove C è una componente, P un insieme di permessi e Φ un insieme di politiche. Si utilizzerà C_F , P_F e Φ_F per indicare gli elementi del frame $F = \langle C_F, P_F, \Phi_F \rangle$.

Definizione 3.2. Uno *stack delle componenti* S può essere sia vuoto, (ε) , sia la composizione di un frame F con uno stack S' , ossia $F :: S'$. Si utilizzerà $F \in S$ per indicare che il frame F appare, in una qualche locazione, nello stack S e S^i per indicare l' i -esimo frame $F_i \in S$.

Definizione 3.3. Una *configurazione* $\Sigma = [S_1; S_2; \dots]$ è una sequenza finita di stack di componenti S . Si indicherà con \emptyset una configurazione vuota, con $S \in \Sigma$ uno stack in una qualunque posizione di Σ , e con Σ^i l' i -esimo stack $S_i \in \Sigma$.

3.2 Definizione e validazione delle politiche

Il linguaggio delle politiche fornisce gli strumenti per poter discriminare a run-time tra comportamenti legali e non delle configurazioni del sistema.

Definizione 3.4. Una politica ϕ, ϕ' è una formula generata dalla seguente sintassi.

$$\begin{aligned}\pi, \pi' ::= & \top \mid p_i \mid \neg\pi \mid \pi \wedge \pi' \mid \pi \vee \pi' \mid \pi \rightarrow \pi' \\ \phi, \phi' ::= & \nabla\pi \mid \Diamond\pi \mid \Box\pi\end{aligned}$$

dove p_i sono nomi di permessi appartenenti all'insieme finito $\mathcal{P} = \{p_1, \dots, p_N\}$.

In poche parole, una politica di sicurezza consiste di un operatore di *scope* seguito da una formula proposizionale. Lo *scope* di una politica può essere *diretto*, ossia ∇ , *locale*, ossia \Diamond , o *globale*, ossia \Box . Esso ha effetto sui permessi che vengono considerati quando si valuta la politica: una politica diretta viene validata rispetto i permessi di un singolo frame, una locale rispetto l'insieme dei permessi di uno specifico stack e una globale rispetto l'insieme dei permessi di tutti gli stack della configurazione presi singolarmente. Inoltre, una politica può anche essere classificata come *sticky*. Essa viene verificata come le politiche normali, ma si *propaga* su tutti i frame dello stack. Le politiche sticky dirette, locali e globali vengono rispettivamente indicate con i simboli $\blacktriangledown\pi$, $\blacklozenge\pi$ e $\blacksquare\pi$.

La validazione di una configurazione richiede di confrontare l'insieme dei suoi permessi rispetto alla sua politica corrente. La politica della configurazione si ottiene attraverso la composizione di tutte le politiche che appaiono negli stack di componenti, mentre l'insieme dei permessi viene costituito opportunamente a partire da quelli dai singoli frame soggetti alla politica.

Definizione 3.5. Un insieme di permessi P soddisfa una formula proposizionale π , in simboli, $P \models \pi$, se e solo se:

$$\begin{aligned}P \models \top & \\ P \models p & \iff p \in P \\ P \models \neg\pi & \iff P \not\models \pi \\ P \models \pi \wedge \pi' & \iff P \models \pi \text{ e } P \models \pi' \\ P \models \pi \vee \pi' & \iff P \models \pi \text{ o } P \models \pi' \\ P \models \pi \rightarrow \pi' & \iff P \models \pi \text{ implica } P \models \pi'\end{aligned}$$

Si indica con $F \models \pi$ ogni volta i permessi del frame F soddisfano la politica π , ossia $P_F \models \pi$ e, con $S \models \pi$, ogni volta l'unione dei permessi di tutti i frame appartenenti allo stack S soddisfano la politica π , ossia $\bigcup_{F \in S} P_F \models \pi$.

In generale, la valutazione di una politica che appare in una configurazione dipende dalla sua posizione in termini di stack e frame. A tale scopo introduciamo una coppia di indici i, j per denotare la posizione della politica ϕ all'interno di una configurazione Σ . In simboli, scriviamo $\Sigma_{i,j} \models \phi$ per indicare che la configurazione Σ soddisfa la politica di sicurezza ϕ che appare nel j -esimo frame sull' i -esimo stack di Σ , ossia in $(\Sigma^i)^j$. Data una configurazione Σ e una politica ϕ localizzata in $(\Sigma^i)^j$, la validità di Σ rispetto ϕ è definita tramite le seguenti regole.

$$\Sigma_{i,j} \models \nabla\phi \iff (\Sigma^i)^{j-1} \models \phi \quad (3.1)$$

$$\Sigma_{i,j} \models \Diamond\pi \iff \Sigma^i \models \pi \quad (3.2)$$

$$\Sigma_{i,j} \models \Box\pi \iff \forall k, \Sigma^k \models \pi \quad (3.3)$$

Generalizzando, diciamo che una configurazione Σ è *valida*, in simboli $\vdash \Sigma$, sse:

$$\forall_{i,j}, \Sigma \models \Phi_F, \text{ con } F = (\Sigma^i)^j \quad (3.4)$$

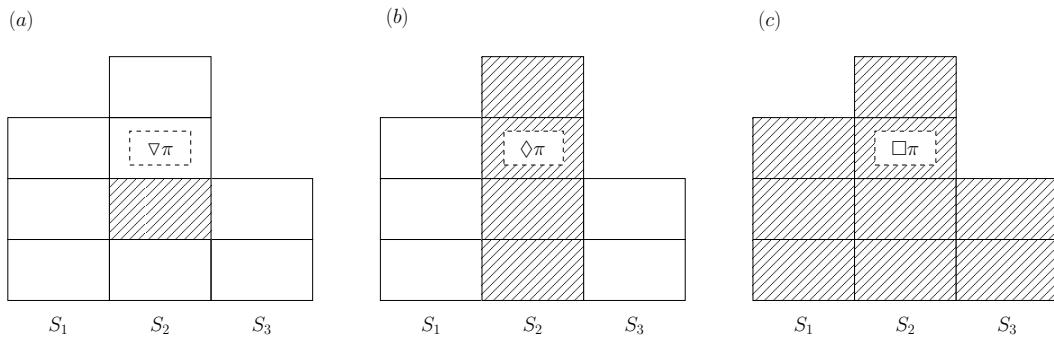


FIGURA 3.1: Una rappresentazione del contesto di valutazione degli scope delle politiche.

Esempio 3.1. Fig. 3.1 rappresenta graficamente il contesto di valutazione delle politiche dirette, locali e globali all'interno di tre configurazioni costituite, ciascuna, da tre stack. Nella configurazione (a), S_2 possiede un frame avente una politica diretta $\nabla\pi$, che deve essere validata rispetto ai permessi del frame sottostante. Lo stack S_2 della configurazione (b) possiede invece un frame con una politica locale $\Diamond\pi$, che deve essere verificata rispetto all'insieme dei permessi del suo stack di appartenenza. Infine, nella configurazione (c), lo stack S_2 presenta un frame con una politica globale $\Box\pi$, che richiede una validazione rispetto all'insieme dei permessi dei singoli stack appartenenti alla configurazione. ■

Esempio 3.2. Consideriamo il working example definito in § 2 e i requisiti di sicurezza definiti in Sezione 2.4, dove si è mostrato come essi siano modellabili in RBAC, CTL e LTL.

Mostreremo ora come sia ugualmente possibile modellare tali specifiche di sicurezza con il linguaggio descritto in Sezione 3.2.

Pagamenti verso contatti.

MaplePay permette alle applicazioni esterne di inviare denaro a contatti, mediante l'azione `send`, soltanto in presenza di un'autorizzazione esplicita, fornita o dallo stesso utente o dal servizio MaplePay.com. Questa regola è implementabile introducendo i permessi `USER AUTHORIZED (UAP)` e `APPLICATION PAYMENT (APP)`, e definendo la politica: $\nabla \neg APP \rightarrow UAP$.

Inoltre, come ulteriore controllo sull'applicazione chiamante, possiamo stabilire che solamente stack di invocazione aventi i permessi `READ CONTACTS (RCP)` e `GET ACCOUNTS (GAP)`, rispettivamente, accesso alla lista dei contatti e alle informazioni utente, possano inviare richieste a `ContactReceiver`. Pertanto, definiamo la seguente seconda politica di sicurezza: $\Diamond RCP \wedge GAP$.

Pagamenti verso e-commerce.

L'applicazione, mediante l'azione `pay`, fornisce due differenti modalità di pagamento verso e-commerce, per pagamenti normali e micro. Introducendo i due permessi `NORMAL PAYMENT (NPP)` e `MICRO PAYMENT (MPP)`, che le applicazioni chiamanti dovranno possedere per poter interagire con MaplePay, e utilizzando i permessi di autenticazione `UAP` e `APP` definiti precedentemente, è possibile definire le due politiche che coinvolgeranno `NormalPaymentReceiver` e `MicroPaymentReceiver` come: $\nabla NPP \wedge UAP$ e $\nabla MPP \wedge (UAP \vee APP)$.

Eavesdropping mitigation.

I sensori del dispositivo mobile possono essere utilizzati in maniera malevola per estrapolare informazioni riservate durante l'inserimento dei dati di autenticazione. È possibile evitare tali comportamenti vietando, all'interno della configurazione, l'esistenza di componenti che possano accedere a queste periferiche hardware durante l'interazione dell'utente con la `LoginActivity`. Pertanto, coinvolgendo i permessi `RECORD AUDIO (MIC)` e `CAMERA (CAM)`, possiamo definire la seguente politica: $\Box \neg (CAM \vee MIC)$.

Controllo dei flussi di dati.

Alcune informazioni sensibili potrebbero essere involontariamente condivise durante l'apertura degli estratti conto mediante applicazioni esterne di visualizzazione di documenti. Per prevenire questa fuga d'informazioni, il visualizzatore di documenti invocato dovrà essere impossibilitato ad accedere agli eventuali canali d'uscita disponibili, come, ad esempio, la memoria SD condivisa, le connessioni a internet o il trasmettitore bluetooth. Tale regola è esprimibile coinvolgendo, rispettivamente, i permessi: `WRITE_SD (WSD)`, `INTERNET (NET)` e `BLUETOOTH (BTT)`. Inoltre, dichiarando la politica come *sticky*, è possibile

prevenire possibili flussi di uscita di dati indiretti.

Infine, per permettere l'interoperabilità interna delle componenti che utilizzano i permessi soggetti a restrizioni appena considerati, definiamo il permesso **AUTHORIZED COMPONENT (ACP)** che autorizza tali componenti ad operare liberamente. La formula rappresentata la politica di sicurezza descritta risulta quindi: $\blacklozenge(\neg ACP \rightarrow \neg(NET \vee WSD \vee BTT))$.

Fig. 3.2 mostra l'architettura di MaplePay arricchita con le annotazioni di sicurezza appena descritte. Vengono utilizzati i simboli \emptyset e \top per rappresentare, rispettivamente, gli insieme vuoti dei permessi e delle politiche.

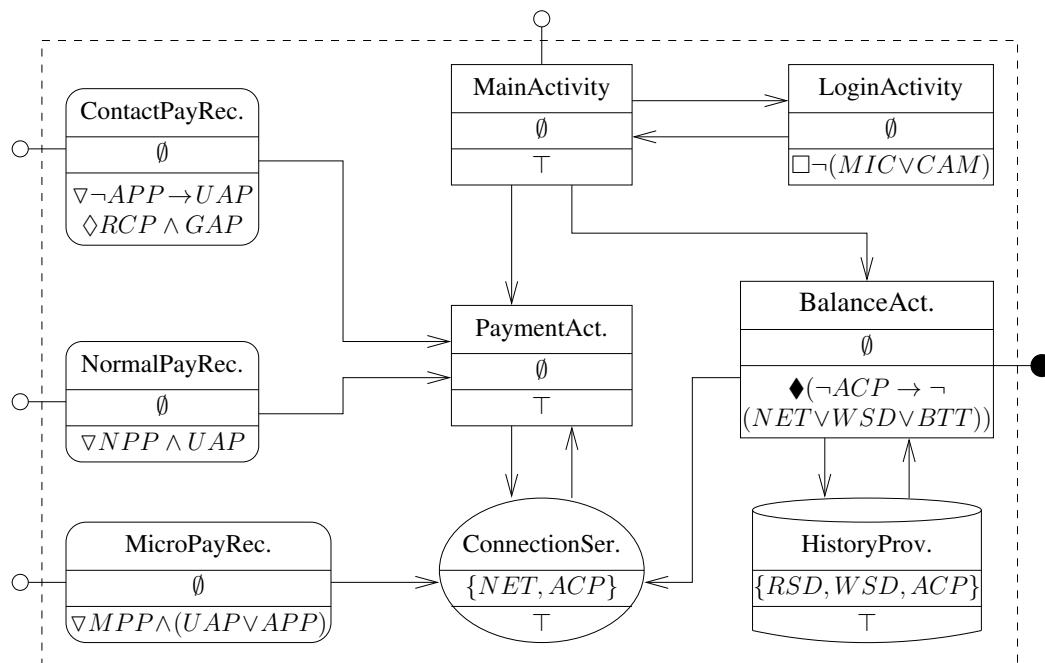


FIGURA 3.2: Struttura di MaplePay comprensiva di permessi e politiche di sicurezza.

3.3 Semantica operazionale

Gli stack di una configurazione consistono delle componenti coinvolte nella computazione in corso. Quando una componente viene invocata, il corrispondente frame viene aggiunto sulla cima dello stack che ha generato l'invocazione. Analogamente, nuovi stack possono essere allocati quando (i) viene lanciata l'esecuzione di una nuova applicazione o (ii) quando viene lanciato un servizio.

L'operazione inversa consiste nel rimuovere un frame dalla configurazione quando la corrispondente componente viene deallocata dal sistema operativo. Queste due operazioni, che chiamiamo di *push* e di *pop*, avvengono secondo le seguenti specifiche regole.

Operazione di push del frame $\langle C, P, \Phi \rangle$ invocata da uno stack $S_i \in \Sigma$.¹

- Precondizioni:
 1. C non è un service;
 2. Φ' è pari all'insieme di politiche sticky appartenenti a Φ ;
 3. Φ'' è pari all'insieme di politiche sticky contenute in S_i ;
 4. Σ' è pari a Σ dove S_i è sostituito da $\langle C, P, \Phi \cup \Phi'' \rangle :: (S_i)^{\Phi'}$ (dove $(S_i)^{\Phi'}$ è ottenuto sostituendo $\Phi_j \cup \Phi'$ in tutti i frame $F_j = \langle C_j, P_j, \Phi_j \rangle$);
 5. $\vdash \Sigma'$;
- La configurazione risultante è Σ' .

Operazione di push del frame $\langle C, P, \Phi \rangle$ invocata da uno stack $S_i \in \Sigma$.

- Precondizioni:
 1. C è un service;
 2. Φ' è pari all'insieme di politiche sticky contenute in Φ ;
 3. Φ'' è pari all'insieme di politiche sticky contenute in S_i ;
 4. Σ' è pari a $[\Sigma; \overline{S}]$, con (i) $\Sigma'' = \Sigma$ dove S_i è sostituito da $(S_i)^{\Phi'}$ (pari a S_i ma sostituendo $\Phi_j \cup \Phi'$ in tutti i frame $F_j = \langle C_j, P_j, \Phi_j \rangle$) e (ii) $\overline{S} = \langle C, P, \Phi \cup \Phi'' \rangle :: \varepsilon$;
 5. $\vdash \Sigma'$;
- La configurazione risultante è Σ' .

Operazione di pop invocata da uno stack $S_i \in \Sigma$ con $S_i = F :: S$.

- Precondizioni:
 1. Σ' è pari a Σ con lo stack S al posto di S_i ;
 2. $\vdash \Sigma'$;
- La configurazione risultante è Σ' .

Nel seguito, si utilizzerà $\Sigma \xrightarrow{push(F,S)} \Sigma'$ per indicare il successo di un'operazione di push del frame F sullo stack S e, con $\Sigma \not\xrightarrow{push(F,S)}$, per indicarne un suo fallimento. Analogamente, indicheremo con $\Sigma \xrightarrow{pop(S)} \Sigma'$ il successo di un'operazione di pop eseguita sullo stack S e, con $\Sigma \not\xrightarrow{pop(S)}$, un suo eventuale fallimento.

Esempio 3.3. Riprendiamo ancora il nostro caso di studio, arricchito nell'esempio 3.2 con permessi e politiche di sicurezza, e analizziamo l'evoluzione della configurazione di sistema durante una possibile interazione di MaplePay con l'utente. In particolare, descriviamo le operazioni che avvengono sugli stack nel caso in cui, a seguito dell'accensione del suo dispositivo mobile, l'utente avvia MaplePay come prima applicazione per visualizzare l'ultimo estratto conto del suo account registrato su MaplePay.com.

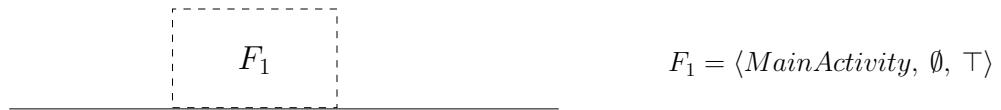
1. All'avvio di MaplePay da parte dell'utente, il sistema operativo invoca tramite un intent `start` la componente `MainActivity`, che, come mostrato in

¹Si noti che, considerando $S_i = \varepsilon$, la regola è valida anche nel caso di avvio della componente dal sistema operativo.

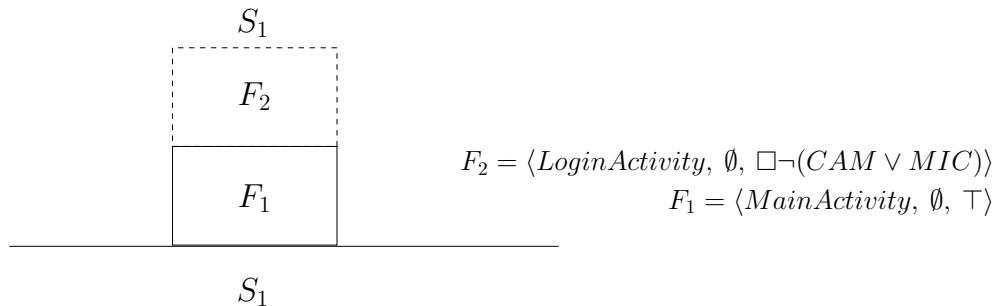
Fig. 3.2, non possiede né politiche né permessi. Viene quindi creato il frame $F_1 = \langle \text{MainActivity}, \emptyset, \top \rangle$ e richiesto il suo inserimento all'interno della configurazione del sistema Σ . Ipotizziamo che questa risulti ancora vuota; banalmente, si ha che la nuova configurazione Σ' costituita dal solo stack $F_1 :: S_1$ è valida, ossia $\Sigma \xrightarrow{\text{push}(F_1, S_1)} \Sigma'$.

2. *MaplePay* automaticamente invoca la *LoginActivity* per eseguire l'autenticazione dell'utente. Viene pertanto creato il frame $F_2 = \langle \text{LoginActivity}, \emptyset, \square \neg(\text{CAM} \vee \text{MIC}) \rangle$ e richiesto il suo inserimento sullo stack S_1 . Data la sola presenza di S_1 in Σ e l'assenza di permessi e di altre politiche oltre a quella indicata da F_2 , si ha anche in questo caso la validità della nuova configurazione Σ' , pertanto $\Sigma \xrightarrow{\text{push}(F_2, S_1)} \Sigma'$.
3. Terminata la procedura di autenticazione all'interno della *LoginActivity*, la componente ritorna un token di autenticazione alla *MainActivity* e termina. Viene pertanto verificata l'eseguibilità dell'operazione $\Sigma \xrightarrow{\text{pop}(S_1)} \Sigma'$, che, anche in questo caso ha successo.
4. A questo punto l'utente desidera visualizzare i propri estratti conti. Viene richiesta l'invocazione della *BalanceActivity*. Analogamente ai casi precedenti, si avrà $F_3 = \langle \text{BalanceActivity}, \emptyset, \blacklozenge(\neg \text{ACP} \rightarrow \neg(\text{NET} \vee \text{WSD} \vee \text{BTT})) \rangle$ e $\Sigma \xrightarrow{\text{push}(F_3, S_1)} \Sigma'$. Oltre a ciò, essendo Φ_{F_3} una politica *sticky*, si ha che F_1 la eredita, diventando $F_1 = \langle \text{MainActivity}, \emptyset, \blacklozenge(\neg \text{ACP} \rightarrow \neg(\text{NET} \vee \text{WSD} \vee \text{BTT})) \rangle$.
5. Automaticamente viene chiamato in background il *ConnectionService* per il download, dal server di *MaplePay.com*, dei dati aggiornati dell'account dell'utente. Si ha pertanto $F_4 = \langle \text{ConnectService}, \{\text{NET}, \text{ACP}\}, \top \rangle$. A differenza dei casi precedenti, però, la nuova componente possiede dei permessi: si deve quindi verificare che $\{\text{NET}, \text{ACP}\} \models \blacklozenge(\neg \text{ACP} \rightarrow \neg(\text{NET} \vee \text{WSD} \vee \text{BTT}))$. Anche in questo caso la nuova configurazione Σ' risulta valida, quindi viene eseguita $\Sigma \xrightarrow{\text{push}(F_4, S_2)} \Sigma'$, che comporta, essendo C_{F_4} un service, la creazione di un secondo stack S_2 all'interno di Σ .
6. *BalanceActivity* lista le informazioni aggiornate ottenute dal service e richiede ad *HistoryProvider* di fornirgli quelle passate già memorizzate sul dispositivo. Viene quindi preparato $F_5 = \langle \text{HistoryProvider}, \emptyset, \{\text{RSD}, \text{WSD}, \text{ACP}\} \rangle$ e richiesta la sua invocazione. Anche in questo caso, è necessario verificare i permessi della componente con le politiche presenti sullo stack S_1 , ossia verificare $\{\text{RSD}, \text{WSD}, \text{ACP}\} \models \blacklozenge(\neg \text{ACP} \rightarrow \neg(\text{NET} \vee \text{WSD} \vee \text{BTT}))$. Grazie al permesso *ACP*, anche questa invocazione può avere successo, ossia $\Sigma \xrightarrow{\text{push}(F_5, S_1)} \Sigma'$.
7. *HistoryProvider* carica quindi i passati estratti conti e, grazie a suoi permessi, li scrive sulla memoria SD del dispositivo, in modo da poterli condividere con

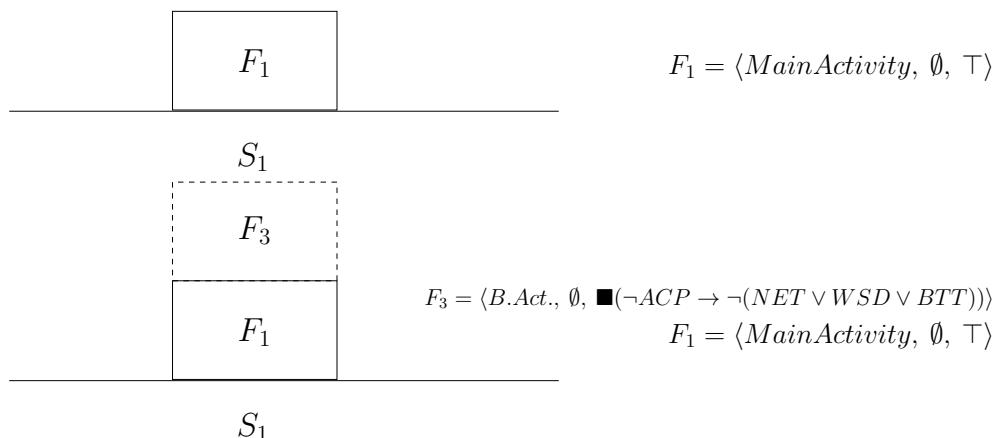
1



2



3

FIGURA 3.3: Evoluzione della configurazione Σ dell’Esempio 3.3, passi 1-4.

eventuali applicazioni di visualizzazione di documenti. Fatto ciò ritorna a *BalanceActivity* un acknowledgement per comunicargli l’avvenuta copia dei dati. Viene quindi richiesta una $\Sigma \xrightarrow{\text{pop}(S_1)} \Sigma'$, che, essendo valida Σ' , rimuoverà F_5 dallo stack S_1 .

8. A questo punto *BalanceActivity* può mostrare tutte le informazioni sull’account disponibili. L’utente, interessato a visualizzare i suoi ultimi movimenti, preme l’apposito pulsante per avviare la procedura di visualizzazione e l’activity prepara l’apposito intent **start**, come definito nella tabella 2.1.

Sul dispositivo sono installate due applicazioni di visualizzazione dei documenti: *DocView1* e *DocView2*. Entrambe forniscono un’activity per la visualizzazione dei file di testo, rispettivamente *ViewAct.1* e *ViewAct.2*, in grado di gestire l’intent inviato da *BalanceActivity*, ma possiedono permessi diversi: *RCP*, *GAP* e *MIC* per *ViewAct.1*, *NET* e *RCP* per *ViewAct.2*. Vengono pertanto creati i due frame $F_6 = \langle \text{ViewAct.1}, \{RCP, GAP, MIC\}, \top \rangle$ e

$F_7 = \langle \text{ViewAct.2}, \{\text{NET}, \text{RCP}\}, \top \rangle$, per entrambi i quali si dovrà valutare la soddisfabilità rispetto alle politiche dello stack, ossia, rispettivamente: $\{\text{RCP}, \text{GAP}, \text{MIC}\} \models \blacklozenge(\neg \text{ACP} \rightarrow \neg(\text{NET} \vee \text{WSD} \vee \text{BTT}))$ e $\{\text{NET}, \text{RCP}\} \models \blacklozenge(\neg \text{ACP} \rightarrow \neg(\text{NET} \vee \text{WSD} \vee \text{BTT}))$.

Data la presenza del permesso NET all'interno di F_7 , si avrà $\Sigma \xrightarrow{\text{push}(F_6, S_1)} \Sigma'$ e $\Sigma \xrightarrow{\text{push}(F_7, S)} \Sigma'$ in quanto solamente Σ' risulta valida.

Le Fig. 3.3 e 3.4 mostrano l'evoluzione della configurazione Σ durante nell'interazione spiegata nei precedenti punti. \blacksquare

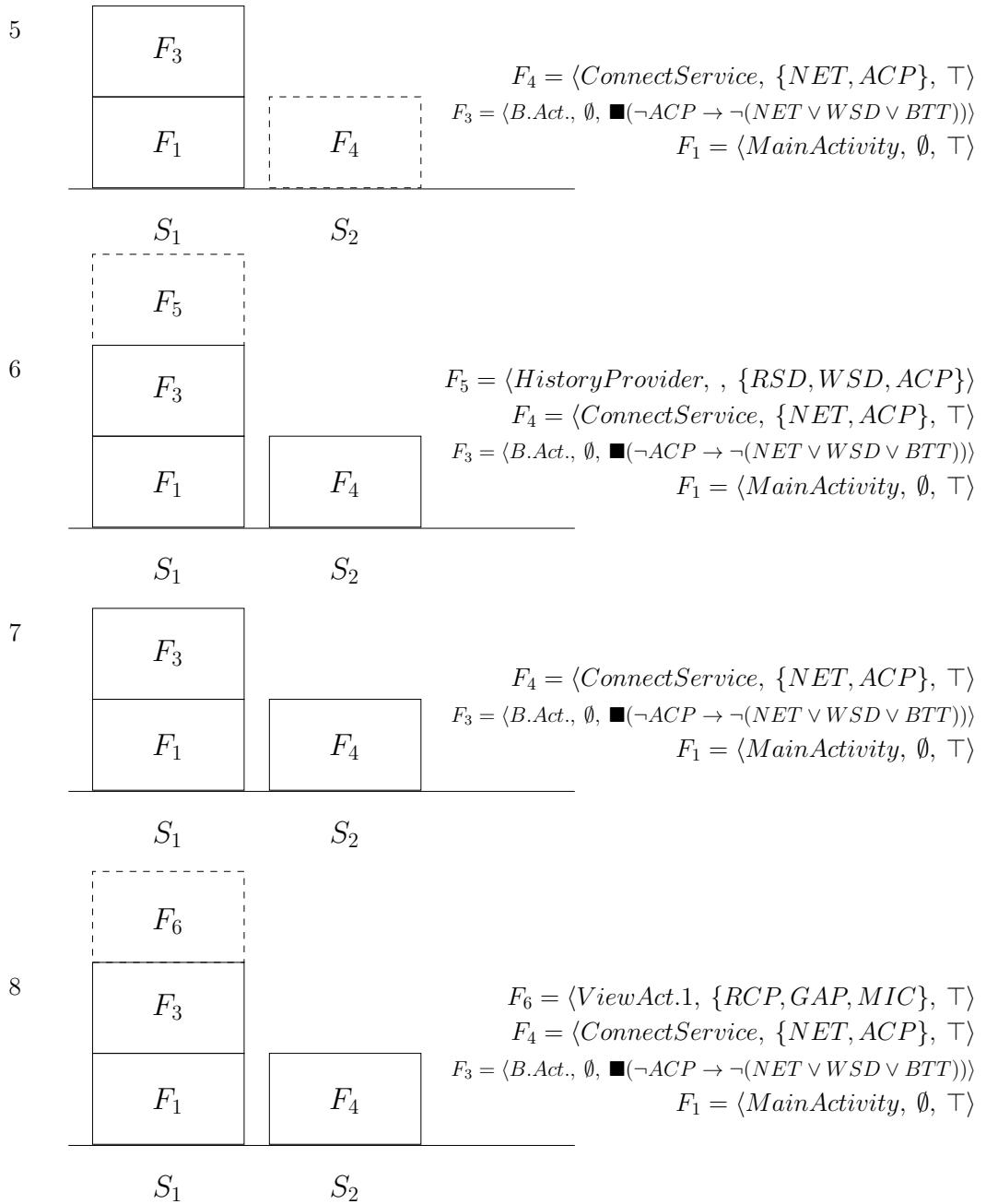


FIGURA 3.4: Evoluzione della configurazione Σ dell'Esempio 3.3, passi 5-8.

Considerazioni

Le tre condizioni di validità delle tipologie di politica riportate in Sezione 3.2, permettono alle componenti di imporre restrizioni sul contesto (cioè la configurazione) nel quale vengono eseguite. Infatti, tramite gli operatori di scope, una componente può discriminare sulle caratteristiche della componente chiamante (∇), dello stack d'esecuzione (\diamond) o dell'intera configurazione (\square). Inoltre, le politiche sticky interessano i cambiamenti della configurazione nel *tempo*, dato che persistono anche dopo la rimozione della componente che le ha definite.

Una prima proprietà d'interesse, per caratterizzare il potere espressivo del linguaggio di politiche, riguarda la possibilità di estendere effettivamente l'insieme delle politiche definibili dal security framework di Android. Tale proprietà è dimostrata in [22] e di seguito riportata.

Proprietà 1. *Il linguaggio di specifica delle politiche, introdotto in Sezione 3.2, include strettamente l'insieme delle politiche definibili in Android.*

Anche se non è stato condotto uno studio rigoroso della tassonomia del linguaggio di specifica delle politiche utilizzato, possiamo delineare una caratterizzazione informale. Chiaramente, tutte le politiche definibili sono un sottoinsieme delle politiche *Safety*, in quanto, nel nostro modello, vengono considerate solamente tracce *finite* di esecuzione.

Inoltre, le tre modalità $\nabla \diamond$ e \square , delineano insiemi di politiche indipendenti aventi intersezione non vuota. Per esempio, alcune politiche banali come sempre verificato o sempre violato sono definibili in tutte e tre le modalità.

Queste considerazioni sono riassumibili dal diagramma in Fig.3.5.

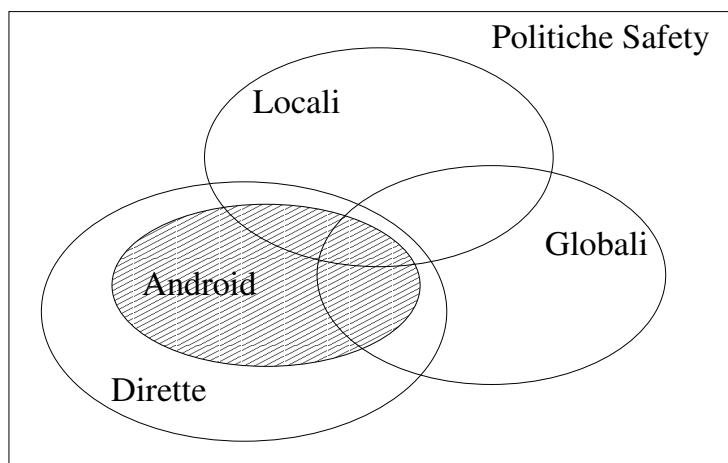


FIGURA 3.5: Una rappresentazione della tassonomia del linguaggio di specifica delle politiche utilizzato dal modello.

3.4 Validazione come problema di soddisfacibilità

Il problema di stabilire se una data configurazione Σ sia valida, ossia $\vdash \Sigma$, può essere ridotto a un problema di soddisfacibilità booleano (SAT). Ad ogni richiesta di un'operazione di *push* o di *pop*, il modello genera un'istanza di problema SAT, che ammetterà assegnamenti validi se e solo se una data configurazione potrà evolvere senza violare le politiche di sicurezza in essa definite.

Quindi, le tre condizioni di validità della configurazione presentate in sezione 3.2, assumono, rispettivamente, le seguenti forme in clausole booleane:

$$\bigwedge_{F_i \in S} \left(\bigwedge_{p \in P_{i-1}} p \rightarrow \bigwedge_{\forall \pi \in \Phi_i} \pi \right) \quad (3.5)$$

$$\bigwedge_{F_i \in S} \bigwedge_{p \in P_i} p \rightarrow \bigwedge_{F_j \in S} \bigwedge_{\Diamond \pi \in \Phi_j} \pi \quad (3.6)$$

$$\bigwedge_{S \in \Sigma} \left(\bigwedge_{F_i \in S} \bigwedge_{p \in P_i} p \rightarrow \bigwedge_{S' \in \Sigma} \bigwedge_{F_j \in S'} \bigwedge_{\Box \pi \in \Phi_j} \pi \right) \quad (3.7)$$

Dati Σ e $S \in \Sigma$, scriviamo che $\text{SAT}(S, \Sigma)$ se le formule 3.5, 3.6 e 3.7 sono soddisfacibili.

Il seguente teorema, riportato da [22], fornisce un fondamento teorico/matematico per l'applicazione dei SAT-Solver al problema di individuazione di componenti legali nei sistemi Android.

Teorema 3.6. *Per ogni F , S e Σ' tali che $\text{SAT}(F :: S, \Sigma')$, non accade mai che $\Sigma \xrightarrow{\text{push}(F, S)}$, o che $\Sigma \xrightarrow{\text{pop}(S)}$*

Informalmente, è sempre possibile eseguire una push o una pop di un frame F su o da uno stack S se la configurazione risultante genera un assegnamento soddisfacibile.

Capitolo 4

Secure Component Provider

In questo capitolo verranno descritti gli strumenti progettati ed implementati per fornire agli sviluppatori le funzionalità del modello logico presentato all'interno del § 3. In particolare, verrà presentato il *Secure Component Provider*, un prototipo costituito dall'applicativo *SCPApp* e dalle librerie di sviluppo *SCPLib*.

Esso è stato sviluppato sulla base dei seguenti obiettivi.

1. Abilitare i dispositivi alla verifica delle politiche di sicurezza e al monitoraggio delle applicazioni senza personalizzare il sistema operativo Android.
2. Permettere alle applicazioni di interagire correttamente con i meccanismi richiesti dall'obiettivo 1 senza intaccare l'esperienza utente.

Nel seguito del capitolo verrà presentata, in Sezione 4.1, l'architettura e il funzionamento del prototipo sviluppato. In Sezione 4.2 verrà analizzata nel dettaglio la struttura e gli elementi che compongono l'applicazione SCPApp. In Sezione 4.3, verranno descritte le librerie SCPLib fornite agli sviluppatori e, infine, in Sezione 4.4, verranno analizzate alcune problematiche riscontrate e motivate le scelte implementative effettuate.

4.1 Architettura e funzionamento del prototipo

SCP è un prototipo costituito da due elementi: un'applicazione, la *SCP Application* (SCPApp), che si occupa della verifica delle politiche di sicurezza e della gestione dell'interazione delle componenti delle applicazioni, e le *SCP Library* (SCPLib), un pacchetto di librerie che forniscono un meccanismo di interazione con SCPApp alle applicazioni.

Per una corretta esecuzione del prototipo, sono richieste tre precondizioni.

1. SCPApp deve essere installata sul dispositivo utente.
2. Le applicazioni che intendono utilizzare il framework di sicurezza proposto devono essere sviluppate per mezzo delle SCPLib.
3. I file Manifest.xml di tali applicazioni devono specificare, per ogni componente, i permessi e le politiche di sicurezza definite dallo sviluppatore.

Quindi, se le tre precondizioni sono soddisfatte, il funzionamento di SCP è descrivibile, in sezioni, come segue.

Specifiche dei permessi e delle politiche

Come spiegato in Sezione 1.1.1, il file Manifest riassume le informazioni dell'applicazione, come i permessi che utilizza e il nome e la tipologia di ogni componente che la costituisce. Inoltre, le specifiche strutturali di tale file permettono di associare per ogni possibile componente, uno o più campi <meta-data>. Questi vengono utilizzati all'interno del prototipo per permettere allo sviluppatore di specificare sia i permessi sia le politiche di sicurezza delle singole componenti. I primi, vengono indicati per mezzo dei loro nomi comprensivi del package dell'applicazione che li definisce, proprio come nel tradizionale framework di sicurezza di Android. Le politiche, invece, devono essere strutturate in Conjunctive Normal Form (CNF).

Installazione delle applicazioni

Durante la procedura d'installazione di un'applicazione, SCPApp rileva automaticamente tutte le componenti che la costituiscono, e, per ognuna di esse, memorizza le eventuali informazioni di sicurezza presenti (permessi e politiche). Come verrà descritto nel dettaglio in Sezione 4.2.4, tali informazioni vengono memorizzate per mezzo di un database implementato all'interno di SCPApp. In particolare, ad ogni nuovo permesso rilevato viene associato un identificativo univoco, che permette una codifica consistente tra tutte le politiche di sicurezza definite dalle applicazioni installate sul dispositivo.

Invocazione delle componenti

Come specificato in Sezione 1.1.1, le invocazioni delle componenti possono provenire da qualunque applicazione, sia essa facente parte del sistema operativo che non. A seconda del chiamante, come spiegato in Sezione 3.3, la componente si troverà a far parte o di uno stack già esistente all'interno della configurazione o di uno nuovo. Distinguiamo pertanto i seguenti due casi:

- *Avvio della componente da applicazione esterna.* In questo caso la richiesta d'invocazione proviene da una componente facente già parte di uno stack della configurazione. La componente chiamante contatta SCPApp per richiedergli la validazione di un set di possibili componenti candidate all'invocazione.

Ogni componente viene quindi elaborata allo stesso modo, mediante i seguenti passi.

1. Per mezzo del nome univoco della classe che la implementa, vengono caricate dal database le informazioni di sicurezza (permessi e politiche) relative alla componente;
2. Mediante tali informazioni, e quelle sull'attuale configurazione di sistema, vengono create e valutate le istanze di problemi SAT.

3. Nel caso in cui fossero tutte soddisfacibili con i soli permessi riportati all'interno di ciascuna, la componente viene aggiunta a una lista di ritorno, altrimenti viene scartata.

Se al termine dell'elaborazione la lista è vuota, viene lanciata un'opportuna eccezione. Altrimenti, in presenza di una o più componenti invocabili, viene creato un frame temporaneo, che viene posto sullo stack di riferimento, e viene ritornata alla componente chiamante la lista di componenti SAT ottenuta.

A questo punto, nel caso in cui la lista contenga una solo componente, essa viene avviata per mezzo dei meccanismi standard di Android. Altrimenti, un messaggio di dialog permette all'utente di scegliere quale componente avviare. Al termine della sua invocazione, la nuova componente contatta SCPApp per comunicargli il successo dell'operazione. Quindi, SCPApp, aggiorna il frame temporaneo precedentemente creato con le informazioni corrette.

- *Avvio della componente da sistema operativo.* In questo caso è la componente stessa che contatta SCPApp, per metterlo al corrente che è stata richiesta una sua invocazione. Viene quindi eseguito un procedimento analogo al precedente per valutare la validità dell'operazione. Tuttavia, nel caso in cui tutte le istanze SAT siano soddisfacibili, viene creato un nuovo stack all'interno della configurazione, e, su di esso, eseguita l'operazione di push del frame, completo di ogni informazione di sicurezza associata alla componente invocata. Infine, viene ritornato un messaggi di conferma dell'avvenuta operazione, che permette alla componente di terminare la sua procedura di creazione.

Rimozione delle componenti

Quando Android stabilisce la distruzione di un'istanza di una componente, questa, prima di terminare, contatta SCPApp per comunicargli che sta per abbandonare il contesto di esecuzione del sistema. Tuttavia, l'operazione di pop da uno stack può avvenire solamente nel caso in cui il corrispondente frame della componente di interesse si trovi sulla cima dello stack di appartenenza. Ciò è dovuto al fatto che, a causa dei differenti cicli vitali delle componenti delle applicazioni, è possibile che il sistema operativo distrugga temporaneamente elementi non più utili all'interazione con l'utente, che potrebbero essere associati in frame posti all'interno, e non sulla cima, di uno stack. Tuttavia, tali frame devono essere conservati, in quanto necessari per mantenere consistenti i permessi e le politiche di sicurezza della configurazione di sistema.

Nel caso in cui, la richiesta di rimozione coinvolga effettivamente una componente posta sulla cima di uno stack, viene generata, come descritto in Sezione 3.4, e valutata un'istanza di problema SAT. Se la rimozione della componente risulta legale, viene eseguita l'operazione di pop sullo stack interessato. Altrimenti, il frame permane all'interno della configurazione.

Infine, in caso di disinstallazione di un'applicazione, SCPApp rimuove dal database tutte le entry relative alle componenti che la costituiscono.

4.2 Organizzazione di SCPApp

SCPApp è un'applicazione organizzata in cinque moduli, ognuno dei quali avente un preciso ruolo. La Fig. 4.1, mostra i flussi d'interazione tra i vari moduli e l'ambiente esterno.

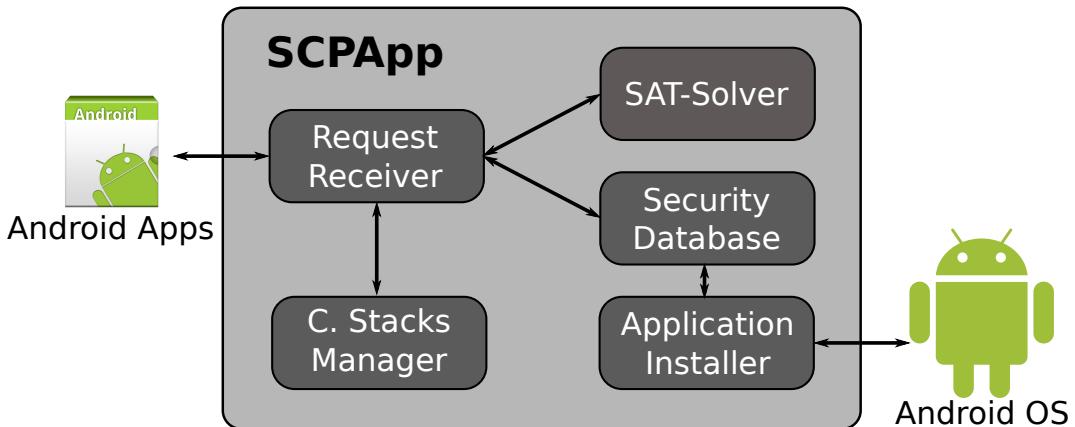


FIGURA 4.1: Struttura del prototipo SCP.

4.2.1 Request Receiver

Il Request Receiver è il modulo centrale del prototipo. Costituito da un content provider, gestisce l'interazione tra i moduli di SCPApp e interagisce con le applicazioni client occupandosi delle loro richieste. Queste vengono inoltrate al provider mediante il meccanismo degli URI e gestite mediante l'implementazione dei tre metodi *query()*, *insert()* e *delete()*.

Inoltre, mediante i metodi di *encode*, il request receiver si occupa anche di codificare le formule CNF da inviare al SAT-Solver per la verifica delle invocazioni delle componenti.

Metodo Query() Viene utilizzato per registrare le componenti ad SCP nel caso di invocazioni originate da altre componenti che utilizzano il protocollo di sicurezza proposto. Gli URI indirizzati a questo metodo possiedono il *path* `COMPONENT_REQUESTED` e contengono, all'interno del parametro *selection*, l'elenco delle possibili componenti invocabili, e, come *fragment* dell'URI, l'identificativo dello stack di appartenenza della componente che ha effettuato la richiesta.

Il funzionamento del metodo prevede, per prima cosa, di verificare l'esistenza del campo *fragment* e del relativo stack a cui fa riferimento. Successivamente, la

lista delle componenti ricevuta viene inoltrata, mediante il metodo nativo *ArrayList <Component> query(String componentName)*, direttamente al modulo *Security Database*, dove opportuni metodi interrogheranno il DB per ottenere le informazioni delle componenti candidate all'invocazione.

La risposta del metodo sarà costituita da una lista di componenti: se vuota, nessuna tra le componenti richieste utilizza il nostro framework di sicurezza, pertanto viene lanciata un'eccezione che simula il normale comportamento di Android quando non viene trovata una componente invocabile. Invece, nel caso in cui la lista contenesse uno o più elementi, ognuno di essi verrebbe elaborato mediante i metodi *encode*, che si occupano di comporre le formule CNF da verificare tramite il SAT-Solver.

Nel caso in cui una o più componenti soddisfino la configurazione del sistema, viene eseguita, mediante il modulo *Component Stacks Manager*, un'operazione di pop di un frame temporaneo. Questo contiene solamente un identificatore univoco temporaneo, utilizzato per tenere traccia della componente all'interno delle configurazioni fino alla sua rimozione. Fatto ciò, le componenti SAT vengono organizzate, assieme a tutte le informazioni necessarie alle loro eventuali invocazioni e future interazioni con SCP, all'interno di un'opportuna lista che verrà ritornata alla componente mittente.

Se invece nessuna delle componenti risultasse SAT, verrebbe lanciata la stessa eccezione del caso precedente.

Metodo Insert() Viene utilizzato sia per registrare in SCP la componente d'ingresso invocata all'avvio di un'applicazione, sia per aggiornare eventuali frame temporanei presenti sugli stack delle componenti. In entrambi i casi, il funzionamento del metodo prevede di verificare, all'interno del parametro *ContentValues*, la presenza del nome della classe e dell'id temporaneo della componente. Tali valori sono essenziali in quanto permettono, rispettivamente, il recupero delle informazioni della componente dal database e il controllo della legittimità della richiesta.

Fatto ciò, viene determinata la tipologia della chiamata, analizzando il *path* degli uri ricevuti. Nel caso di **COMPONENT_INVOKED**, la richiesta proviene da una componente che è già stata verificata da SCP e che ha appena terminato la sua fase d'invocazione. Pertanto, si accede, tramite il metodo *Component query(String componentName)*, al database per ottenere le informazioni della componente, e si contatta il Component Stacks Manager affinché le inserisca all'interno del frame temporaneo presente sullo stack indicato.

Invece, nel caso di **NEW_APPLICATION_STARTED**, la richiesta proviene da una componente d'ingresso di una applicazione appena avviata dal sistema operativo. Mediante un procedimento simile a quello utilizzato dal metodo *query()*, si verifica l'invocabilità della componente. In questo caso, però, non viene creato un frame

temporaneo, ma, eventualmente, si inserisce un frame completo su un nuovo stack, in quanto la componente è già stata parzialmente invocata dal sistema operativo.

Infine, per entrambi i casi, una notify avverte il mittente dell'avvenuta operazione, ritornando o l'id temporaneo della componente, per confermare il successo dell'operazione, o zero in caso di errore.

Metodo Delete() Viene utilizzato per rimuovere le componenti dalla configurazione presente all'interno del Component Stacks Manager. Gli URI indirizzati a questo metodo possiedono il *path* `COMPONENT_REMOVED` e contengono, all'interno del parametro *selectionArgs*, l'id temporaneo della componente, il suo stack di appartenenza e il nome della sua classe.

Il funzionamento prevede, per prima cosa, di recuperare dal database le informazioni della componente specificata e di verificare, mediante gli identificatori ricevuti, l'esistenza del frame e dello stack. Nel caso in cui il frame indicato si trovi effettivamente sulla cima dello stack specificato, viene richiesto ai metodi *encode* di comporre le opportune CNF per verificare, mediante SAT-Solver, la legittimità dell'operazione. Nel caso in cui le istanze dei problemi SAT risultino soddisfacibili, viene richiesto al Components Stack Manager di eseguire l'operazione di pop del frame dallo stack indicato, altrimenti la richiesta viene ignorata.

Metodi Encode Sono tre metodi che vengono utilizzati per creare le formule da inviare al SAT-Solver per la verifica di invocabilità delle componenti. Le CNF vengono composte a seconda della tipologia delle politiche presenti nella componente.

- *String encodeDirectCnf(int stackId, Component component)*, ritorna una formula CNF costituita dall'*and logico* (\wedge) della politica diretta presente in *component* e dai permessi del frame sulla cima dello stack individuato da *stackId*.
- *String encodeLocalCnf(int stackId, Component component)*, ritorna una formula CNF costituita dal \wedge di tutte le politiche locali e permessi di *component* e di tutti i frame presenti sullo stack individuato da *stackId*.
- *String[] encodeGlobalCnf(int stackId, Component component)*, ritorna una formula CNF per ogni stack presente nella configurazione, costituita dal \wedge di tutte le politiche globali presenti nella configurazione, comprese le eventuali di *component*, e di tutti i permessi di tutti i frame presenti sullo stack considerato. Nel caso in cui lo stack in considerazione coincidesse con quello individuato dal parametro *stackId*, vengono considerati anche i permessi di *component*.

4.2.2 Component Stacks Manager

Il Component Stacks Manager, CSM, è il modulo che si occupa di mantenere un'immagine aggiornata della configurazione run-time del sistema all'interno del prototipo. Ciò avviene fornendo al Request Receiver delle opportune funzioni che permettono di mantenere una traccia del flusso d'invocazione delle componenti.

Inoltre, il CSM è in grado di generare, ogni volta che gli viene richiesto, le CNF riguardanti la configurazione del sistema, necessarie per comporre le opportune formule da inviare al SAT-Solver per la verifica delle politiche di sicurezza.

Gli elementi principali di questo modulo sono le due classi *ScpStack* e *ScpStacks-Set*.

ScpStack La classe si occupa di implementare gli stack delle componenti, descritti in Sezione 3.1, mediante l'utilizzo dei seguenti campi e metodi.

- *int stackId*, identifica univocamente lo stack.
- *Stack <Integer> stack*, utilizzato per tenere una traccia ordinata degli identificatori dei frame appartenenti allo stack.
- *HashMap <Integer, String > permissionMap*, per tenere traccia dei permessi posseduti dai frame dello stack.
- *HashMap <Integer, ArrayList <ScpPolicy >> directPoliciesMap* e *HashMap <Integer, ArrayList <ScpPolicy >> localPoliciesMap* per, rispettivamente, le politiche dirette e locali che ogni frame può avere.
- *void fillMaps(Component component, int frameId)*, completa i frame temporanei inseriti sullo stack. Il suo funzionamento prevede di recuperare un riferimento alla lista di politiche contenuta all'interno dell'oggetto *component*, e di analizzare, una alla volta, le politiche in essa inserite. A seconda del caso, viene aggiunta la politica nell'opportuna mappa dello stack, utilizzando il parametro *frameId* per mantenere un riferimento al frame di appartenenza. Nel caso in cui una o più politiche fossero di tipo *sticky*, esse vengono assegnate a tutti gli altri frame presenti sullo stack. Infine, viene aggiornata *permissionMap* inserendo i permessi posseduti dal nuovo frame.
- *void push(Component component, int frameId)*, esegue l'operazione di push del frame sullo stack. Brevemente, il metodo prevede di eseguire l'inserimento sullo stack, per poi verificare se si tratti di un frame temporaneo o meno. Nel caso positivo, il metodo ritorna immediatamente, altrimenti viene chiamato il metodo *fillMaps(component, frameId)* per inserire le opportune informazioni della componente nelle mappe dello stack.
- *int pop(int frameId)*, esegue l'operazione di pop del frame dallo stack. Per prima cosa, come spiegato in 4.1, viene controllato che la componente sia effettivamente sulla cima dello stack. Nel caso positivo, l'operazione viene eseguita e, per mezzo dell'identificatore del frame, si vanno a rimuovere le eventuali politiche, locali e dirette, e permessi dalle mappe dello stack. Infine, viene ritornato o zero, in caso di errore, o l'id della componente rimossa dallo stack, nel caso di operazione eseguita correttamente.

- *String getPermissionsAsString()*, per ritornare una stringa contenente l'AND di tutti i permessi dello stack.
- *String getDirectPermissionAsString()*, per ritornare una stringa contenente l'AND di tutti i permessi del frame sulla cima dello stack.
- *String getLocalPoliciesAsString()*, per ritornare una stringa contenente l'AND di tutte le politiche locali dello stack.

ScpStacksSet La classe si occupa di gestire l'insieme di ScpStack che rappresentano la configurazione run-time del sistema Android, avvalendosi dei seguenti campi e metodi.

- *HashMap <Integer, ScpStack> stacksSet*, utilizzata per tenere traccia degli stack di componenti rappresentanti la configurazione di sistema.
- *HashMap <String, ScpBox> globalPoliciesMap*, utilizzata per tenere traccia delle politiche globali presenti nei frame appartenenti alla configurazione.
- *int pushComponent(int dadStackId, int frameId, Component component)*, esegue l'operazione di push del frame sullo stack indicato dal parametro *dadStackId*. Per prima cosa, il metodo verifica se la componente sia un service o meno. In caso positivo, viene creato un nuovo stack a partire dagli insiemi delle politiche e dei permessi dello stack padre, quindi, viene chiamato il metodo *push(frameId, component)*.

Nel caso negativo, viene controllato il valore di *stackId* ricevuto. Se esso è pari al valore dell'identificatore del frame, significa che la componente è appena stata avviata dal sistema operativo. Pertanto, viene creato un nuovo stack vuoto e chiamato il metodo *push(frameId, component)*. Se invece i numeri differiscono, si controlla che il valore di *stackId* esista all'interno di *stacksSet* ed, eventualmente, si completa l'operazione d'inserimento del frame.

Infine, solamente nei casi di avvenuta operazione di push, viene controllata in *component* la presenza di politiche globali, che vengono eventualmente inserite all'interno della mappa della configurazione.

Al termine del metodo viene ritornato, zero come codice di errore, l'id del frame inserito come codice di conferma.

- *void updateComponent(int dadStackId, int frameId, Component component)*, completa un determinato frame temporaneo presente sullo stack indicato dal parametro *dadStackId*. Semplicemente, il metodo esegue un controllo sull'esistenza dello stack specificato e, in caso positivo, invoca su tale stack il metodo *fillMaps(component, frameId)* per inserire le informazioni aggiornate.

Fatto ciò controlla la presenza di politiche globali presenti nella componente ed, eventualmente, le inserisce nella mappa della configurazione.

- *int popComponent(int dadStackId, int frameId, String className)*, esegue l'operazione di pop del frame dallo stack indicato. Semplicemente, il metodo

verifica l'esistenza dello stack specificato dal parametro *dadStackId* e invoca su di esso il metodo *pop(frameId, className)*.

Infine, nel caso in cui l'operazione di pop abbia avuto successo, vengono rimosse le eventuali politiche globali associate al frame considerato, e controllato se lo stack interessato dall'operazione sia diventato vuoto. In caso positivo, esso viene rimosso dalla configurazione di sistema.

- *String getDirectCnf(int stackId)*, ritorna il metodo *getDirectPermissionAsString()* eseguito sullo stack individuato da *stackId*.
- *String getLocalCnf(int stackId)*, ritorna il metodo *getLocalPoliciesAsString()* eseguito sullo stack individuato da *stackId*.
- *String getGlobalCnf()*, ritorna una CNF contenente l' \wedge di tutte le politiche globali della configurazione.

4.2.3 SAT-Solver

Il SAT-Solver utilizzato per la validazione delle politiche sui permessi della configurazione è MiniSAT [23]. La scelta è stata motivata dalle ottime prestazioni che questo SAT-Solver possiede [24], oltre che alla disponibilità del codice sorgente scritto in C++. Infatti, data la complessità dei calcoli richiesti, utilizzare un solver a livello nativo ha permesso, come verrà mostranto nel § 5, di ottenere ottime prestazioni.

4.2.4 Application Installer

Il modulo Application Installer è costituito da un broadcast receiver. Esso è stato progettato per intercettare i messaggi di intent di sistema relativi all'installazione, disinstallazione e aggiornamento delle applicazioni. Inoltre, implementa i metodi per eseguire le codifiche necessarie per memorizzare correttamente, all'interno del database, le politiche e i permessi delle componenti delle applicazioni.

Il funzionamento della classe si basa sul metodo *onReceive()*. Al suo interno, mediante la action dell'intent ricevuto, si determina la tipologia del messaggio, eseguendo l'operazione più opportuna. Distinguiamo i seguenti tre casi.

ACTION_PACKAGE_ADDED Significa che una nuova applicazione è stata installata sul dispositivo. Pertanto, viene interrogato, tramite il nome del package specificato all'interno dell'intent, il PackageManager, ossia il gestore delle informazioni relative alle applicazioni installate fornito da Android. In questo modo è possibile ottenere dei riferimenti a tutte le componenti dell'applicazione appena installata, ed elaborarli per ottenere le informazioni da memorizzare sul database.

In particolare, come spiegato nel ¶ 4.1, vengono analizzati i campi `<meta-data>` presenti nelle componenti e indicanti i permessi e le politiche definite dallo sviluppatore. Quindi, il database viene interrogato sia per memorizzare eventuali permessi

definiti all'interno dell'applicazione, sia per codificare correttamente le politiche di sicurezza mediante gli identificatori univoci dei permessi.

Infine, ogni componente rilevata viene memorizzata all'interno del database specificando il nome della classe che la implementa, il package dell'applicazione dei appartenenza, il tipo di componente, i permessi che detiene e le politiche, correttamente codificate, che definisce.

ACTION_PACKAGE_REMOVED Significa che il package specificato dall'intent è stato rimosso, ossia che l'applicazione è stata disinstallata dal dispositivo. Vengono pertanto rimosse dal database tutte le componenti appartenenti all'applicazione, aventi cioè package uguale a quello indicato dall'intent ricevuto.

Tuttavia, non vengono rimossi dal database eventuali permessi definiti dall'applicazione, in quanto potrebbero essere utilizzati dalle politiche delle altre componenti ancora presenti nel sistema.

ACTION_PACKAGE_CHANGED Significa che il package di un'applicazione è cambiato a causa di un suo aggiornamento. Data l'impossibilità di determinare esattamente le modifiche apportate alle componenti dell'applicazione, questo caso viene gestito come una combinazione dei due precedenti. Viene pertanto richiesta un'operazione di *delete* sul database per cancellare tutte le entry relative al package specificato dall'intent, e poi interrogato il Package Manager per ottenere le informazioni aggiornate da memorizzare.

4.2.5 Security Database

Come per il modulo contenente il SAT-Solver, anche questo del database è stato implementato mediante codice nativo. Sebbene ogni versione di Android fornisca un database SQLite, implementarne una direttamente all'interno di SCP garantisce una totale compatibilità con il resto dell'applicazione. Infatti, versioni differenti delle API di Android possiedono versioni differenti del database, che potrebbero causare incompatibilità durante l'interazione tra i vari moduli.

Il database, molto semplicemente, è costituito da due sole tabelle; la prima, *Components*, contiene tutte le componenti delle applicazioni che utilizzano il nostro framework di sicurezza. La seconda, *Pesmission*, contiene l'elenco di tutti i permessi registrati sul sistema, ossia, i 146 permessi standard definiti dalle API 20 di Android [25], più gli eventuali permessi definiti da ogni applicazione, che vengono aggiunti durante la loro fase d'installazione.

In particolare, la tabella Components contiene i seguenti campi.

.id: di tipo INTEGER PRIMARY KEY AUTOINCREMENT, identifica univocamente la componente all'interno della tabella. Funge da chiave primaria e permette, mediante l'utilizzo dei cursori di android, una corretta rappresentazione delle componenti nel caso le si debbano mostrare all'interno di interfacce grafiche.

package: di tipo TEXT NOT NULL, contiene il nome del package dell'applicazione di appartenenza della componente. Esso è univoco a livello di applicazione, nel senso che non possono coesistere applicazioni aventi lo stesso package su uno stesso dispositivo. Ciò viene garantito dallo stesso sistema operativo in fase d'installazione, dove, nel caso in cui una nuova applicazione possiede un package analogo a quello di una già presente sul dispositivo, la procedura viene interpretata come un aggiornamento che sovrascrive l'applicazione meno recente.

name: di tipo TEXT NOT NULL UNIQUE, contiene il nome della classe che implementa la componente. È univoco, nel senso che non possono esistere sullo stesso sistema più classi rappresentanti componenti aventi lo stesso nome.

type: di tipo TEXT NOT NULL, indica la tipologia della componente che, come spiegato nella Sezione 1.1.1, può essere di tipo activity, content provider, service o broadcast receiver.

permission: di tipo TEXT NOT NULL, contiene una stringa dei permessi richiesti dalla componente, strutturata ponendo in \wedge i singoli permessi codificati per mezzo della tabella *Permission*.

policy: di tipo TEXT NOT NULL, contiene una stringa di una o più politiche di sicurezza dettate dalla componente e strutturate secondo la formattazione CNF. Eventuali più politiche sono separate per mezzo di un'opportuna codifica.

scope: di tipo TEXT NOT NULL, contiene una stringa indicante uno o più operatori di scope, descritti in Sezione 3.2, delle politiche indicate dal campo *policy*. Anche in questo caso, più operatori di scope vengono separati da una codifica.

Invece, la tabella Permission contiene i seguenti campi.

id: di tipo INTEGER PRIMARY KEY AUTOINCREMENT, identifica univocamente il permesso all'interno del sistema e assicura l'utilizzo della stessa codifica all'interno di tutte le politiche delle componenti delle applicazioni installate.

name: di tipo TEXT NOT NULL UNIQUE, indica il nome del permesso, che, come da specifica di Android [26], deve essere univoco.

4.3 Struttura di SCPLib

Le librerie d'accesso SCPLib implementano le metodologie per una corretta interazione tra le componenti delle applicazioni ed SCPApp. A grandi linee, l'idea alla loro base consiste nell'interagire con SCP durante le fasi di avvio, distruzione e invocazione delle diversi componenti, in modo da poter mantenere, all'interno del Component Stacks Manager, una configurazione aggiornata del contesto d'esecuzione del sistema.

Le librerie fornite implementano una versione modificata delle quattro classi delle componenti definite dalle API Android, più la classe *ContextScp* utilizzata per le fasi d'invocazione.

ActivityScp

La classe esegue automaticamente, all'interno del metodo *void onCreate(Bundle savedInstanceState)*, una fase di verifica della corretta registrazione ad SCPApp. In particolare, vengono distinti i seguenti casi.

- La componente è stata invocata all'interno di un flusso d'esecuzione registrato ad SCP. Significa che l'intent ricevuto possiede al suo interno l'identificatore temporaneo assegnato alla componente e quello del suo stack di appartenenza. Questi vengono utilizzati per completare la fase d'invocazione contattando il Request Receiver mediante il metodo *insert()*, specificando il *path COMPONENT_INVOKED*.
- La componente è stata invocata dal sistema operativo, all'avvio della sua applicazione di appartenenza. Significa che non fa parte di alcun flusso di esecuzione conosciuto ad SCPApp, pertanto, avvia la fase di registrazione invocando il metodo *insert()* di Request Receiver, specificando il *path NEW_APPLICATION_STARTED*.
- La componente è appena stata risvegliata dallo stato *pause*. Significa che ha già effettuato la registrazione ad SCPApp, pertanto, si limita a recuperare dallo *savedInstanceState* le informazioni necessarie ad eventuali future interazioni col prototipo.

All'interno del metodo *void onDestroy()*, invece, viene eseguita la chiamata al metodo *delete*, specificando il *path COMPONENT_REMOVED*, per avviare la procedura di rimozione del frame dal relativo stack di appartenenza.

ContentProviderScp

Classe che fornisce allo sviluppatore un content provider che rispetta il protocollo definito dal prototipo. Dato che i content provider sono legati al processo in cui il sistema operativo li invoca, non possiedono un ciclo vitale simile a quello visto per le activity. Pertanto, l'iscrizione ad SCP avviene in più punti della classe, in particolare all'interno dei metodi *Uri insert(...)*, *Cursor query(...)*, *String getType(...)*, *int update(...)* e *int delete(...)*.

La cancellazione da SCP, invece, avviene per mezzo di un meccanismo di *Weak Reference* implementato direttamente all'interno del Component Stacks Manager.

ServiceScp

La classe che implementa le componenti service, possiede un ciclo di vita avente dei punti d'ingresso e di uscita ben precisi. Questi vengono sfruttati per eseguire le operazioni di push e di pop dagli stack delle componenti gestiti da SCPApp. In particolare, i metodi *IBinder onBind(...)*, *void onRebind(...)*, e *int onStartCommand(...)* vengono utilizzati avviare la fase di push del frame associato alla componente, mentre, *boolean onUnbind(...)* e *void onDestroy()* per quella di pop.

BroadcastReceiverScp

Il ciclo vitale dei Broadcast Receiver è concentrato tutto nel metodo *void onReceive(Context context, Intent intent)*. Ciò ‘e dovuto al fatto che queste componenti vengono create dal sistema operativo solamente quando vi è un intent a loro indirizzato, e vengono distrutte non appena il metodo *onReceive()* ha termine. Ciò semplifica notevolmente l’implementazione della libreria, in quanto è sufficiente eseguire l’operazione di registrazione ad SCPApp all’ingresso del metodo *onCreate()*, e l’operazione di cancellazione subito prima del suo termine.

ContextScp

Classe che si occupa di gestire tutti i possibili meccanismi d’invocazione delle componenti. Ciò avviene fornendo agli sviluppatori dei metodi del tutto simili a quelli standard forniti da Android, ma che in realtà eseguono una serie di operazioni prima di avviare la normale invocazione della componente.

Per prima cosa, alla richiesta di una certa componente, il sistema operativo viene interrogato per ottenere la lista delle possibili componenti invocabili. Questa viene passata ad SCPApp mediante una chiamata al metodo *query()* del Request Receiver, che, come visto in Sezione 4.2.1, si occuperà di eseguire tutte le verifiche del caso. Se nessuna tra le componenti inviate all’applicazione può essere invocata, viene generata un’appropriata eccezione per comunicare l’assenza di una componente sicura. Altrimenti, distinguiamo i due seguenti casi.

- È stata individuata una sola componente sicura: essa viene invocata immediatamente attraverso i normali meccanismi forniti dall’Application Framework di Android.
- Sono state individuate più componenti sicure: vengono tutte mostrate all’utente all’interno di un elemento grafico, e invocata quella da lui scelta.

4.4 Problematiche riscontrate e scelte implementative

Durante la fase di progettazione di sviluppo del prototipo si sono presentate le due seguenti problematiche.

Mostrare all’utente, all’interno di SCPLib, più componenti sicure

Volendo implementare, all’interno delle SCPLib, la possibilità di far scegliere all’utente quale componente invocare nel caso in cui venissero individuate, da SCPApp, più componenti SAT, è stato necessario inserire un elemento grafico per poter mostrare all’utente le diverse possibilità.

Tale elemento consiste di un widget grafico detto *dialog*, ossia una schermata che compare al di sopra dell’interfaccia dell’activity correntemente mostrata all’utente.

Quando vengono mostrati all’utente elementi grafici, come dialog o activity, Android termina istantaneamente il flusso di codice del metodo che ha invocato l’elemento grafico. Così facendo, risulta impossibile mantenere una comunicazione sincrona tra l’applicazione cliente e le SCPLib nel caso in cui più componenti fossero candidate ad essere eseguite.

Per questo motivo è stato scelto di fornire allo sviluppatore due differenti modalità di interazione con le librerie di SCP. La prima, impone di non mostrare mai all’utente le informazioni riguardanti le componenti invocabili individuate. Pertanto SCPLib automaticamente avvia la componente avente il minor numero di permessi, garantendo un comportamento sincrono.

La seconda, invece, permette allo sviluppatore di specificare una porzione di codice che verrà eseguita dalla libreria non appena l’utente avrà terminato la scelta della componente da invocare, implementando, quindi, un comportamento asincrono.

Comunicazione tra SCPLib e SCPApp

La scelta del meccanismo di comunicazione tra SCPLib ed SCPApp ha richiesto un’attenta analisi dei meccanismi di IPC disponibili in Android. Infatti, non tutte le componenti che costituiscono le applicazioni sono in grado di utilizzare tutti i meccanismi forniti, e, alcuni di essi, non sono disponibili a livello nativo.

In seguito sono descritte brevemente le caratteristiche principali dei meccanismi di IPC disponibili su Android.

Socket, forniti direttamente dal Kernel Linux, forniscono un meccanismo di IPC che richiede, da parte dell’applicazione che intende utilizzarlo, il permesso `android.permission.INTERNET`. Sono utilizzabili sia a livello nativo sia in applicazioni scritte in Java.

Memoria Condivisa e Pipe, similmente ai precedenti, sono disponibile sia per applicazioni native sia per quelle in Java e richiedono, a causa del meccanismo di *sandbox* presentato in Sezione 1.1.2, i permessi `android.permission.READ EXTERNAL STORAGE` e `android.permission.WRITE EXTERNAL STORAGE`.

Binder, meccanismo di IPC ottimizzato per Android. È disponibile solamente per applicazioni scritte in Java, in quanto la Native Development Kit (NDK) non fornisce API per il suo utilizzo [27].

Permette sia una comunicazione sincrona (mediante *Android Interface Definition Language* (AIDL)) sia asincrona (mediante *Messenger*) per la comunicazione verso componenti di tipo service [28]. Non richiede alcun permesso per essere utilizzato.

Intent, meccanismo basato sul precedente [29], è disponibile solamente per applicazioni scritte in Java. Permette di comunicare con componenti di tipo service, broadcast receiver ed activity. Non richiede alcun permesso per essere utilizzato.

ContentResolver, meccanismo basato su URI che permette una comunicazione sia sincrona sia asincrona verso componenti di tipo content resolver. È disponibile solamente per applicazioni scritte in Java. Non richiede alcun permesso per essere utilizzato.

Risulta quindi chiaro come la scelta del meccanismo di IPC ponga dei forti vincoli sull'implementazione del prototipo. Infatti, sono stati scartati tutti i meccanismi richiedenti permessi, in quanto, a causa delle SCPLib, ciò avrebbe conseguentemente obbligato le applicazioni client a richiederli. Inoltre, abbiamo stabilito che il meccanismo di comunicazione dovesse fornire sia un comportamento sincrono, sia uno asincrono, in modo da non dover imporre vincoli agli sviluppatori.

In un primo momento la progettazione si è orientata verso l'utilizzo dei Binder. Ciò è dovuto al fatto che, dovendo SCPApp lavorare in background costantemente, risultava logico e sensato implementarlo tramite Service. Inoltre, tale soluzione forniva due vantaggi non trascurabili:

1. Per comunicare con i service è possibile utilizzare il meccanismo dei binder, più efficiente di quello degli intent [29];
2. I service possono essere dichiarati come *foreground*, ossia, come mostrato in Tab. 4.1, possono assumere il massimo livello di priorità (liv. 1) nel meccanismo di Android di rilascio dei processi e delle componenti in caso di limitate risorse computazionali.

Livello	Tipo di Processo	Descrizione
1	Proc. Foreground	Processi che supportano l'interazione corrente del sistema con l'utente. Es: activity correntemente visualizzata a schermo, service ad essa legati o lanciati mediante il metodo <code>startForeground()</code> , eventuali broadcast receiver che stanno eseguendo il metodo <code>onReceive()</code> ..
2	Proc. Visibili	Processi che non possiedono alcuna componente foreground ma che possono comunque interessare ciò che l'utente può vedere sullo schermo. Es: processi aventi un'activity non più in foreground ma che risulta ancora visibile all'utente o processi aventi service legati ad activity visibili o in foreground.
3	Proc. di Servizio	Processi che eseguono servizi non ricadenti nelle prime due categorie.
4	Proc. in Background	Processi in cui risiede un activity non più visibile dall'utente.
5	Proc. Vuoti	Processi che non possiedono componenti di applicazioni attive.

TABELLA 4.1: Gerarchia dei processi in ordine di importanza, dal più alto (Liv.1) al più basso (liv.5).

Tuttavia, come mostrato in questo capitolo, i meccanismi di IPC scelti sono stati quelli di intent e di contentResolver. Tale scelta, è dovuta al fatto che SCPApp deve poter essere contattata da tutte le componenti delle applicazioni client. Nel caso in cui il Request Receiver fosse stato implementato mediante service, non si sarebbero

potuti eseguire binding diretti da parte degli eventuali broadcast receiver presenti nelle applicazioni clienti.

Infatti, Android impone dei vincoli molto severi sui broadcast receiver: essi non devono né invocare interfacce utente proprie, come eventuali dialog, né eseguire il bind a servizi [30].¹ Questi vincoli sono dovuti al fatto che non possono essere eseguite operazioni a lungo termine all'interno dei broadcast receiver. Inoltre, Android imposta un timer di dieci secondi a ogni chiamata del metodo `onReceive()`, allo scadere del quale, a meno che non abbia già concluso la propria operazione, il broadcast receiver viene considerato come bloccato e candidato a essere liberato dal sistema operativo [30].

Dato che le SCPLib devono fornire agli sviluppatori dei meccanismi del tutto simili alle normali API di Android, abbiamo ritenuto troppo restrittivo il vincolo di obbligare lo sviluppatore a utilizzare le componenti broadcast receiver sempre e solo su thread a loro dedicati.

¹I broadcast receiver permettono l'esecuzione di alcuni comandi su service. Ad esempio, `startService()` e `peekService()` permettono, rispettivamente, di avviare un service e di eseguire un bind solo su service già avviati. Tuttavia, non è possibile eseguire le due chiamate in maniera sincrona senza ricadere nel caso del normale bind a service, altamente sconsigliato dalla documentazione android.

Capitolo 5

Risultati e Test

L’attività computazionale più onerosa svolta dal nostro prototipo è la risoluzione dei problemi SAT. Come mostrato nel § 4, all’invocazione di ogni nuova componente vengono generate più istanze del problema SAT che devono essere valutate. Chiaramente, le prestazioni del SAT-Solver sono cruciali per la fattibilità del nostro approccio. Al meglio delle nostre conoscenze, non sono stati presentati in letteratura esperimenti e risultati riguardanti la valutazione di istanze di problemi SAT su dispositivi mobili.

I risultati ottenuti sono presentati in seguito e mostrano come le prestazioni del SAT-Solver preso in considerazione, MiniSat, siano compatibili con un utilizzo su dispositivi mobili.

5.1 Obiettivo dei test

L’obiettivo dei test condotti è stato quello di valutare la capacità computazionale di un dispositivo mobile durante l’esecuzione delle operazioni di verifica delle politiche di sicurezza eseguite dal SAT-Solver. Tali operazioni possono diventare onerose al crescere del numero di politiche e di componenti. Affinchè il prototipo sia utilizzabile, deve essere garantito che tali calcoli non influiscano negativamente sull’interazione utente-dispositivo. Eccessivi tempi di attesa o ritardi potrebbero infatti essere interpretati come malfunzionamenti delle applicazioni, se non addirittura provocare la soppressione del processo da parte del sistema operativo Android.

A tale scopo sono state valutate diverse applicazioni ottenute dal Google Play Store, per determinare delle condizioni realistiche.

5.2 Struttura degli esperimenti

Gli esperimenti condotti sono stati studiati a partire dall’analisi delle 804 applicazioni gratuite più scaricate per Android.¹ In particolare, per ogni applicazione, sono stati

¹Le applicazioni fanno riferimento a Marzo 2013.

considerati il numero di componenti che la costituiscono e la quantità di permessi da essa richiesti. La Tabella 5.1 mostra i valori individuati.

Tipo di Elemento	N. Medio	N. Massimo
Componenti	21	298
Permessi	11	85

TABELLA 5.1: Componenti e permessi delle applicazioni più diffuse su Android.

Questi valori hanno permesso di caratterizzare delle possibili configurazioni d'esecuzione del sistema in termini di permessi e componenti coinvolte. Tali configurazioni sono costituite da liste di formule CNF generate per mezzo del servizio online Tough SAT Project [31].

I test sono stati suddivisi in tre categorie che si differenziano per il numero di variabili/permessi coinvolti. In particolare, abbiamo testato il nostro sistema con 50, 150 e 300 permessi.

In ogni test abbiamo generato 31 specifiche formule CNF, contenenti un numero di clausole crescente (da 10 a 500). I valori sono arbitrari in quanto non esistono valori di riferimento, ma si consideri che nel nostro caso di studio descritto nell'Esempio 3.2, il numero di clausole dato dal \wedge di tutte le politiche di tutte le componenti dell'applicazione è pari a 12.

Per ogni gruppo di test, sono state inserite all'interno della tabella Components del database, Sezione 4.2.5, un numero di 31 entry, tutte invocabili con una stessa azione di intent.

Anche in questo caso, si consideri che gli intent più comuni, per la condivisione di immagini e di porzioni di testo tra le applicazioni, possono essere gestiti, in un dispositivo *out of the box*, al più da due applicazioni.² La stessa prova, condotta su un dispositivo personale utilizzato quotidianamente, ha mostrato come tali intent possano essere gestiti, rispettivamente, da 18 e 17 differenti applicazioni.

Per ogni test sono stati valutati i tempi impiegati dal prototipo a soddisfare la richiesta d'invocazione di una componente. In particolare, gli istanti temporali considerati sono i seguenti.

- t_0 , momento d'interazione dell'utente con un'interfaccia grafica che avvia la procedura d'invocazione di una componente mediante intent implicito.
- t_1 , istante di rappresentazione a video del dialog riportante tutte le componenti individuate dal SAT-Solver che possono soddisfare la configurazione del sistema.

Ambiente di test

Per l'esecuzione dei test è stato adoperato un dispositivo mobile nella fascia di

²Le prove sono state condotte su un dispositivo virtualizzato, mediante l'Android Virtual Device, avente versione 4.4.2 del sistema operativo.

prezzo di 150\$, modello LG Nexus 7 (2012), equipaggiato con un processore ARM quad-core da 1.2 GHz, 1 GB di memoria RAM e versione del sistema operativo Android 4.4.4.

Risultati

Le Figure 5.2 e 5.1 riportano i risultati ottenuti per le tre classi di test descritte in Sezione 5.2. In particolare, per ogni test, sono mostrati i tempi richiesti dal dispositivo per eseguire 31 verifiche e il numero di componenti *SAT* individuate.

Considerazioni

Come prevedibile, dato che le specifiche sono generate in maniera casuale dal sistema, all'aumentare delle clausole si hanno meno componenti *SAT* individuate.

È interessante notare che, anche sotto condizioni particolarmente pessimistiche (ad esempio, 300 permessi usati contro gli 85 permessi massimi riportati dalla Tae. 5.1), le prestazioni non decadono in maniera sensibile. Infatti, i risultati ottenuti non superano mai gli 1,5 secondi. Pertanto ci aspettiamo un corretto funzionamento del prototipo in un suo utilizzo a regime.

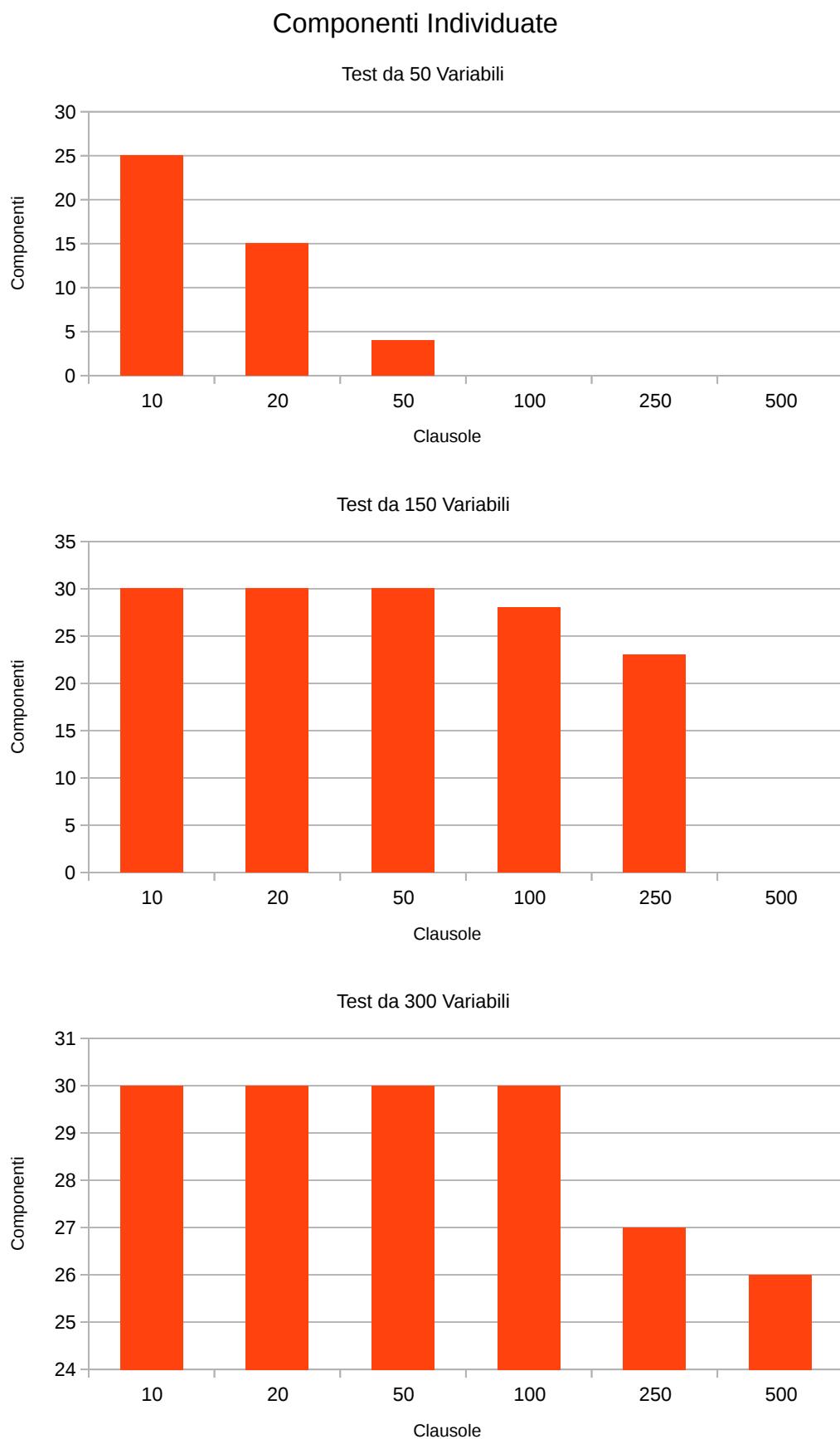


FIGURA 5.1: Risultati dei test sulle componenti individuate con 50, 150 e 300 variabili.

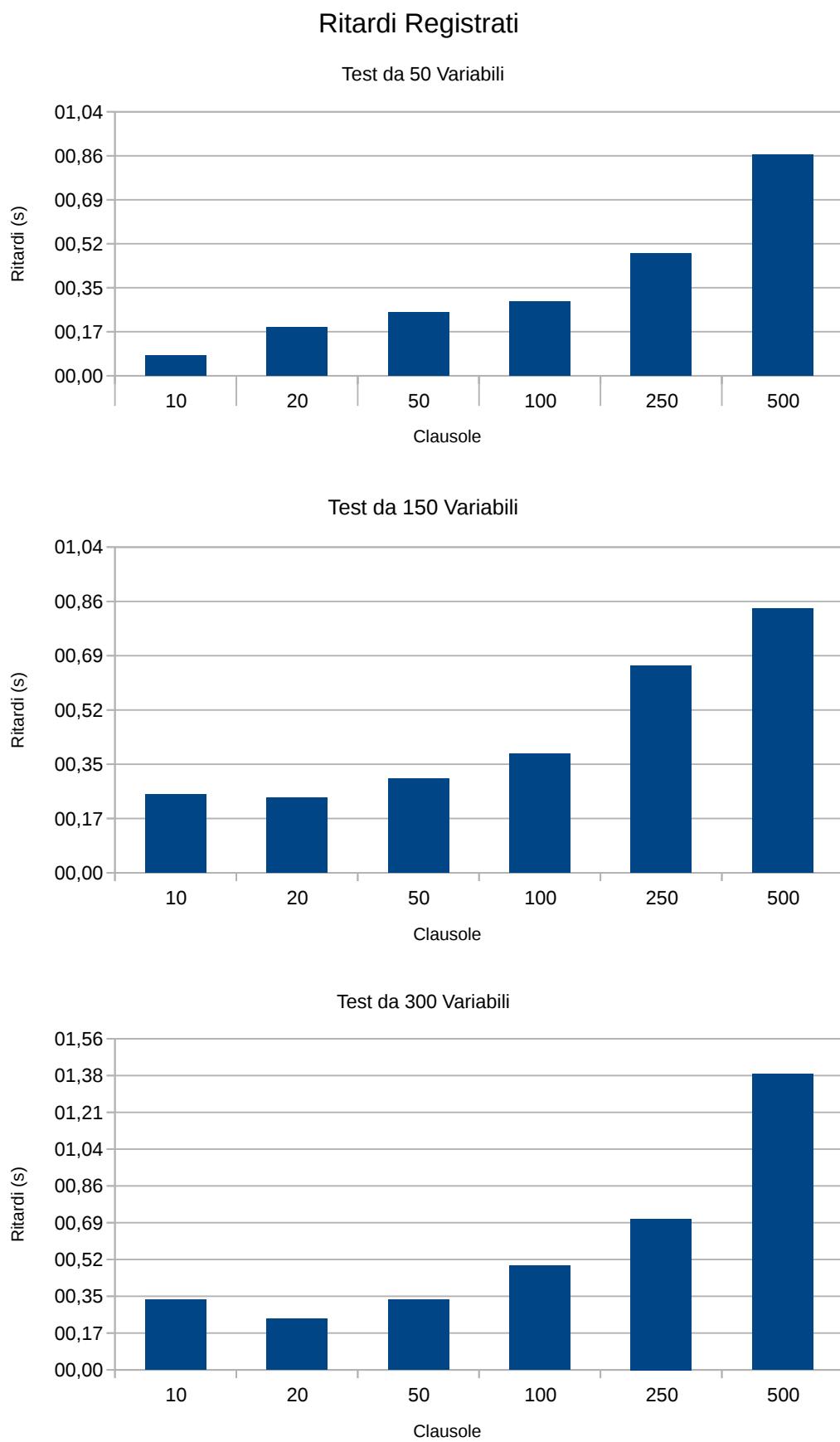


FIGURA 5.2: Risultati dei test sui ritardi misurati con 50, 150 e 300 variabili.

Capitolo 6

Conclusioni e Sviluppi Futuri

Nella trattazione di questa tesi è stata presentata l'implementazione di un'estensione dell'Android Security Framework (ASF). Essa fornisce utili strumenti agli sviluppatori per specificare politiche di sicurezza più fini di quelle attualmente definibili mediante il framework fornito dal sistema operativo, allo scopo di garantire un'interazione sicura tra le singole componenti delle applicazioni.

Il caso di studio presentato all'interno del § 2, costituisce un chiaro esempio dei limiti posti da ASF. Infatti, applicazioni sensibili come quelle di gestione di pagamenti online, possono richiedere requisiti di sicurezza non implementabili con il framework standard di Android.

Nel § 3 è stato presentato il modello formale dell'estensione proposta, descrivendo la sintassi del linguaggio di politiche utilizzato e come esso possa essere applicato al contesto d'esecuzione dei dispositivi Android. Esso prevede di applicare permessi e politiche di sicurezza alle singoli componenti delle applicazioni, in modo da poter gestire anche l'interazione tra differenti applicazioni. Utilizzando il caso di studio come esempio, si è mostrato come alcune semplici politiche, possano permettere la messa in sicurezza di diverse sezioni dell'applicazione, garantendo una sua interazione sicura con le altre applicazioni presenti sul dispositivo.

Nel § 4, invece, è stato descritto dettagliatamente il prototipo implementato, con i suoi meccanismi di funzionamento, le classi che lo compongono e le scelte implementative effettuate. Infine, nel § 5, i test eseguiti hanno mostrato come SCP sia compatibile ad un utilizzo su dispositivi mobili.

Possiamo individuare due possibili estensioni future.

1. Il prototipo può essere esteso alle applicazioni con codice nativo. Infatti, le librerie fornite agli sviluppatori, permettono attualmente di implementare la maggior parte delle applicazioni disponibili su Android, in quanto forniscono una versione modificata delle quattro componenti activity, service, content provider e broadcast receiver, in grado di interagire col prototipo implementato.

2. Il linguaggio delle politiche può essere esteso. In particolare possiamo introdurre predicati e proposizioni per implementare politiche contestuali. Tali politiche possono riguardare l'utilizzo delle applicazioni in ambienti o orari particolari, ad esempio a lavoro o nel weekend. Ovviamente tale modifica comporta una redifinizione del linguaggio di politiche e del loro metodo di validazione.

Bibliografia

- [1] Mobile Statistics. Quarterly device sales in 2013 q1. Url: <http://www.mobilestatistics.com/mobile-statistics/>, August 2014.
- [2] Android Open Source Project. Android. Url: <https://source.android.com/>, August 2014.
- [3] Android Developers. Android native development kit. Url: <https://developer.android.com/tools/sdk/ndk/index.html#Using>, August 2014.
- [4] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.
- [5] William Enck, Damien Ochteau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.
- [6] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.
- [7] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys '11*, pages 239–252, New York, NY, USA, 2011. ACM.
- [8] Zheran Fang, Weili Han, and Yingjiu Li. Permission based android security: Issues and countermeasures. *Computers & Security*, 43:205–218, 2014.
- [9] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.
- [10] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. Runtime verification meets android security. In *NASA Formal Methods*, pages 174–180. Springer, 2012.

- [11] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.
- [12] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.
- [13] Giovanni Russello, Bruno Crispo, Earlene Fernandes, and Yury Zhauniarovich. Yaase: Yet another android security extension. In *Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)*, pages 1033–1040. IEEE, 2011.
- [14] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.
- [15] Philippe Balbiani, Yannick Chevalier, and Marwa El Houri. A logical approach to dynamic role-based access control. In Danail Dochev, Marco Pistore, and Paolo Traverso, editors, *AIMSA*, volume 5253 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2008.
- [16] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC ’82, pages 169–180, New York, NY, USA, 1982. ACM.
- [17] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.
- [18] Ruj Akavipat, Mahdi N. Al-Ameen, Apu Kapadia, Zahid Rahman, Roman Schlegel, and Matthew Wright. ReDS: A framework for reputation-enhanced DHTs. *IEEE Transactions Parallel and Distributed Systems*, 25(2):321–331, February 2014. doi: 10.1109/TPDS.2013.231.
- [19] Liang Cai, Sridhar Machiraju, and Hao Chen. Defending against sensor-sniffing attacks on mobile phones.
- [20] Felix Rohrer, Yuting Zhang, Lou Chitkushev, and Tanya Zlateva. Poster: Role based access control for android (rbaca). *Boston University, MA USA, Tech. Rep*, 2012.
- [21] Alessandro Armando, Gabriele Costa, and Alessio Merlo. Formal modeling and reasoning about the android security framework. In *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*, pages 64–81, 2012.

- [22] A. Armando, G. Costa, and R. Carbone. Android permissions unleashed (technical report). Technical report, 2014.
- [23] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.
- [24] SAT Competitions. Sat 2007 competition. Url: <http://www.satcompetition.org/>, August 2014.
- [25] Android Developers. Manifest permission. Url: <http://developer.android.com/reference/android/Manifest.permission.html>, August 2014.
- [26] Android Developers. Permission. Url: <http://developer.android.com/guide/topics/manifest/permission-element.html>, August 2014.
- [27] Android Developers. Android ndk revision 9d. Url: <https://developer.android.com/tools/sdk/ndk/index.html/>, March 2014.
- [28] Android Developers. Bound services. Url: <http://developer.android.com/guide/components/bound-services.html/>, August 2014.
- [29] Aleksandar Garganta. Deep dive into android ipc/binder framework. Url: https://thenewcircle.com/s/post/1340/deep_dive_into_android_ipc_binder_framework_at_andevcon_iv/, August 2014.
- [30] Android Developers. Broadcast receiver. Url: <http://developer.android.com/reference/android/content/BroadcastReceiver.html/>, August 2014.
- [31] H. Yuen and J. Bebel. Tough sat project. Url: <https://toughsat.appspot.com/>, August 2014.