Università degli Studi di Udine
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Triennale in Informatica

Tesi di Laurea

Sistema open source a linea di comando per la generazione di documenti PDF legalmente validi

Candidato:
Marco Del Toso

Relatore:
Prof. Marino Miculan

Anno Accademico 2007 - 2008

Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

Alla mia famiglia

Ringraziamenti

Desidero ringraziare tutte le persone che mi hanno sostenuto e mi sono state vicine in questi tre anni di università. Inizio naturalmente dai miei genitori che mi hanno sempre spronato ad andare avanti e appoggiato in tutte le mie scelte. Sin dalle superiori non mi hanno mai imposto nessuna decisione, lasciandomi scegliere in modo indipendente il mio futuro. Poi voglio ringraziare mio fratello Fulvio che mi ha portato sulla strada dell'informatica e solo seguendo le sue orme sono potuto arrivare fino a qui. Poi, con la scusa che doveva cambiare il portatile, ne ha comprato uno nuovo che sto usando proprio adesso per fare la tesi. Non voglio dimenticare tutti gli amici che mi hanno aiutato a distrarmi quando avevo del poco tempo libero. Non faccio nomi per non dimenticare nessuno. In particolare per devo ringraziare Arjuna che mi ha prestato quasi tutti i libri su cui ho studiato. Chiaramente tra gli amici devo includere anche tutti i miei compagni di squadra, anche se con la storia del sito mi hanno fatto dannare. Infine devo ringraziare tutti i compagni di corso e di studio che mi hanno permesso di confrontarmi con idee e punti di vista diversi. Naturalmente, in particolare ringrazio Marco per avermi tenuto compagnia durante le lezioni e per aver prestato il lettore della smart card regionale che sto ancora aspettando!! Infine un ringraziamento doveroso va al Dott. Emanuele Pucciarelli e al Prof. Marino Miculan per il supporto che mi hanno dato durante lo sviluppo del progetto. Ancora grazie a tutti!!

Indice

1

Introduzione

Il presente documento ha il compito di presentare la realizzazione di un'estensione del tool *pdftk* (vedi sezione 4.2.1) per aggiunge l'operazione di firma digitale dei documenti in formato PDF tramite l'uso di smart card. Per poter fare ciò è stato possibile sfruttare le funzionalità appena descritte fornite già da un altro software cioè *OpenSignPDF* (vedi sezione 4.2.3). Si rimanda ai capitoli successivi per i dettagli implementativi.

Il documento è suddiviso in sei capitoli (compreso questo di introduzione), più due appendici, ognuno dei quali descrive uno dei vari aspetti necessari per comprendere più a fondo concetti di base e funzionamento del software creato.

Nel capitolo 2 si parla, in generale, della firma digitale e della marcatura temporale. In particolare si descrive il processo di creazione della firma digitale basata sulla crittografia a chiave asimmetrica e di come si può generare una firma digitale di un documento. Il capitolo descrive il funzionamento del sistema dei certificati a chiave pubblica e di come essi vengono distribuiti attraverso le smartcard. Inoltre viene presentato il concetto di marcatura temporale, con relativa descrizione delle metodologie di creazione. Infine si accenna al quadro normativo italiano riguardo al valore legale della firma digitale dei documenti, in particolare dei documenti in formato PDF.

Nel capitolo 3 vengono descritte le caratteristiche principali del formato PDF. In particolare viene analizzata la sintassi dei file PDF, con riferimento alla struttura dei file e agli oggetti che li compongono. Inoltre si descrive la struttura dei documenti PDF e di tutti gli elementi che possono contenere. Infine si parla di come è possibile inserire una firma digitale e il relativo timestamp all'interno di un file PDF.

Il capitolo 4 contiene l'analisi del problema che il nostro progetto vuole risolvere. In particolare si analizzano i 1 e software che sono stati utili alla realizzazione del programma, con informazioni relative al loro funzionamento. L'ultima sezione de-

scrive gli strumenti e i componenti necessari allo sviluppo del progetto, introducendo, in modo teorico, le soluzioni e metodologie che sono state pensate per realizzare il software.

Nel capitolo 5 vengono descritte in maniera sistematica e molto dettagliata tutte le modifiche che sono state apportate ai software (con riferimenti ai listati riportati nell'appendice A) per creare l'estensione di pdftk. È stato deciso di realizzare più sezioni, ognuna delle quali tratta i cambiamenti relativi ad un specifico software, più una sezione che descrive com'è stato possibile far interagire i vari programmi. In fondo è stata inserita una sezione che descrive i principali problemi che sono stati incontrati durante la realizzazione del progetto.

Nel capitolo 6 vengono tratte le conclusioni su quanto si è riusciti a implementare e su quanto andrebbe migliorato.

In fondo al documento è stata aggiunta l'appendice A in cui abbiamo inserito i listati relativi alle modifiche più importanti e interessanti apportate al codice di pdftk e alle classi di OpenSignPDF.

Per concludere il documento, abbiamo deciso di inserire l'appendice B che vuole essere un piccolo manuale per l'installazione e l'utilizzo della nuova funzionalità di firma aggiunta a pdftk. Sono riportate tutte le istruzioni da seguire per la corretta installazione e successiva configurazione del software, seguite da una spiegazione delle varie opzioni che si possono specificare e una serie di esempi di utilizzo.

2

Firma digitale

In questo capitolo cercheremo di descrivere in modo conciso ed efficace il concetto di firma digitale. La firma digitale è un sistema di autenticazione dei documenti digitali che permette di associare ad un dato documento un'identità. Concettualmente si basa sulle stesse basi della firma autografa su carta.

Prima di iniziare però a parlare di firma digitale vera e propria, dobbiamo introdurre il meccanismo di crittografia a chiave pubblica (detto anche a chiave asimmetrica) su cui la firma si basa.

2.1 Crittografia a chiave pubblica (o a chiavi asimmetriche)

Questo tipo di crittografia è un sostanziale miglioramento della crittografia classica che si basa sulla condivisione di un segreto tra due parti (sorgente e destinatario) per garantire la confidenzialità dei messaggi. Come si può capire dal nome, la crittografia asimmetrica si basa su una coppia di chiavi, chiamate *chiave pubblica* e *chiave privata*. Si parla di chiavi asimmetriche in quanto, in questo caso, sorgente e destinatario del messaggio non sono uguali e non condividono lo stesso segreto. Attualmente la crittografia a chiave pubblica non sostituisce la "vecchia"crittografia a chiave privata, ma piuttosto la completa.

La crittografia a chiave asimmetrica è nata con un duplice scopo:

- garantire lo scambio sicuro di chiavi, senza doversi appoggiare ad una fonte affidabile con cui condividere una chiave privata
- garantire l'integrità e la provenienza di un messaggio tramite la firma digitale

Per raggiungere questi obiettivi e necessario che ogni parte possegga una coppia di chiavi, di cui una (chiave privata) deve essere mantenuta segreta in quanto viene

utilizzata per decifrare i messaggi e creare le firme, mentre l'altra (quella pubblica) può essere conosciuta da chiunque e serve per cifrare i messaggi e verificare le firme. In questo modo chi cifra il messaggio non può poi decifrarlo e chi deve verificare la firma non la può creare.

Naturalmente la coppia di chiavi scelta deve rispettare almeno i due seguenti criteri per poter funzionare. Innanzitutto non deve essere possibile (oppure deve essere computazionalmente troppo difficile) risalire alla chiave privata conoscendo solo l'algoritmo e la chiave pubblica. Secondo, deve essere facile cifrare o decifrare un messaggio se si conosce rispettivamente la chiave pubblica e la chiave privata; solitamente gli schemi di crittografia a chiave asimmetrica utilizzano algoritmi che si basano su problemi che sono P-completi in un verso e NP-completi nel verso opposto (moltiplicazione e fattorizzazione oppure esponenziazione e logaritmi).

Esistono tre diversi possibili tipi di utilizzo per la crittografia asimmetrica:

- cifrare\decifrare messaggi per garantire la confidenzialità del contenuto del messaggio
- creare firme digitali per garantire l'autenticità di un messaggio e legarlo al suo creatore
- scambiare chiavi private in modo sicuro ed affidabile

La sicurezza dello schema a chiave pubblica si basa principalmente sul fatto che il sistema di codifica\decodifica sfrutta problemi che sono computazionalmente facili, mentre i problemi da risolvere per forzare i codici sono difficili (NP-completi). Questo sistema però comporta l'uso di numeri molto molto grandi (ad es. 1024 bit), rendendo i meccanismi a chiave asimmetrica molto più lenti rispetti a quelli a chiave simmetrica.

Il più famoso e più usato schema a chiave pubblica è quello ideato nel 1977 da Rivest, Shamir e Adleman (RSA). Questo algoritmo si basa su un problema matematico che ha complessità $O((\log n)^3)$ (facile) mentre il suo problema inverso ha complessità $O(e^{\log n \log\log n})$ (difficile). Dunque lo schema rispetta il criterio specificato in precedenza.

RSA consiste in due parti principali, una prima fase di inizializzazione della coppia di chiavi e la seguente fase di utilizzo. Una volta generata la coppia di chiavi il possessore deve rendere disponibile a chiunque la sua chiave pubblica e mantenere segreta la sua chiava privata. La fase di utilizzo si suddivide a sua volta nel processo

di cifratura e quello di decifratura del messaggio. Quando due entità devono scambiarsi un messaggio in modo sicuro, il mittente deve, per prima cosa, procurarsi la chiave pubblica del destinatario, cifrare il messaggio con essa e inviarlo al destinatario. Quando il destinatario riceve il messaggio non deve fare altro che decifrarlo utilizzando la sua chiave privata. Il tutto funziona in quanto i due algoritmi per cifrare e decifrare sono uno l'inverso dell'altro e quindi il destinatario ottiene il messaggio decifrato. Una volta che il messaggio è stato cifrato con la chiave pubblica del destinatario, l'unico che può decifrarlo è chi conosce la chiave privata del destinatario. Senza di essa è computazionalmente impossibile risalire al messaggio in chiaro. Inoltre le chiavi utilizzate attualmente da RSA sono di dimensione sufficientemente grande, nell'ordine di 512 2 o più per rendere praticamente impraticabili attacchi di tipo brute force.

Uno dei problemi critici per lo schema di crittografia a chiave pubblica sta nella distribuzione delle chiavi pubbliche. Attualmente esistono diversi metodi, più o meno sicuri, di pubblicare la propria chiave pubblica, tra cui elenchiamo:

- Public announcement. Il proprietario della chiave la distribuisce a tutti color che ne hanno bisogno oppure la pubblica in broadcast a tutta la comunità
- Public available directory. Il proprietario registra la chiave in una directory pubblica accessibile in modo elettronico
- Public-key authority. Simile alla directory pubblica, con la sola differenza che esiste una public authority che regola l'accesso alla directory e la distribuzione delle chiavi
- Public-key certificate. Una Certification Authority crea dei certificati elettronici contenenti il legame tra un'identità e le relativa chiave pubblica

Tra questi meccanismi, il più sicuro, più utilizzato e con le migliori performance è quello dei public-key certificates. In seguito ne analizziamo meglio il funzionamento, in quanto il nostro progetto sfrutta proprio questa tecnica tramite l'uso di una smart card (parleremo di questo nella sezione 2.2).

2.1.1 Public-key certificates

Vediamo più in dettaglio come funziona il meccanismo di creazione e distribuzione dei public-key certificates da parte di una Certification Authority.

Innanzitutto è necessaria l'esistenza di una sorgente attendibile e fidata, chiamata Certificate Authority, di cui è necessario conoscere la chiave pubblica. Qui si pone nuovamente il problema di conoscere in modo fidato una chiave pubblica. Per risolverlo è stata pensata una gerarchia di Certification Authorities in cui i certificati della authorities discendenti sono firmate dal predecessore, fino al nodo radice, il quale deve essere per forza ritenuto fidato. Sul sito del CNIPA (Centro Nazionale per l'Informatica nella Pubblica Amministrazione) è possibile trovare una lista dei certificatori ufficialmente riconosciuti dalla legge italiana.

Una volta ottenuta la chiave pubblica della Certification Authority (CA), per ottenere il certificato èsufficiente inviare all'autorità la propria chiave pubblica (ed eventualmente alcune informazioni sulla propria identità). A questo punto, la Certificate Authority non fa altro che cifrare la nostra chiave pubblica, assieme alle informazioni relativa alla nostra identità e al certificato stesso (parleremo in seguito del formato e del contenuto di un particolare standard per i certificati, nella sezione 2.1.1) con la propria chiave privata. A questo punto il certificato così creato può essere spedito al richiedente. Per verificare l'autenticità ed il contenuto del certificato sarà sufficiente decifrarlo utilizzando la chiave pubblica della CA.

Ognuna delle parti che deve instaurare una comunicazione sicura deve ottenere un certificato per garantire l'autenticità della propria chiave pubblica da distribuire. Se, ad esempio, Alice (A) e Bob (B) devono comunicare, dovranno inizialmente scambiarsi i propri certificati. Entrambi saranno in grado di verificare l'autenticità del certificato (e quindi della chiave pubblica dell'interlocutore) semplicemente conoscendo la chiave pubblica (o il certificato) della CA che ha firmato il certificato dell'altro. Quando A possiede una hash della chiave pubblica di B può utilizzarla per cifrare il messaggio che deve mandare a B, che sarà l'unico in grado di decifrarlo, in quanto è l'unico che conosce la chiave privata di B (cioè se stesso).

Questo meccanismo presenta molti vantaggi rispetto a quelli elencati nella sezione precedente (vedi pagina 5). Infatti il certificato non è mantenuto dalla Certification Authority, ma localmente dal richiedente, evitando l'accesso alla CA per la richiesta della chiave pubblica e risolvendo il problema di bottlenecks ("collo di bottiglia"). Viene inoltre risolto il problema chiamato di *forgery*, cioè di qualcuno che si spaccia per qualcun altro. Infatti il certificato viene firmato con la chiave privata della CA, che è l'unica che la può conoscere. Quindi è possibile capire se il certificato proviene veramente dalla CA solo conoscendone la chiave pubblica. Questo dunque è l'unico problema che ancora rimane per il meccanismo dei public-key certificates. Infine, se

un malintenzionato riesce a "rubare"la chiave privata di Alice, per evitare guai Alice dovrà solo generare una nuova coppia di chiavi e richiedere un nuovo certificato alla CA, bloccando e invalidando il certificato precedente.

X.509 Certificates

Fino ad ora abbiamo parlato di come ottenere un certificato, ma non abbiamo analizzato in modo approfondito quali informazioni specifiche contenga e in quale formato sia definito. Per quanto riguarda i public-key certificates è stato definito uno standard chiamato X.509 che include, tra gli altri, la descrizione dei formati standard da seguire per i certificati. Vediamo dunque, com'è un elenco di cosa contiene un certificato X.509 (definito in modo completo in [1]):

- versione
- numero seriale del certificato (univoco per una specifica CA)
- identificatore dell'algoritmo di firma
- ente emettitore (nome della CA)
- periodo di validità (date da - a)
- soggetto (possessore del certificato)
- algoritmo per l'utilizzo della chiave pubblica
- chiave pubblica
- codice univo dell'emittente (dalla versione 2, facoltativo)
- codice univoco del soggetto (dalla versione 2, facoltativo)
- estensioni (dalla versione 3, facoltativo)
- algoritmo di firma del certificato
- firma del certificato

Dalla descrizione del formato si nota che, oltre alle informazioni relative all'identità del possessore e alla sua chiave pubblica, sono presenti molte informazioni aggiuntive. Tra esse, importante è il periodo di validità, espresso tramite due date, che indicano la data prima della quale e dopo la quale il certificato non è ritenuto valido

(cioè data di emissione e data di scadenza del certificato). Un altro campo interessante è l'algoritmo che deve essere usato per la cifratura con la chiave pubblica del possessore, in quanto specifica che tipo di chiave pubblica è contenuta nel certificato. Fondamentali sono anche gli ultimi due elementi, in quanto rappresentano la firma del certificato apposta dalla CA e l'algoritmo con cui è stata calcolata.

Per funzionare, il meccanismo dei certificati a chiave pubblica necessita di un sistema per poter verificare che essi siano validi. Ci sono due casi in cui un certificato potrebbe essere non più valido. Ogni certificato ha un campo di validità che indica un periodo temporale in cui può essere utilizzato. Dunque, la prima cosa da fare è confrontare la data attuale con le date di validità e, nel caso in cui la data attuale non sia contenuta ne periodo di validità, bisogna scartare il certificato in quanto non è valido. Il secondo caso in cui un certificato non è valido è quello in cui sia stato revocato dalla Certification Authority che lo ha generato. Ci sono diversi motivi per cui una CA 1 invalidare un certificato, ad esempio, se un utente ha richiesto un nuovo certificato perché qualcuno ha rubato la sua "vecchia"chiave pubblica, oppure perché è terminato il periodo di validità del certificato. Per gestire queste situazioni le CA mantengono una Certificate Revocation List CRL in cui vengono elencati tutti i certificati non più validi a partire da una certa data.

Il contenuto dei certificati X.509 viene codificato usando lo standard Distinguished Encoding Rules (DER), che a sua volta rispetta i vincoli imposti da Basic Encoding Rules (BER), due standard che descrivono un metodo indipendente dalla piattaforma per codificare oggetti; entrambi gli standard fanno parte della notazione ASN1. Lo standard DER viene utilizzato in crittografia, per codificare dei dati che devono essere firmati digitalmente. Questa codifica stata introdotta, accanto a BER, in quanto, necessario che i dati da firmare producano un'unica possibile rappresentazione seriale. I certificati X.509 codificati in questo modo hanno estensione .cer o .der. È possibile trovare ulteriori informazioni riguardo ai due standard, sul sito [2].

Questo progetto sfrutta la crittografia a chiave pubblica e il sistema dei certificati X.509 per poter generare e firmare i documenti; prima di procedere, dunque, sarà necessario parlare di come si realizza la firma digitale.

2.1.2 Schema di Firma

Dopo aver introdotto velocemente i principi della crittografia a chiave asimmetrica, possiamo parlare delle basi della firma digitale e degli algoritmi con cui viene

implementata. Come già detto in precedenza, la firma digitale è un sistema di autenticazione che lega un documento ad un'identità in modo sicuro. Vedremo ora in dettaglio come è possibile realizzare un meccanismo di questo tipo.

Per prima cosa dobbiamo definire il concetto di funzione hash sui cui si basa l'implementazione della firma digitale. Con questo termine si identifica una funzione che prende in input un messaggio di lunghezza qualsiasi e restituisce un digest del messaggio di dimensione fissa; questo modo si ottiene una specie di impronta che dipende molto profondamente dal contenuto del messaggio. La funzione hash H per essere definita "buona"deve essere scelta in modo che:

- sia possibile calcolare H a partire da messaggi di qualsiasi lunghezza
- H restituisca un valore di lunghezza fissa per qualsiasi messaggio
- dato un valore h, sia impossibile trovare un messaggio x tale che $h = H(x)$ (*one-way property*)
- dato un messaggio x sia impossibile trovare un altro messaggio y tale che $H(x) = H(y)$ (*weak collision resistance*)
- sia impossibile trovare due messaggi diversi x, y tale che $H(x) = H(y)$ (*strong collision resistance*)

Una buona funzione hash, dunque, può essere utilizzata per riconoscere se il contenuto del messaggio è stato modificato, in quanto anche la modifica di un solo bit del messaggio modifica sensibilmente il valore della funzione di hash.

Ma vediamo come le funzioni hash possono essere utilizzate per crea una firma digitale. La figura 2.1 mostra uno schema del processo di firma di un documento e la successiva verifica dell'autenticità. Una volta creato il messaggio M da firmare, l'autore A applica la funzione di hash H scelta e ottiene l'impronta del messaggio h tale che $h = H(M)$. A questo punto usa la sua chiave privata per cifrare il valore h cos ottenuto e crea la firma f da appendere al messaggio originale; si noti che il contenuto del messaggio è in chiaro e solo il digest viene cifrato. Ora A può tranquillamente inviare il messaggio a B, sicuro che esso arriverà intatto e con la certezza che è stato proprio lui a crearlo.

Quando B riceve il messaggio è in grado di capire se porta con se il contenuto originale e se proviene esattamente da A, conoscendo solo la sua chiave pubblica. Per fare le verifiche prende la firma f e la decifra con la chiave pubblica di A

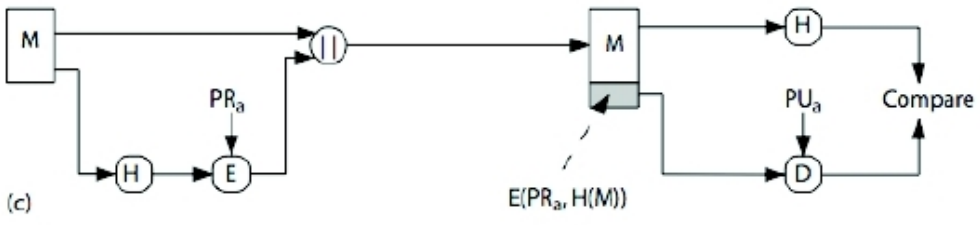

Figura 2.1: Schema di realizzazione e verifica della firma di un documento (tratta da [3] a pagina 64)

ottenendo un valore e. Poi prende il contenuto del messaggio M e applica la stessa funzione hash H che ha scelto A e ottiene un valore $h' = H(M)$. Nell'ultimo passo deve confrontare il valore e e il valore h': se sono uguali significa che il messaggio proviene da A ed il contenuto non è stato modificato, se invece sono diversi significa che c'è qualcosa che non va e il messaggio va scartato.

Secure Hash Algorithm SHA

Uno dei più famosi algoritmi per creare una funzione di hash si chiama Secure Hash Algorithm (SHA) ed è stato inventato da NIST & NSA[1] all'inizio degli anni '90. In realtà SHA è una famiglia di algoritmi di hash che si distinguono principalmente per la lunghezza del digest del messaggio che creano. Il più usato tra gli algoritmi è SHA-1 (quello usato anche per il nostro progetto) il quale crea un'impronta di 160 bit. La limitazione principale è che SHA-1 funziona con messaggi di lunghezza massima $2^{64} - 1$ e che produce un digest piccolo rispetto alle versioni successive della versione SHA-2 (che producono digest di 224, 256, 384 e 512 bit).

Ma vediamo più in dettaglio come funziona l'algoritmo SHA-1. Il messaggio originale viene "riempito"con un padding, per ottenere un oggetto che sia di dimensione in bit pari ad un multiplo di 512. Se il messaggio originale ha già questa proprietà viene comunque aggiunto un blocco di 512 bit. Il padding è composto da un primo bit a 1 seguito da tanti zeri quanti servono. Gli ultimi 64 bit del padding contengono il valore della lunghezza originale del messaggio, che dunque non potrà essere maggiore di $2^{64} - 1$. Tutto ciò viene fatto in quanto l'algoritmo SHA-1 pro-

[1] SHA venne sviluppata a partire dal 1993 dalla National Secutiry Agency (NSA) e pubblicata dal National Institute of Standards and Technology (NIST) come standard (Secure Hash Standard) federale dal governo USA

cessa il messaggio in blocchi da 512 bit. L'hash di 160 bit viene suddivisa in 5 parole da 32 bit, chiamate (a, B, C, D, E) che vengono inizializzate con dei valori definiti dallo standard (vedi [4] per tutti i dettagli) e vengono modificate successivamente dall'algoritmo, in base al contenuto del messaggio per diventarne l'impronta. Ogni blocco di 512 bit del messaggio viene utilizzato per costruire 80 stringhe di 32 bit (5 x 512 bit) di cui le prime 16 sono quelle che compongono il blocco corrente mentre le altre sono composte in modo tale che:

$$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})(\forall i = 16, \ldots, 79) \tag{2.1}$$

In cui $\oplus$ indica l'operatore XOR logico bit a bit; in SHA-1 ogni W_i viene ruotato di un bit verso sinistra per risolvere un problema riscontrato nella prima versione di SHA cioè SHA-0. L'iterazione principale consiste nello scorrere una ad una le 80 parole e ripetere ad ogni passata una serie di operazioni di aggiornamento delle 5 parole (A, B, C, D, E) come riportato in figura 2.2. Il simbolo "enter"indica la rotazione verso sinistra di un numero di bit pari all'indice riportato accanto. Il simbolo (+) indica invece la somma di numeri modulo 2^32. Il valore k_i varia a seconda dell'indice i e può essere definito come:

$$k_i := \begin{cases} \lfloor 2^{30}\sqrt{2} \rfloor & \text{se } i \in [0, 19] \\ \lfloor 2^{30}\sqrt{3} \rfloor & \text{se } i \in [20, 39] \\ \lfloor 2^{30}\sqrt{5} \rfloor & \text{se } i \in [40, 59] \\ \lfloor 2^{30}\sqrt{10} \rfloor & \text{se } i \in [60, 79] \end{cases}$$

I valori W_i sono quelli definiti in precedenza, mentre f_i è funzione dei valori (B, C, D) definita come:

$$f_i(B, C, D) := \begin{cases} (B \wedge C) \vee (\neg B \wedge D) & \text{se } i \in [0, 19] \\ B \oplus C \oplus D) & \text{se } i \in [20, 39] \cup [60, 79] \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{se } i \in [40, 59] \end{cases}$$

La firma digitale, come già detto, viene utilizzata per garantire l'integrità del contenuto del documento e associarlo in modo inconfutabile ad un'identità (solitamente il creatore o possessore del documento). La firma digitale, da sola, non permette di stabilire in nessun modo l'esistenza del documento ad una certa data, in quanto non si fa mai riferimento a meccanismi temporali. Per ovviare a questo problema è possibile aggiungere alla firma una marcatura temporale, per stabilire in modo inconfutabile che il documento è stato firmato in un certo momento.

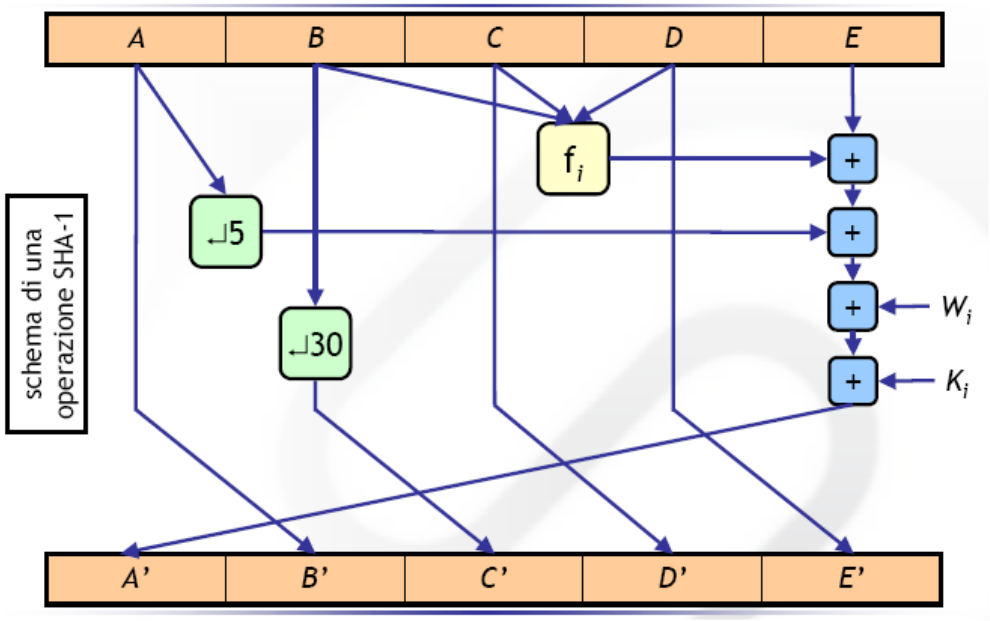

Figura 2.2: Schema di funzionamento di una singola operazione dell'algoritmo SHA1. Immagine tratta da [5]

2.1.3 Marcatura temporale (Timestamp)

Il concetto di timestamp digitale, nasce con lo scopo di definire in modo inconfutabile un istante in cui dei dati sono stati creati, e per garantire che dopo tale istante il contenuto non è stato modificato. Anche per il processo di marcatura temporale è necessaria la presenza di una parte terza fidata, che funge da Time Stamp Authority (TSA), cioè un'entità che crea il timestamp per un certo documento. L'utente che desidera apporre una marcatura temporale ad un documento deve inoltrare una richiesta ad una TSA seguendo le direttive imposte da un Time Stamp Protocol, definito in modo completo e dettagliato in [6].

Si presenta ora il funzionamento di massima del protocollo. Il primo passo consiste nel scegliere una TSA fidata a cui inviare una Time Stamp Request. Una TSR deve contenere le seguenti informazioni:

- *versione* del protocollo TSP (attualmente esiste solo la versione 1)
- *impronta del documento* consiste nella hash del documento e l'identificatore dell'algoritmo con cui è stata calcolata (ad esempio SHA-1)
- *policy* indica con quale prassi deve essere generato il timestamp

- *nonce* un numero random molto grande utilizzato per associare la risposta in modo univoco (opzionale)
- *richiesta certificato* valore booleano che indica se la TSA deve includere il proprio certificato nella risposta
- *estensioni* introdotto per eventuali sviluppi futuri (opzionale)

Quando la TSA riceve una richiesta cerca di interpretarla verificando che sia stata creata in modo corretto. Nel caso la verifica sia andata a buon fine la TSA deve generare il timestamp da includere nella risposta. Il timestamp viene generato estraendo la hash del messaggio, contenuta nella richiesta, a cui viene aggiunto il valore della data e ora attuale, associando in questo modo il contenuto del documento a quell'istante. Per garantire l'autenticità la TSA cifra i byte così ottenuti con la propria chiave pubblica e inserisce questa firma nella risposta. Vediamo più in dettaglio cosa contiene la risposta che verrà inviata al richiedente:

- *stato* indica lo stato della risposta, segnalando se è tutto ok o se ci sono stati degli errori nella richiesta
- *timestamp token* è il timestamp vero e proprio

Un timestamp token viene inserito solo le lo stato della risposta ha un valore 0 o 1, in quanto questi valori indicano che la richiesta era valida e la TSA è stata in grado di generare la marcatura temporale. Se il campo stato contiene valori diversi da 0 o 1, significa che la risposta non contiene il timestamp perché la TSA non è stata in grado di generare la marcatura temporale; in questo caso sono presenti delle informazioni aggiuntive relative a che errore è occorso (sempre nel campo stato). Il campo timestamp, se presente, deve contenere:

- *versione* del protocollo TSP utilizzato per generarlo (attualmente esiste solo la versione 1)
- *policy* indica con quale policy è stato generato
- *impronta del documento* contiene esattamente gli stessi dati del medesimo campo della richiesta
- *numero seriale* indica un numero seriale univoco all'interno della TSA
- *tempo di creazione* rappresenta la data e l'ora fidata di creazione del timestamp in formato Coordinated Universal Time (UTC)

- *accuratezza* indica la sensibilità del campo precedente (opzionale)
- *nonce* contiene lo stesso valore dello rispettivo campo della richiesta (se presente)
- *estensioni* introdotto per eventuali sviluppi futuri (opzionale)

Quando il richiedente riceve la Time Stamp Response può verificarne il contenuto e la correttezza se possiede il certificato (contenente la chiave pubblica) della TSA. Una volta decifrata la risposta, va a leggere, innanzitutto il valore del campo stato per capire se la risposta contiene il timestamp oppure no. In caso affermativo, deve controllare il valore dell'impronta del messaggio e confrontarla con quella che ha spedito. Se i due valori coincidono, significa che il timestamp ricevuto è valido ed è associato al documento giusto.

Una volta ottenuto il timestamp, lo si può "aggiungere" al documento, in base alle proprie necessità certificando che il contenuto attuale era presente in quell'istante e non è stato modificato in momenti successivi. Chiunque voglia verificare l'autenticità del timestamp deve solo procurarsi il certificato della TSA per poterlo decifrare.

2.2 Smart card

Le smart card sono dei dispositivi hardware simili ad una carta di credito che possiedono capacità di calcolo e memorizzazione con un grado molto alto di sicurezza. In pratica sono delle tessere in cui viene impiantato un "microchip" costituito da un insieme di circuiti integrati, microprocessori, memorie, antenne, o altri dispositivi elettronici. Il microchip dispone di un'interfaccia di collegamento che gli permette di collegarsi con degli altri dispositivi elettronici, chiamati lettori di smart card. Esistono due tipi principali di interfacce, cioè a contatto, oppure tramite antenna che permette la letture wireless della tessera.

I primi esemplari di smart card a essere messe in commercio risalgono all'inizio degli anni '80, quando nacquero le prime smart card a microprocessore, utilizzate principalmente come carte di credito bancarie. Alla fine di quel decennio l'International Organization for Standardization (ISO) creò e definì lo standard ISO 7816 che descrive le caratteristiche fisiche, la sicurezza e i comandi per lo scambio di dati con le smart card a contatto. Esiste anche lo standard ISO 14443 che descrive le smart card contactless.

Le smart card hanno avuto uno sviluppo molto rapido, in quanto possono essere utilizzate per numerosissime applicazioni come, ad esempio, la telefonia (SIM Subscriber Identification Module), le banche (carte di credito), i trasporti (in sostituzione degli abbonamenti, solitamente contactless) e infine l'e-government. In questo ultimo ambito, le smart card sono utilizzate per l'identificazione e la memorizzazione di informazioni personali riservate, grazie al loro grado di sicurezza molto alto. Per questo motivo, attualmente, in Italia, le smart card vengono utilizzate per il processo di firma dei documenti con valore legale. Ma vediamo com'è possibile sfruttare le smart card per questo scopo.

Le smart card utilizzate per la firma devono contenere un microprocessore, oltre ad una parte di memoria non volatile (per memorizzare le informazioni riservate), in grado di eseguire tutte le funzioni crittografiche necessarie. Queste funzioni, essendo molto veloci ed efficienti, possono essere calcolate on-board direttamente sulla smart card e realizzate tramite hardware (coprocessore crittografico).

In pratica, al momento della creazione, vengono scritti, nella memoria di tipo ROM (Raed Only Memory) oppure EEPROM (Electrically Erasable and Programmable Read Only Memory) la chiave privata e il certificato (contenente le informazioni sull'identità e la chiave pubblica) del possessore della tessera. Alcune tessere sono in grado, addirittura, di generare automaticamente la coppia di chiavi, in modo che la chiave privata risieda solamente al loro interno e in nessun altro luogo. In ogni caso tutte le smart card implementano dei sistemi di sicurezza in grado di impedire la copia o l'esportazione della chiave privata all'esterno della tessera. Questo sistema però non garantisce la completa sicurezza, in quanto se la smart card viene smarrita, chiunque la può utilizzare. Per questo motivo è stato introdotto un altro meccanismo che permette di aumentare la sicurezza delle smart card, cioè quello del Personal Identification Number (PIN). Per poter accedere alla tessera è necessario conoscere un codice PIN segreto, che funziona come una password. Alla creazione della smart card viene associato un codice casuale, che potrà poi essere sostituito tramite appositi comandi. In questo modo, anche se la tessera viene persa oppure rubata, senza conoscere il PIN non è possibile utilizzarla.

Per poter usare una smart card è necessario possedere un lettore che si interfaccia col il microprocessore inviando richieste di esecuzione dei metodi definiti dall'interfaccia di programmazione della tessera stessa. Oltre al lettore è necessario possedere il software che utilizzi la smart card per la firma. Il processo di firma è molto semplice e avviene, per la parte di cifratura, all'interno della smart card stessa, così da

evitare l'esportazione della chiave privata. Il software deve semplicemente calcolare la hash del documento e inviarla alla smart card che, ricevuto il comando di firma, esegue la cifratura del messaggio e restituisce la hash codificata, cioè la firma del documento.

2.3 Il quadro normativo in Italia

Abbiamo sin qui visto cos'è, come funziona e come si può implementare la firma digitale. Non abbiamo ancora parlato del valore legislativo che viene associato alla firma digitale (se non molto sommariamente). In Italia la firma digitale a crittografia asimmetrica è equiparata (cioè ha lo stesso valore legale) della firma autografa. Il nostro Paese è stato uno dei primi ad aver attribuito pieno valore legale alla firma digitale, già nel 1997 ed è attualmente il paese europeo in cui questa tecnologia è più diffusa.

A partire dal 1997 (con il D.P.R. 10 Novembre 1997 n 513), la legge italiana è stata modificata molte volte per adeguarsi al continuo sviluppo tecnologico e alle normative comunitarie in tema (Direttiva 99/93 in materia di firme elettroniche). Senza dilungarci troppo nella spiegazione di tutte le modifiche avvenute del tempo, vediamo qual è la situazione ad ottobre 2008. L'insieme dei documenti legislativi validi a tale data sono:

- Decreto legislativo 7 marzo 2005, n. 82
- DPCM del 13 gennaio 2004
- Deliberazione CNIPA n.4 del 17 febbraio 2005
- Deliberazione CNIPA n.34 del 18 maggio 2006 con relativo allegato
- Circolare CNIPA n.48 del 6 settembre 2005

È possibile trovare altre informazioni e i link a copie di questi documenti sul sito del Centro Nazionale per l'informatica nella Pubblica Amministrazione (CNIPA), all'indirizzo internet [7] nella sezione relativa alla firma digitale.

I documenti elencati introducono tre importanti termini di cui riportiamo la definizione:

- *Firma elettronica*: L'insieme dei dati in forma elettronica, allegati oppure connessi tramite associazione logica ad altri dati elettronici, utilizzati come metodo di autenticazione informatica

- *Firma elettronica qualificata*: La firma elettronica avanzata che sia basata su un certificato qualificato, creata mediante un dispositivo sicuro per la creazione della firma

- *Firma digitale*: È un particolare tipo di firma elettronica qualificata basata su un sistema di chiavi asimmetriche a coppia, una pubblica e una privata, che consente al titolare tramite la chiave privata e al destinatario tramite la chiave pubblica, rispettivamente, di rendere manifesta e di verificare la provenienza e l'integrità di un documento informatico o di un insieme di documenti informatici

Dunque, la firma digitale con chiave asimmetrica è riconosciuta come strumento valido per la firma e la certificazione dei documenti digitali, a patto che si utilizzino dei sistemi per mantenere segreta in modo affidabile la chiave segreta mentre sia possibile distribuire facilmente la chiave pubblica. Secondo la legge italiana un valido strumento a supporto della protezione della segretezza delle chiavi è costituito dalle smart card protette da PIN. Come visto nella sezione 2.2, esse garantiscono un doppio sistema di protezione e garantiscono il collegamento univoco ad un'identità.

La legge italiana, inoltre, definisce il concetto di *certificatori* come soggetti con particolari requisiti (tra le altre avere un capitale non inferiore a quello per svolgere attività bancarie), i quali mantengono dei registri contenenti le chiavi pubbliche (o i certificati) e altre informazioni sui firmatari dei documenti. È dunque necessario che i certificatori accertino le reali identità dei firmatari prima del rilascio della coppia di chiavi contenuta nelle smartcard. Sul sito del CNIPA è possibile trovare un elenco di certificatori accreditati che svolgono l'attività appena descritta.

La definizione di firma elettronica indica che sono ammessi anche altri sistemi di firma oltre alla firma digitale a chiavi asimmetriche, anche se attualmente questo rimane l'unico sistema realmente implementato e usato. Inoltre viene specificato che la firma digitale

> fa piena prova fino a querela di falso se colui contro il quale la scrittura prodotta ne riconosce la sottoscrizione, ovvero se questa legalmente considerata come riconosciuta (Decreto legislativo 7 marzo 2005, n. 82)

equiparando la firma digitale di documenti informatici alle scritture private sottoscritte con firma autografa. L'unica differenza permane nel modo di disconoscere la firma. Per la firma autografa non è necessario presentare alcune prova del fatto che

la firma non sia autentica, mentre per quella digitale il titolare deve presentare delle prove per disconoscere una firma.

2.3.1 Formati dei documenti firmati

Quanto abbiamo visto finora, riguarda il concetto di firma digitale in generale. Non abbiamo ancora mai parlato di come, dal punto di vista tecnico deve essere realizzata in concreto la firma. In particolare, è utile capire in quale formato devono essere creati i documenti contenenti delle firme. Il documento che identifica i formati riconosciuti in modo ufficiale dalla legge è la *Deliberazione CNIPA n.4 del 17 febbraio 2005*, in particolare l'art 12 titolo IV che parla di formati di firma. Il comma 1 definisce come formato standard per la busta crittografica, cioè formato dei file firmati, lo standard PKCS7 versione 1.5 definito in [8]. I file firmati inoltre devono assumere per legge l'estensione "p7m". La delibera dunque, definisce il formato p7m come unico standard riconosciuto, anche se, negli articoli 8 e 9, si attribuisce il diritto di stabilire (in seguito) nuovi formati standard per la firma digitale.

Nel 2006 il CNIPA ha siglato un protocollo di intesa con Adobe System, che riconosce il formato PDF come formato valido e legalmente riconosciuto per la firma di documenti digitali. Grazie a questo accordo, da quella data è possibile utilizzare il formato PDF, e gli strumenti software relativi anche per firmare i documenti. Non sarà più necessario, dunque, dotarsi di software appositi per gestire i file firmati in formato p7m. Questo è stato reso possibile, in quanto il CNIPA ha riconosciuto al formato i criteri definiti ai commi 8 e 9 della *Deliberazione CNIPA n.4 del 17 febbraio 2005*. Infatti il formato PDF rende disponibili tutte le specifiche tecniche per la realizzazione di software di creazione e verifica della firma e inoltre la Adobe distribuisce in modo gratuito un prodotto di verifica della firma, il famoso Adobe Reader. Questo accordo è risultato da subito molto importante per lo sviluppo della firma digitale, perché alcune pubbliche amministrazioni utilizzavano già il formato PDF come strumento per la distribuzione e archiviazione dei documenti. Inoltre, nello spazio di poco tempo, sono nati numerosi nuovi software (alcuni open source, altri proprietari) che implementano la creazione e la verifica delle firme dei file PDF.

3

Portable Document Format

3.1 Introduzione al formato PDF

Il Portable Document Format, comunemente noto con l'acronimo *PDF*, è un formato di file basato su un linguaggio di descrizione di pagina creato da Adobe per la prima volta nel 1993. A partire da allora sono state definite diverse versioni del formato, fino ad arrivare oggi alla versione PDF 1.4. Questo formato di file è stato concepito per poter distribuire e visualizzare in modo facile ed affidabile documenti digitali indipendentemente dal software e dall'hardware usati per generare e visualizzare il file. Inoltre i file PDF possono contenere immagini e testo a qualsiasi risoluzione.

Il formato PDF eredita ed estende molte delle caratteristiche presenti nel linguaggio di descrizione di pagina *PostScript* creato anch'esso da Adobe. La differenza fondamentale sta nel fatto che PostScript è un vero e proprio linguaggio di programmazione (contenente comandi su come e dove disegnare elementi grafici), che deve quindi essere interpretato, mentre il PDF è solo un formato e non necessita quindi di un interprete. Questo significa che visualizzare o stampare un file PDF consiste nel leggere le descrizioni relative alla costruzione della pagina contenute al suo interno. Il formato PDF mantiene comunque tutte le funzionalità di PostScript, come caratteri, layout e misure.

Il formato PDF è libero, in quanto, anche se Adobe ha numerosi brevetti su di esso, è possibile creare applicazioni che leggano o creino file in formato PDF seguendo quanto scritto in [9] senza pagare le royalties ad Adobe. Prima di vedere e descrivere come viene creato e strutturato un file PDF, è utile vedere quali sono le principali caratteristiche che contraddistinguono i prodotti di Adobe ePaper[1] tra cui anche il formato PDF:

[1] Adobe ePaper Solutions: famiglia di prodotti Adobe che seguono la visione di Adobe del Network Publishing cioè di creazione utilizzo e accesso di contenuti digitali su diverse 0 e dispositivi

- Riproduzione fedele del documento all'interno dell'impresa, indipendentemente da piattaforma, dispositivi e software
- Fusione di contenuti provenienti da diverse sorgenti in un documento indipendente, mantenendo l'integrità delle sorgenti originali
- Modifica interattiva dei documenti da diverse piattaforme e/o luoghi
- *Firme digitali per garantire l'autenticità*
- Sicurezza e permessi per permettere al creatore il controllo dei file e relativi diritti
- Accessibilità dei documenti da parte di persone disabili
- Estrazione e riuso dei contenuti in altri formati e con altre applicazioni

Di seguito si cercherà di 0 com'è strutturato un file PDF. Come si può leggere in [9] un file PDF può essere visto in astratto come una collezione di oggetti che creano il documento PDF, più alcune informazioni strutturali associate, rappresentati come una singola sequenza autonoma di 0.

In pratica una pagina di un documento PDF può contenere una qualsiasi combinazione di testo, elementi grafici e immagini. L'aspetto della pagina viene descritto tramite un PDF *content stream*, il quale contiene una sequenza di *graphic objects* che devono essere disegnati sulla pagina, assieme a tutte le decisioni relative al layout e alla formattazione.

Oltre all'aspetto grafico statico del documento è possibile inserire nel file elementi interattivi come note testuali, markup, link ipertestuali, file allegati, suoni e video. Inoltre si possono aggiungere campi di moduli interattivi da fare compilare all'utente o ad un'applicazione e da cui estrarre valori.

Infine si possono aggiungere tutta una serie di informazioni sulla struttura logica del documento per facilitare ricerche, modifiche e riuso del contenuto.

3.2 Sintassi di un file PDF

Si vuole ora ad analizzare la sintassi di un documento in formato PDF.

Per capire meglio come definita la sintassi del formato PDF è utile vederla suddivisa in 4 sezioni:

- *Oggetti.* Come detto in precedenza un file PDF è composto da una sequenza di 0, che fanno parte di un limitato insieme di tipi fondamentali elencati nella sezione 3.2.1.
- *Struttura del file.* Descrive come gli oggetti sono memorizzati, acceduti e modificati nel file PDF, senza dipendenza dalla semantica dei vari oggetti.
- *Struttura del documento.* Descrive come gli oggetti di base sono utilizzati per creare gli elementi del documento come pagine, font, note, etc.
- *Content streams.* Contengono una sequenza di 0 che indicano come dovrà apparire ogni elemento grafico del documento. Anche se questi elementi fanno parte degli oggetti di un file PDF, data la loro importanza e specificità, verranno descritti in una sezione a parte 3.2.4.

3.2.1 Oggetti

Il formato PDF considera otto tipi di oggetti base che descriveremo nelle sezioni successive. Ogni oggetto può essere etichettato come oggetto indiretto, per creare un identificatore univoco tramite cui ogni altro elemento può referenziarlo. Ogni identificatore si compone di due parti:

- Un numero intero positivo detto identificatore dell'oggetto (spesso questo numero è sequenziale in un file PDF, ma non è necessario).
- Un numero intero non negativo chiamato numero di generazione che avrà valore 0 alla creazione del file per ogni oggetto ma potrà essere aggiornato in modifiche successive del file.

Per definire un oggetto indiretto è necessario descrivere il numero di identificatore, il numero di generazione e poi dichiarare l'oggetto tra le keyword *obj* e *endobj*, ad esempio:

12 0 *obj*
(Ciao)
endobj

Per poter poi referenziare l'oggetto così creato sarà sufficiente indicare i due numeri componenti l'identificatore seguiti dalla keyword *R*. Nel caso in cui il numero di identificatore non esista, non viene generato un errore, semplicemente è come se ci si riferisse all'oggetto null (vedi sezione 3.2.1).

Boolean

Gli oggetti di questo tipo sono rappresentati dalle keyword *true* e *false*. Possono essere usati come elementi di array oppure come voci di un dizionario. Sono ereditati dal PostScript che li 0, ad esempio, come operandi di operatori booleani (*if* o *ifelse*).

Numeri interi e numeri reali

Il formato PDF definisce due tipi di oggetti di tipo numerico, uno per i numeri interi e uno per i numeri reali. Per quanto riguarda il primo tipo saranno convertiti valori del tipo:

4 63 -3 0 +31

Naturalmente il range dei valori è limitato dalla rappresentazione interna dei valori interi e nel caso in cui questo avvenga, il valore viene convertito in un oggetto di tipo reale.

I valori ammessi per tipi reali possono essere ad semepio:

12.3 +7.1 -5.43 7. 0.0 -.005

La rappresentazione interna solitamente del tipo a virgola fissa piuttosto che a virgola mobile, in ogni caso con range e precisione limitati. In questo caso se il valore non compatibile, vieni generato un errore.

Stringhe

Questo tipo è leggermente pi complesso in quanto rappresenta una sequenza di byte (quindi valori compresi tra 0 e 255 anche se non rappresenta un tipo numerico) di lunghezza limitata in base all'implementazione.

Le stringhe possono essere di due tipi in cui varia sia il significato dei valori memorizzati, sia la rappresentazione:

- *Stringhe letterali.* Queste sono le stringhe più comuni, cioè sequenze di caratteri raggruppati tra parentesi (e). Il carattere backslash (\) è viene utilizzato come carattere di escape per poter inserire caratteri non ASCII standard oppure i caratteri riservati come le parentesi, il backslash stesso o caratteri come ritorno a capo o tabulazioni.

- *Stringhe esadecimali.* Queste stringhe vengono scritte come una sequenza di cifre esadecimali (cifre 0-9 e lettere a-z o A-Z). Ogni coppia di caratteri rappresenta un byte della stringa e nel caso i caratteri siano dispari viene aggiunto uno zero in fondo. In queste stringhe i caratteri devono essere racchiusi tra parentesi angolari <e >. Nelle stringhe esadecimali vengono ignorati tutti i caratteri come lo spazio e le tabulazioni. Vengono tipicamente utilizzate per inserire sequenze arbitrarie di dati 2.

Nomi

Gli 0 di tipo nome in PDF rappresentano dei simboli atomici definiti univocamente da una sequenza di caratteri. *Atomici* significa che questo tipo di oggetti non ha nessuna struttura interna e che, due oggetti di tipo nome costruiti con la stessa sequenza di caratteri, rappresentano lo stesso identico oggetto. Tutti i nomi iniziano con lo slash (/) e non possono avere uno spazio o caratteri di tabulazione come primo elemento dopo lo slash. È possibile però inserire un qualsiasi carattere ASCII anche non standard facendo precedere il codice del carattere dal carattere cancelletto (#). Anche per i nomi esiste una limitazione al numero massimo di caratteri in base all'implementazione interna.

Array

Gli Array in PDF, contrariamente a quanto avviene in molti linguaggi, sono di tipo eterogeneo, cioè possono essere formati da una sequenza di oggetti di tipo diverso. Per indicare un oggetto di tipo Array è sufficiente includere i vari valori componenti l'Array tra parentesi quadre ([e]). Il PDF supporta solamente i vettori uni-dimensionali, ma è possibile descrivere matrici o vettori 0 inserendo degli Array come elementi di 08 Array con qualsiasi profondità.

Dizionari

Questi 0 sono delle tabelle associative contenenti coppie di oggetti, chiamate entry del dizionario. Il primo di questi due valori la chiave, che deve essere necessariamente un nome, ed il secondo è il valore, che invece può essere un oggetto di qualsiasi tipo anche un altro dizionario. L'elenco delle coppie chiave-valore di un dizionario deve essere racchiuso tra doppie parentesi angolari (<<e >>). I dizionari sono i blocchi fondamentali che costituiscono i documenti PDF in quanto raggruppano tutti gli

attributi (e relativi valori) di oggetti complessi. Per convenzione è spesso presente una entry del tipo:

<<\Type \Font
...
>>

che indica il tipo di oggetto che viene descritto dal dizionario.

Streams

Come le stringhe, questi oggetti sono delle sequenze di byte, ma a differenza di quest'ultime possono essere letti in modo incrementale; possono avere una lunghezza illimitata e per questo vengono utilizzati per rappresentare elementi come le immagini. Ogni stream è composto da un dizionario che descrive tutta una serie di caratteristiche relative allo stream e dallo stream vero e proprio, incluso tra le keyword *stream* e *endstream*. Ogni stream deve essere un oggetto di tipo indiretto mentre il suo dizionario deve essere di tipo diretto.

Tra le entry del dizionario di uno stream ce ne sono alcune che sono comuni per tutti come ad esempio:

- *Length* di tipo integer. Indica il numero di byte che compongono lo stream. Se lo stream è "cifrato"questo valore rappresenta il numero di byte dello stream codificato.

- *Filter* di tipo name o array. Se lo stream è cifrato in qualche modo questa entry indica in che modo i dati dello stream devono essere trasformati prima di poter essere processati. Se è specificato più di un filtro devono essere inseriti nell'array nell'ordine in cui devono essere applicati.

- *DecodeParam* di tipo dizionario o array. Indicano i parametri dei filtri (specificati nella entry precedente) da applicare ai dati nello stream. Se i valori dei parametri dei filtri sono quelli di default questa entry può essere omessa.

Oggetto null

Questo è un oggetto che ha tipo e valore diversi da ogni altro oggetto. Vieni utilizzato per referenze a oggetti non esistenti e può essere indicato con la keyword *null*.

3.2.2 Struttura del file

Fino ad ora si è visto il significato individuale dei vari oggetti che compongono un file PDF. In questa sezione si cercherà di 0 come questi oggetti sono combinati per l'accesso casuale e l'aggiornamento incrementale dei documenti.

Solitamente un file appena creato si compone di 4 parti fondamentali:

- Un *header* del file, composto da una sola linea, che contiene la versione delle specifiche PDF a cui il file è conforme
- Il *body* (o corpo) che contiene gli oggetti che creano il contenuto del documento
- La *cross-reference table* che contiene i riferimenti agli oggetti indiretti
- Il *trailer* che contiene la posizione della cross-reference table e di alcuni oggetti speciali del corpo del file

Questa struttura permette di accedere facilmente e molto velocemente a tutti gli oggetti del file leggendolo dalla fine (il trailer contiene i riferimenti alla cross-reference table) e permettere la modifica in modo incrementale. Infatti le modifiche che vengono apportate dopo la creazione vengono appese in fondo al file, senza modificare il contenuto precedente. La modifica consiste nell'inserire, dopo l'ultimo trailer, una nuova sezione *body*, una nuova cross-reference table e un nuovo trailer. Il body conterrà solo gli oggetti nuovi oppure quelli che sono stati cambiati rispetto a quelli precedentemente definiti, mentre gli oggetti che sono stati cancellati rimangono invariati e non vengono rimossi. La cross-reference table manterrà le voci solo per gli oggetti che sono stati spostati, modificati o cancellati e non 6 di tutti gli elementi del file. La sezione trailer invece avrà tutte le entry (modificate) del trailer precedente e una voce che punta alla sezione cross-reference precedente.

La tecnica delle modifiche incrementali è molto utile in contesti in cui non è possibile sovrascrivere il contenuto del file di partenza. Questo può accedere, ad esempio, quando si modifica un file tramite una connessione HTTP. Molto più interessante è il discorso che riguarda i file che contengo delle firme digitali. Infatti per non invalidare la firma è necessario non modificare il contenuto del file, per quanto riguarda gli incrementi precedenti.

Un altro aspetto interessante dei file PDF concernente la sicurezza è la possibilità di cifrarne il contenuto. Questa caratteristica permette di impedire accessi non autorizzati a interi file o ad alcune sue parti. Ci sono diversi modi per proteggere

il contenuto di un file, ad esempio tramite cifratura con una chiave, oppure tramite password.

3.2.3 Struttura del documento

Passando ad una visione ancora più astratta di un file PDF si parla di struttura del documento. Un file PDF può essere descritto come una gerarchia di oggetti contenuti nelle varie sezioni body del file (vedi figura). L'oggetto principale, radice della gerarchia, è il *document catalog*, un dizionario che contiene i riferimenti a tutta una serie di altri oggetti che definiscono il contenuto del documento, l'*outline*, e diversi altri attributi. Il contenuto del documento è definito dall'albero delle pagine, una struttura che 3 i riferimenti e definisce l'ordine delle pagine. Ogni pagina infatti è un oggetto di tipo dizionario che contiene i riferimenti al contenuto della pagina e tutti i suoi attributi. La struttura ad albero consente alle applicazioni di visualizzazione dei PDF di poter visualizzare documenti con centinaia di pagine con una quantità limitata di memoria.

3.2.4 Content Stream and Resources

I *contet stream* sono il metodo principale per definire l'aspetto delle pagine e degli altri elementi grafici del documento PDF. Un content stream ha bisogno inoltre di un dizionario delle risorse da cui estrarre tutte le informazioni necessarie. I content stream sono degli oggetti PDF di tipo stream contenenti una sequenza di istruzioni che descrivono come gli elementi grafici devono essere disegnati sulla pagina. A loro volta ogni istruzione è un oggetto PDF, anche se, al contrario di tutti gli altri, non sono visti come oggetti statici ad accesso casuale, ma piuttosto come elementi ad accesso sequenziale. Queste istruzioni non sono altro che sequenze di operandi e operatori (presi dal linguaggio PostScript). Gli operandi sono degli oggetti diretti (non è possibile usare oggetti indiretti o riferimenti) di uno dei tipi base del PDF ad esclusione del tipo stream e del tipo dizionario (che può essere usato solo per pochi operatori particolari). Gli operatori sono invece delle keyword che specificano delle azioni da compiere come, ad esempio, disegnare una forma grafica sulla pagina. Gli operatori sono denotati in notazione post-fissa, cioè ogni operando necessario deve essere presente prima del rispettivo operatore nello stream.

Come detto in precedenza, gli operandi non possono essere degli oggetti indiretti, ma alcuni operatori necessitano di riferirsi ad alcuni oggetti definiti all'esterno del content stream. Per questo motivo è stato introdotto il meccanismo dei dizionari

delle risorse, in cui un content stream definisce tutti gli oggetti esterni a cui deve riferirsi. Dunque ogni dizionario delle risorse ha significato solo nel contesto del relativo content stream.

3.2.5 Strutture dati comuni

Accanto ai tipi di oggetti di base del PDF (descritti nella sezione 3.2.1), esistono alcune strutture dati che sono molto comuni e che sono definite a partire dai tipi di base. Nei paragrafi successivi vengono definite in particolare quelle che sono utili nell'ambito del progetto, cioè: stringhe di testo, date e rettangoli.

Stringhe di testo

Questo tipo si stringhe deriva dal tipo di 0 stringhe, con la sola differenza che rappresentano delle informazioni in un formato human-readable (leggibile dall'uomo). Per trasformare una stringa qualsiasi in una stringa di testo è necessario codificarla in un formato leggibile come Unicode o in un'altro formato di tipo ISO Latin1.

Date

Un'altra struttura dati molto importante è quella che rappresenta le date. Il PDF definisce un formato di data che rispecchia fedelmente le direttive indicate dallo standard internazionale Abstract Syntax Notation One (ASN1). Una data quindi dovrà essere espressa nella seguente notazione:

(D:YYYYMMDDHHmmSSOHH'mm')

in cui:

- *YYYY* rappresenta l'anno
- *MM* rappresenta il mese
- *DD* rappresenta il giorno (01-31)
- *HH* rappresenta l'ora (00-23)
- *mm* rappresenta i muniti (00-59)
- *SS* rappresenta i secondi (00-59)
- *O* rappresenta il rapporto tra l'ora locale rispetto all'Universal Time (UT) e può essere uno dei seguenti caratteri '+', '-', 'Z'

- *HH'* rappresenta l'offset in valore assoluto rispetto al UT in ore (00-23)
- *mm'* rappresenta l'offset in valore assoluto rispetto al UT in minuti (00-59)

Tutti i campi dopo l'anno sono opzionali e, se non espressi assumono il valore 01 per giorno e mese e 0 per i restanti. Il segno (+) del campo O rappresenta che l'ora è in avanti rispetto al UT, il segno (-) invece che è in ritardo, infine il carattere (Z) indica che l'ora locale è uguale al UT; se il campo O non è definito si ritiene che il rapporto con l'UT sia sconosciuto.

Rettangoli

Questo tipo di oggetti viene creato per descrivere zone delle pagine oppure contenitori per altri oggetti. La struttura che li rappresenta è un vettore di 4 elementi in cui si devono specificare le coordinate dei vertici che stanno su una delle diagonali. Comunemente si usa descrivere il rettangolo con il vertice in basso a sinistra e il vertice in alto a destra; dunque la rappresentazione sarà del tipo:

$$\begin{bmatrix} ll_x & ll_y & ur_x & ur_y \end{bmatrix}$$

dove ll sta per lower left (vertice in basso a sinistra) e ur sta per upper right (vertice in alto a destra).

3.3 Caratteristiche interattive

Oltre alle molte caratteristiche "statiche"fin qui descritte, il PDF consente di inserire una serie di caratteristiche interattive all'interno dei documenti. In questo modo è possibile fare interagire l'utente con il documento tramite l'uso di tastiera e mouse. Tra le features messe a disposizione si possono citare:

- *Gestione delle preferenze* per modificare la visualizzazione del documento
- *Aiuti alla navigazione* per spostarsi all'interno del documento
- *Annotazioni* per aggiungere al documento note testuali, suoni o video al documento
- *Azioni* che devono scattare in risposta a particolari eventi
- *Moduli interattivi* (interactive forms) per raccogliere informazioni dall'utente
- *Suoni* da riprodurre tramite l'altoparlante del computer

- *Video* da mostrare sul monitor

Tra queste è di nostro interesse descrivere i moduli interattivi, in quanto vedremo in seguito, che le firme digitali fanno parte di questi oggetti.

3.3.1 Moduli interattivi

I moduli interattivi (in inglese interactive forms oppure AcroFroms) sono una collezione di campi (fields) per raccogliere informazioni in modo interattivo dall'utente. Un documento PDF può contenere un numero qualsiasi di questi campi, sparsi in qualsiasi pagina, che però complessivamente formano un unico e globale module che si estende su tutto il documento.

I vari campi di un form possono essere raggruppati (come avviene per le pagine del documento) in una struttura gerarchica che permette una più facile definizione e nomenclatura di questi oggetti. La gerarchia permette inoltre ai figli di ereditare caratteristiche dai propri predecessori. Un campo figlio può contenere una *widget annotation* che descrive quale sarà l'aspetto del campo sul documento e quali sono le modalità di interazione da parte dell'utente. A sua volta una widget annotation si basa su una o pi *appearence stream* per modificare l'aspetto che l'annotazione assume in base alle varie interazioni a cui può 1. Più precisamente una annotazione può definire non più di tre diversi aspetti:

- *Normale* è l'aspetto di base che l'annotazione assume quando l'utente non sta interagendo con essa; è anche l'aspetto che si utilizza per la stampa dell'annotazione
- *Roll-over* è l'aspetto che ha l'annotazione quando ci si passa sopra con il puntatore senza premere alcun tasto
- *Down* è l'aspetto che assume l'annotazione quando ci si posiziona con il cursore su di essa e viene premuto un tasto

Ogni aspetto può essere definito come un singolo appearence stream oppure come un insieme di appearence streams in base ai diversi stati in cui l'annotazione si può trovare (ad esempio una checkbox che si trova in stato on oppure off).

Tornando ai campi di un form interattivo, come per ogni altro oggetto complesso, essi hanno bisogno di un *field dictionary*, cioè un dizionario del campo, che definisce tutte le caratteristiche necessarie alla visualizzazione e il contenuto del campo. Tra i 0 campi ci sono il nome del campo (che ne identifica il tipo) e il valore (cioè il

contenuto) che varia a seconda del tipo di campo. I tipi principali di campo sono i quattro seguenti:

- *Button fields* sono dei comandi interattivi sullo schermo che l'utente può attivare con il mouse; i principali tipi sono *pushbuttons*, *checkboxes* e *radio button*
- *Text fields* sono dei contenitori in cui l'utente può inserire del testo da tastiera
- *Choice fields* sono dei campi che contengono diversi elementi testuali, di cui al massimo uno può essere selezionato come valore del campo; si dividono in *list boxes* e *compo boxes*
- *Signature fields* sono dei campi che rappresentano firme digitali per autenticare l'identità di un utente; possono contenere semplici funzioni matematiche come digest del documento con chiave pubblica/privata oppure forme biometriche di identificazione

Dunque, per poter firmare un documento PDF bisogna creare e inserire nel documento un campo interattivo di tipo firma, come descriveremo nella sezione successiva.

3.4 Firme digitali

Il formato PDF definisce un tipo particolare di campo interattivo per gestire le firme digitali. Come per tutti gli altri oggetti complessi dei PDF, è necessario definire anche per i campi firma un dizionario, denominato "signature dictionary". Il dizionario deve contenere le seguenti chiavi:

- *Type* indica il tipo del campo, in questo caso deve avere valore *Sig* (opzionale)
- *Filter* rappresenta il nome dell'handler di firma da usare per autenticare il contenuto del campo
- *SubFilter* il nome del metodo specifico (appartenente all'handler specificato in precedenza) da usare (opzionale)
- *ByteRange* un array di coppie del tipo (offset iniziale, lunghezza in byte) per descrivere la posizione di porzioni del digest presenti nel campo contenuto; se il range è unico, significa che il contenuto del campo è una firma, nel caso

in cui siano descritti range discontinui, il contenuto non è un digest che non contiene la firma

- *Contents* la firma cifrata
- *Name* il nome della persona o dell'ente che ha firmato il documento
- *M* la data della firma; può essere un timestamp semplice o acquisito da un timestamp server (opzionale)
- *Location* il luogo fisico dove è stata generata la firma (opzionale)
- *Reason* il motivo per cui è stato firmato il documento (opzionale)

Come ogni altro campo, anche la firma può essere descritta tramite una *widget annotation* con il relativo dizionario, che contiene entries relative all'annotazione e al campo. Tra di esse molto importanti sono quelle che descrivono il posizionamento e l'aspetto del campo firma. Ad esempio, è possibile associare all'annotazione un rettangolo che ne descrive, però, solo la posizione all'interno della pagina. È inoltre possibile decidere se rendere visibile oppure no il campo della firma, impostando valori nulli per altezza e larghezza del rettangolo nel secondo caso. Oltre alla posizione all'interno della pagina è possibile impostare l'aspetto che assume il campo firma all'interno del documento (che non deve essere necessariamente un rettangolo). Questa caratteristica viene gestita modificando opportunamente il dizionario della widget annotation e in particolare la voce AP che indica l'appearence dictionary. Solitamente, se un campo firma non è stato ancora riempito, risulterà blank, cioè come se non ci fosse. Nel caso in cui invece la firma sia presente, si presentano due differenti casi a seconda che sia stato definito un handler appropriato al tipo specifico di firma o no. Infatti il formato PDF consente di gestire diverse forme di firma come ad esempio firme basate su firma digitale a chiave asimmetrica, fingerprinting, scan della retina oppure firme autografe. In ogni caso, se non è specificato un handler apposito per la gestione del campo firma, il suo aspetto sarà una rappresentazione standard appropriata come ad esempio del testo per firme a chiave asimmetrica o un'immagine per la firma autografa e per il l'impronta digitale. Se invece è stato definito un handler specifico, il campo firma può apparire in tre diversi aspetti:

- *da validare* quando non è ancora stata verificata la sua validità
- *valida* quando è stata verificata e accettata dall'handler apposito

- *non valida* quando è stata verificata ma respinta dall'handler apposito

Inizialmente, (cioè appena aperto il documento), si assume che la firma sia da validare, appena viene richiesta la verifica all'handler (può avvenire in modo automatico all'apertura, oppure cliccandoci sopra con il mouse), esso modifica l'aspetto del campo a seconda del risultato della verifica. In [9] si consiglia di inserire, in sovraimpressione, un punto interrogativo giallo quando la firma è ancora da validare, una 'X'rossa se non è valida e un simbolo di "visto"verde se è valida.

3.5 Timestamp

Oltre alla firma, il formato PDF supporta anche l'inserimento delle marcature temporali per certificare in modo affidabile l'esistenza del documento dopo un certo istante. Nel nostro caso, però il timestamp non viene fatto in relazione al documento, bensì in relazione alla firma. In questo modo si certifica che il documento è stato firmato in un certo istante di tempo certificato da un'autorità fidata. Rimangono valide comunque, tutte le spiegazioni fatte nella sezione 2.1.3, solo che, invece di inviare l'hash dell'intero documento, inviamo la firma che comunque è delle dimensioni volute, essendo l'hash cifrata del documento.

Anche se il timestamp è relativo alla firma, esso non viene inserito nel documento PDF come attributo firmato (come viene fatto per la firma), ma semplicemente come attributo non firmato, in quanto non è firmato dal firmatario ma dalla TSA. In ogni caso, viene inserito assieme alla firma, all'interno di una busta PKCS7 (formato standard per buste di firma definito in [8]) e referenziato all'entry M del dizionario del campo firma.

4

Analisi e Progetto

In questo capitolo si parlerà di quali sono i principali componenti che formano il progetto e le interazioni tra di essi. Verranno analizzate le caratteristiche del problema e spiegate quali soluzioni sono state effettuate per risolverlo. In questa sezione, inoltre, verranno descritte le principali caratteristiche dei software e 1 utilizzati.

4.1 Analisi del problema

Con il riconoscimento da parte del CNIPA del PDF come formato valido per la firma, sono nati numerosi software che mettono a disposizione strumenti per la creazione e la verifica della firma digitale dei documenti in formato PDF. Questi strumenti innovativi sono andati a sommarsi a tutta una serie di altri software o tools, già esistenti, per la manipolazione dei documenti in formato PDF. Proprio a questo punto è nata l'idea che sta alla base della realizzazione di questo progetto. In primo luogo, la maggior parte dei nuovi software sono stati creati utilizzando le tecnologie pi innovative, e spesso si basano su applicativi con una interfaccia utente grafica. Questo può essere un vantaggio, in quanto semplifica molto l'interazione da parte degli utenti. D'altro canto, questo può essere visto anche come un difetto, soprattutto se si pensa alla Pubblica Amministrazione. Non è pensabile che un dipendente debba interagire con questi software "manualmente", soprattutto se il numero di documenti da certificare è molto grande. Spesso infatti si utilizzano, in questi casi, software a riga di comando (o comunque senza interfaccia utente) con una limitata interazione, che possono essere utilizzati, ad esempio negli script. Il secondo problema (legato comunque al primo) è che probabilmente alcuni enti utilizzavano dei 1 (a riga di comando) per manipolare i documenti PDF. Sarebbe, dunque, molto utile cercare di integrare le nuove funzionalità nei software già esistenti.

4.2 Tools e software utilizzati

Il progetto ha come scopo, come già detto, quello di integrare le funzionalità di firma digitale all'interno di un toolkit a riga di comando che offre già una serie di operazioni di manipolazione dei documenti PDF. In particolare, si è pensato di sfruttare due software che sono open source, quindi distribuiti gratuitamente e con il codice disponibile. Si tratta di:

- *pdftk* un toolkit per la manipolazione dei PDF
- *OpenSignPDF* un'applicazione per la firma dei PDF tramite smart card

Si cercherà di descrivere ora più in dettaglio le caratteristiche di questi due software.

4.2.1 Pdftk

Pdftk è un toolkit open source (scaricabile all'indirizzo [10]) per la manipolazione dei documenti PDF che mette a disposizione una serie di operazioni molto interessanti:

- Merge di due o più documenti PDF
- Suddivisione delle pagine di un in nuovi documenti singoli
- Rotazione di interi documenti o di singole pagine
- Decifrazione di documenti protetti da password
- Cifratura di documenti
- Riempimento di form interattivi
- Generazione di FDF Data Stencil da un form PDF
- Aggiunta di un Background Watermark o Foreground Stamp
- Stampa dei metadati
- Aggiornamento dei metadati del PDF
- Aggiunta di allegati
- Scompattamento degli allegati
- Scomposizione di un documento nelle sue pagine

- Comprimere e de-comprimere Page Streams
- Riparare documenti corrotti (ove possibile)

Questo tool è stato realizzato con interfaccia a riga di comando, per essere utilizzato negli script. Il tool, oltre ad essere molto utile per manipolare i PDF, è molto interessante dal punto di vista del codice. Infatti, è scritto in linguaggio C++, ma sfrutta al suo interno del codice Java opportunamente compilato. Per fare ciò utilizza a sua volta due strumenti software molto interessanti chiamati *gcj* e *gcjh* (di cui si parlerà in maniera più approfondita nella sezione 4.2.2). Grazie a questi software di supporto, *pdftk* può utilizzare le librerie Java *i-Text* (vedi [11] per maggiori informazioni) che includono tutta una serie di classi per gestire in modo semplice i documenti PDF e tutto ciò che li riguarda.

Di seguito cercheremo di capire meglio come funziona pdftk. Si vedrà, inizialmente, come sono strutturate le directory. La cartella principale contiene tre 0:

- **debian**: contiene alcuni file specifici per l'installazione e il funzionamento in sistemi con installata una distribuzione Debian
- **java_ libs**: contiene le librerie java necessarie al funzionamento (in particolare i sorgenti di i-Text)
- **pdftk**: contiene i file sorgente C++, gli header e i Makefile

Si analizza, in seguito, un po più in dettaglio come sono composti i Makefile. Innanzitutto è presente un Makefile diverso per ogni possibile ambiente cioè Windows, MacOSX, Debian, RedHat, ecc. In questi file vengono solo impostate alcune variabili come il nome dei comandi di compilazione e le loro opzioni, in base al specifico ambiente. In ogni caso, ognuno di questi file include un file chiamato Makefile.Base, che è quello che esegue la compilazione vera e propria. In questo file ci sono i comandi per creare tutti gli oggetti necessari a pdftk, per compilare i sorgenti C++, per installare il programma e anche quelli per fare la pulizia (rimuovere i file oggetto). A sua volta questo file richiama il Makefile presente nella cartella java_libs che è quello che si occupa di compilare le librerie java con gcj e gcjh. Per riuscire a compilare tutte le classi, esso non fa altro che richiamare altri Makefile, uno per ogni cartella che contiene file .java o .class dei package delle librerie java. Ognuno dei Makefile richiamati, contiene i comandi di compilazione con gcj veri e propri. In pratica, per ogni file .java:

- dal file .java crea il corrispondente file .o con gcj
- dal file .java crea il corrispondente file .class con gcj
- dal file .class crea il corrispondente file .h con gcjh
- cancella tutti i file .class con rm
- crea una libreria .a contenente tutti i file .o con ar

Il file Makefile.Base utilizzerà, poi, tutte queste librerie .a più i file .h per compilare il sorgente C++ di pdftk.

Si introduce ora, in modo sommario, il funzionamento di pdftk. In pratica il codice di pdftk consiste in una classe chiamata *Tk_Session* che rappresenta una sessione di esecuzione del tool. Questa classe C++ prevede un costruttore che analizza gli argomenti passati sulla riga di comando e aggiorna lo stato interno dell'oggetto che sta creando in base ad essi. In particolare, il costruttore tenta di creare dei PDF Reader, cioè oggetti che permettono di leggere il contenuto di file in formato PDF, a partire dal nome del file passato come parametro. Inoltre, in base al tipo di operazione richiesta, fa avanzare lo stato interno dell'oggetto, riconoscendo eventuali errori nell'ordine o nella sintassi dei parametri digitati. Nel caso in cui i parametri siano stati passati in modo corretto, il costruttore memorizza quali operazioni dovrà eseguire in seguito. Dopo aver richiamato il costruttore, infatti, nel main, se non ci sono errori (relativi, ad esempio, all'apertura dei file di input o al loro numero), viene chiamato il metodo create_output() responsabile di eseguire le operazioni richieste. Questo metodo, assieme al costruttore, è il cuore del codice di pdftk. In pratica consiste in uno switch-case che analizza lo stato dell'oggetto di tipo Tk_Session per capire quali operazioni deve eseguire con quali input e quali output generare. Naturalmente, in questo metodo vengono gestiti tutti gli errori relativi alla gestione dei PDF, segnalandoli opportunamente all'utente.

4.2.2 Gcj e gcjh

Questi software fanno parte della famiglia dei GNU compiler, in particolare sono quelli che servono per compilare codice java. Entrambi sono distribuiti gratuitamente (vedi [12] per ulteriori informazioni) e, usati insieme, permettono di utilizzare del codice Java (sorgente o compilato) all'interno di progetti scritti in C/C++. Gcj permette di compilare sorgenti java per ottenere Java bytecode oppure codice

macchina nativo; inoltre può trasformare java bytecode direttamente in codice macchina nativo. Gcjh invece, permette di generare file header a partire da file di tipo class (in java bytecode). Per poter funzionare, i file così compilati, si appoggiano su un sistema runtime, *libgcj*, che include parte delle librerie java standard, un garbage collector e un interprete di java bytecode. Oltre a contenere il core delle librerie java, libgcj è il collante tra le librerie, il compilatore e il sistema operativo sottostante. Libgcj è stato fuso con GNU Classpath, che provvede la maggior parte delle librerie che formano il core di Java 4 e il supporto per alcune librerie di Java 5.

Gcj sfrutta il concetto di Compiled Native Interface (CNI), che permette di vedere tutti gli oggetti Java, come oggetti C++, e tutte le classi Java come classi C++. Questo può essere fatto in quanto il linguaggio C++ rispecchia tutte le caratteristiche del linguaggio Java, come la presenza di tipi primitivi accanto agli oggetti (linguaggio object-oriented ibrido), la presenza di metodi virtuali, metodi statici e metodi di istanza, ecc. Per poter sfruttare questo sistema, le classi C++ devono includere l'header *gcj/cni.h* più un header per ogni classe che utilizza. Per generare questi header (file .h), si può utilizzare gcjh, che crea questi file a partire dai file in java bytecode (file .class). Nei file sorgenti C++ che utilizzano CNI è possibile utilizzare delle macro o delle funzioni per creare e gestire tutti i tipi primitivi o gli altri oggetti Java. Infatti i tipi primitivi java vengono identificati in CNI con lo stesso nome, preceduto però dal carattere 'j', ad esempio *char* diventa *jchar*. CNI gestisce anche i package java, mappandoli con i namespace. Quindi, per importare un package con le relative classi, basta creare un namespace, ad esempio *import java.lang;* diventa *using namespace java::lang*. È possibile anche creare un namespace che ne contiene degli altri namespace, ad esempio, per tutte le classi Java standard più usate si può definire:

```
namespace java {
        using namespace java::lang;
        using namespace java::io;
        using namespace java::util;
        }
```

Per quanto riguarda gli oggetti (e le interfacce), invece, in CNI vengono gestiti tramite i puntatori; ad esempio java.lang.String diventa un puntatore di tipo *java::lang::String **. CNI supporta anche la creazione e gestione degli array, distinguendo il caso degli array e dei Vector. Per quanto riguarda il primo, si utilizza la sintassi *JArray<tipo>* dove tipo indica uno dei tipi primitivi come jchar, jint, jbyte ... Per quanto riguarda il secondo caso, invece, bisogna creare oggetti con *JvNewObjactArray(jsize, jclass, jobject)* specificando la lunghezza e il tipo di oggetti che il

vettore deve contenere. Queste sono le informazioni base riguardo a gcj e gcjh, che ci servono per spiegare il codice nel capitolo 5; per ulteriori informazioni visitare [12].

4.2.3 OpenSignPDF

Questo software è stato realizzato da Roberto Iacono (sito personale [13]) e permette di firmare documenti in formato PDF utilizzando una smartcard contenente una coppia di chiavi asimmetriche. Anche OpensignPDF è open source e viene distribuito gratuitamente (scaricabile da [14]). È stato scritto in linguaggio Java ed è disponibile per diverse ambienti (Windows, Linux, Mac). Per gestire i file PDF utilizza le librerie Java *i-Text* proprio come accade per pdftk. Per interagire con la smart card ha bisogno di un wrapper, cioè di un modulo software Java che renda disponibili tutte le funzioni primitive (solitamente in C) per la gestione della tessera di tipo PKCS11. A sua volta il wrapper ha bisogno di un modulo nativo (solitamente una libreria condivisa .so oppure .dll) che permette al sistema di collegarsi al lettore della smart card. Tra le varie scelte possibili, OpenSignPDF, ha scelto un wrapper proprietario, quello di IAIK (vedi [15]). L'acronimo IAIK significa Institute for Applied Information Processing and Communication, un organo della Graz University of Technology, che ha sviluppato una Java Cryptography Extension per la gestione dell'interfaccia di programmazione relative alle funzionalità crittografiche di java. Tra gli elementi sviluppati, di nostro interesse è il wrapper pkcs11, il quale, interagento con la libreria del modulo pkcs11 installato su un sistema, permette di utilizzare le funzionalità crittografiche che esso implementa, tramite librerie java. Oltre ai moduli appena elencati, OpenSignPDF si appoggia anche alla libreria log4j (vedi [16]) per creare dei log contenenti tutte le informazioni di debug sull'esecuzione del programma.

Di seguito si analizzeranno le classi più importanti e, senza scendere in tutti i dettagli, il funzionamento di OpenSignPDF. Il cuore del software è composto dalle classi:

- FirmaPdf : rappresenta l'interfaccia utente, gestisce l'input e usa le altre classi per la creazione della firma

- PDFSigner : è la classe che concretamente modifica il documento PDF aggiungendo la firma ed il timestamp

- MyPkcs11 : è la classe che mette a disposizione i metodi per interagire con il wrapper 0 e quindi il modulo di gestione della smart card

- TimeStampClient : è la classe che interagisce con la TSA per ottenere il timestamp token

La classe che è più interessante è la prima, in quanto è quella di cui si dovrà copiare il comportamento. La prima cosa che deve fare è estrarre il certificato (dal file CA.cer) della CA che ha generato il certificato contenuto nella smart card, che servirà alla classe PDFSigner. Il passo successivo consiste nel creare un oggetto di tipo MyPkcs11 per poter instaurare il collegamento con la smart card (metodo *initSession*). A questo punto si deve creare un'istanza di PDFSigner con tutti i parametri necessari letti dai campi dell'interfaccia utente, e richiamare in seguito il metodo *signPDF*. Questo metodo è quello che svolge tutto il lavoro relativo sia alla firma (interfacciandosi con la smart card), sia al timestamp (utilizzando la classe TimeStampClient), sia all'inserimento di questi elementi nel documento PDF (utilizzando le librerie i-Text). In pratica, per aggiungere la firma al documento, si deve creare un campo di una form interattiva di tipo Sign, definendo un opportuno dizionario (come visto in 3.4). Per creare i vari oggetti necessari si utilizzano le classi di i-Text, come ad esempio *PdfName* o *PdfDictionary*. Per quanto riguarda i dati contenuti nel certificato e quelli che devono essere firmati, si utilizza la codifica DER (accennata in 2.1.1), sfruttando le classi del package *sun.security.util*. Essendo lo standard DER parte della più generale notazione ASN1, la gestione di oggetti codificati in questo modo, viene realizzata tramite le classi del package *com.lowagie.bc.asn1*.

4.3 Progetto

In sintesi, il progetto consiste nel modificare il codice C++ di pdftk, utilizzando le classi Java messe a disposizione da OpenSignPDF, per aggiungere la funzionalità di firma dei documenti PDF. La figura 4.1 rappresenta lo schema dei componenti utilizzati.

Nella sezione 4.2.1 sono state descritte quali sono le funzionalità supportate da pdftk in versione originale. Si vedrà di seguito invece, in concreto, qual è il nuovo comportamento che vogliamo dare al tool per quanto riguarda la nuova funzionalità di firma e con che modalità si possono esprimere le varie opzioni. I parametri che si devono impostare per il funzionamento sono:

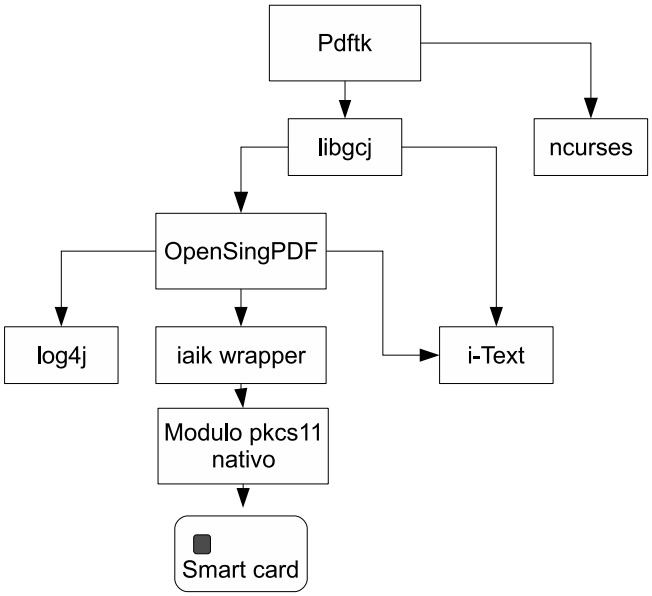

Figura 4.1: Schema dei componenti del progetto

- Percorso del modulo PKCS11 nativo (necessario per il funzionamento del wrapper 0)
- PIN della smart card
- File in cui cercare il certificato della CA che ha rilasciato il certificato contenuto nella smart card
- Nome del file di output
- Posizione della firma nel documento
- Visibilità o meno della firma
- Ragione della firma
- Indirizzo del server di timestamp (che funziona da TSA)
- Nome utente e password per accedere al servizio di timestamp (opzionali)

Per quanto riguarda la posizione della firma, si intende la posizione che assumerà il rettangolo (che rappresenta appunto la firma) all'interno della prima pagina del documento da firmare. Per individuarne la posizione è sufficiente specificare le coordinate di uno dei suoi vertici, nel nostro caso quello in basso a sinistra. Per

quanto riguarda la pagina del PDF, l'angolo in basso a sinistra ha coordinate (0,0), le ascisse sono crescenti da sinistra a destra e le ordinate crescono dal basso verso l'alto.

Si vedrà nel capitolo 5 com'è possibile impostare o modificare questi parametri. In ogni caso, seguendo il funzionamento originale di pdftk, si vorrebbe che, per firmare un documento PDF, si debba scrivere un comando di questo tipo:

pdftk <file-input> sign [<opzione> valore ...] [nome-file-output]

5

Implementazione

Dopo aver introdotto il problema e le basi teoriche necessarie, in questo capitolo si parlerà delle questioni tecniche e di come è stato implementato praticamente il progetto.

Si inizia con il descrivere l'architettura e l'ambiente in cui è stato realizzato il progetto. Infatti, anche se, teoricamente il codice dovrebbe essere portabile, sono stati riscontrati alcuni problemi soprattutto riguardo alla parte di installazione dei componenti software. Data la natura open source dei vari 1 e software da utilizzare si è deciso di sviluppare il progetto in un ambiente Unix/Linux, in particolare la distribuzione Ubuntu release 8.04 (Hardy) con Kernel Linux 2.6.24-19-generic e server grafico GNOME 2.22.3 . Per quanto riguarda i software, sono stati installati in questo sistema le seguenti versioni:

- pdftk versione 1.41
- OpenSignPDF versione 0.0.4
- gcj e g++ parte di GCC versione 4.2.4 (Ubuntu 4.2.4-1ubuntu3)
- gcjh parte di GNU Classpath versione 0.95
- ar (GNU Binutils for Ubuntu) 2.18.0.20080103
- rm (GNU coreutils) 6.10

Oltre alla dotazione software è stato necessario procurarsi una smartcard adatta (cioè con supporto per la firma digitale) e il relativo lettore. Fortunatamente la regione Friuli Venezia Giulia, distribuisce a tutti i cittadini una smart card CNS per la gestione dei dati sanitari. Questa smart card è di tipo adatto, in quanto contiene la coppia di chiavi asimmetriche e il relativo certificato. Dunque, anche se la tessera non è indicata alla firma legalmente riconosciuta, è possibile utilizzarla

per la fase di sviluppo e test del progetto. Inoltre, per semplificare l'accesso ai vari servizi tramite internet, viene distribuito gratuitamente, a chi ne fa richiesta, anche il lettore relativo con interfaccia USB da collegare al computer. Il lettore è un modello ACS ACR38U della bit4id. Per poterlo far funzionare, è necessario installare due ulteriori moduli, cioè:

- opensc versione 0.11.4-svn il modulo nativo per la gestione delle funzioni crittografiche
- pcscd versione 1.4.99 un demone per il rilevamento di lettori di smart card

Una volta installati tutti i software appena descritti, è stato possibili installare e testare il comportamento di pdftk e OpenSignPDF. Verificato il comportamento corretto dei due programmi, è iniziata la fase di modifica del codice. Si può suddividere questa fase in sotto fasi, a seconda del tipo di modifiche apportate ai sorgenti originali e agli altri file del progetto. Verranno presentati i dettagli delle modifiche apportate nelle sezioni 5.1 5.2, 5.3, 5.4.

5.1 Modifiche generali

La prima cosa da fare è quella di inserire nella gerarchia di cartelle di pdftk tutti i file necessari a supportare la nuova funzionalità di firma. In pratica bisogna aggiungere le classi di OpenSignPDF più tutte le librerie di cui necessitano, cioè quelle di log4j e quelle del wrapper IAIK. Questi package vanno tutti copiati nella cartella java_libs dove sono già presenti le librerie i-Text. Per quanto riguarda log4j e il wrapper IAIK, che in OpenSignPDF sono usati come file jar (un archivio contenente l'insieme di tutti file .class necessari), è stato necessario scompattare l'archivio e riprodurre la gerarchia dei package nelle cartelle log4j e iaik. Questa scelta è stata effettuata, non per necessità, in quanto gcj è in grado di compilare anche file in formato jar, ma solo per maggiore semplicità nella gestione delle librerie .a che vengono generati. Inoltre, vedremo nella sezione 5.4 che è stato necessario modificare alcuni makefile all'interno delle cartelle di iaik. Stessa operazione è stata fatta per il package log4j.

Oltre a tutte le classi necessarie per il funzionamento di OpenSignPDF, ci si è accorti, durante le prime compilazioni, che il sistema runtime di gcj, cioè libgcj non contiene tutte le classi che servono per eseguire il codice. Questo avviene non per mancanze di libgcj, ma perché OpenSignPDF utilizza delle classi proprietarie Sun, che, invece, sono incluse in tutte le versioni di jdk. Infatti, compilando il codice

java di OpenSignPDF con il compilatore javac della Sun, si ottengono dei warning riguardo proprio a queste classi che sono proprietarie e non standard, e potrebbero essere sostituite nelle versioni successive di java. Dunque, per fare funzionare tutto il progetto, è necessario procurarsi le classi proprietarie di Sun. Fortunatamente cercando sul web, si è scoperto che è nato un progetto open source chiamato OpenJDK (vedi informazioni su [17]), che rende disponibile il codice sorgente di jdk 6. Tra i sorgenti, dunque, è possibile reperire anche il codice delle classi proprietarie della Sun che servono. Per cercarle in modo più semplice, si può utilizzare un altro sito [18], che permette di trovare tutte o quasi le classi di OpenJDK. Le classi che mancavano e che si devono utilizzare nel progetto sono state scaricate proprio dal sito [18]. In particolare si sta parlando delle classi contenute nella cartella sun e che fanno parte dei package:

- *sun.util.calendar.** insieme delle classi per la gestione di calendari, date, e time zone
- *sun.misc.** insieme di classi per la gestione della codifica base 64 e di altre codifiche
- *sun.nio.ch.** che contiene l'interfaccia DirectBuffer implementata da ...
- *sun.security.action.** insieme di classi da utilizzare per la gestione di un Provider

Si vedrà nella sezione 5.4 come vanno gestite queste classi. Una nota abbastanza importante è che la compilazione di queste classi con gcj genera degli errori di casting che con il compilatore javac Sun non si ottengono. Tutti questi errori riguardano i metodi in cui vengono restituiti degli oggetti (principalmente di tipo byte[], array di byte), creati chiamando il metodo clone(). La soluzione è stata quella di aggiungere, per ognuno di questi errori segnalati, un casting esplicito al tipo adeguato, dell'oggetto restituito dal metodo clone(). Riprenderemo questo argomento nella sezione 5.5.

5.2 Modifiche a pdftk

Le modifiche più importanti, naturalmente, riguardano il codice sorgente C++ di pdftk (pdftk.cc) e del suo header (pdftk.h).

5.2.1 Riconoscimento del comando di firma e opzioni

Per prima cosa è necessario modificare il costruttore della classe Tk_Session, in modo che riconosca il nuovo comando *sign* con la relativa sintassi e le relative opzioni. Per farlo è necessario modificare il file pdftk.h, in particolare il campo *keyword* di tipo enumerazione, che contiene le stringhe che rappresentano le operazioni o le opzioni riconosciute valide tra i parametri passati a pdftk. Accanto a *cat_k*, *burst_k*, *filter_k*, ecc., sonostate aggiunte la stringhe:

- *sign_k* che rappresenta la nuova operazione di firma
- *pin_k* che rappresenta l'opzione per specificare il PIN della smart card come parametro
- *coordx_k* e *coordy_k* che rappresentano le opzioni per modificare le coordinate della posizione della firma nel documento
- *visible_k* che rappresenta l'opzione per rendere invisibile la firma
- *reason_k* che rappresenta l'opzione per specificare una stringa come ragione della firma

Per poter gestire tutte queste opzioni che possono essere specificate tutte, solo alcune, oppure nessuna, è stato necessario specificare una nuovo stato interno dell'oggetto di tipo Tk_Session. Come fatto per le nuove keyword, si è dovuto modificare il campo di tipo enumerazione *ArgState* che contiene tutti gli stati possibili che può attraversare l'oggetto, in base ai parametri passati. Il nuovo valore è stato chiamato *optional_args_e*, per ricordare il fatto che tutte le opzioni del comando firma possono essere omesse.

Oltre a queste modifiche, è necessario dichiarare tutte la nuove variabili necessarie. In particolare c'è bisogno di:

- *string pin* che conterrà il valore del PIN della smart card
- *string reason* che conterrà la stringa che rappresenta il motivo della firma
- *jint coordx, coordy* che conterrà le coordinate del rettangolo che rappresenta la firma nel documento
- *bool visible* che conterrà il valore true se vogliamo che la firma sia visibile, false altrimenti

Dopo aver inserito questi nuovi valori nell'header pdftk.h è possibile modificare la parte di riconoscimento dei parametri da parte del costruttore Tk_Session. Prima di tutto, però, bisogna modificare il metodo *is_keyword* che prende in input una stringa e restituisce uno dei possibili valori dell'enumerazione *keyword*. Se la stringa non è riconosciuta come operazione o opzione, viene restituito il valore *none_k*, altrimenti viene restituito il valore adatto tra gli altri possibili. Ad esempio, per quanto riguarda la firma, si hanno le seguenti corrispondenze:

- la stringa "sign"restituisce il valore *sign_k*
- la stringa "pin"restituisce il valore *pin_k*
- le stringe "coordx"e "x"restituiscono il valore *coordx_k*
- la stringa "coordy"e "y"restituiscono il valore *coordy_k*
- la stringa "invisible"restituisce il valore *visible_k*
- la stringa "reason"restituisce il valore *reason_k*

In questo modo tutte le stringhe descritte verranno riconosciute da pdftk come parametri ammissibili. Questo non è sufficiente, però, in quanto bisogna fare in modo che il costruttore riconosca i parametri nell'ordine corretto. In particolare si vuole che i comandi per la firma rispettino la forma descritta in 4.3. Per quanto riguarda il file di input, è possibile appoggiarsi al funzionamento standard di pdftk, in quanto ogni comando possibile, accetta uno o più file di input come primi parametri. Quindi questo aspetto non comporta nessuna modifica del costruttore. Si deve invece fare in modo che, appena legge la stringa "sign"il programma si ricordi che deve, in seguito, svolgere le funzioni per creare la firma, e aspettarsi una delle opzioni specificate in precedenza. Per fare ciò è sufficiente impostare, una volta letta tale stringa, il valore del campo *m_operation* a *sign_k* e *arg_state* al valore *optional_args_e*. Il costruttore, infatti, contiene uno switch-case anche per quanto riguarda il campo *arg_state*, in modo da riconoscere il corretto ordine negli argomenti. Ad esempio, lo stato iniziale sarà *input_files_e* in quanto il primo parametro è necessariamente un file di input, inoltre il valore *output_filename_e* sarà raggiunto solo quando tutte le operazioni e le opzioni relative sono state specificate, bisogna indicare (o meno, a seconda del comando) i file di output. Per distinguere tra i vari stati è presente uno switch-case sulla variabile *arg_state*. Per far riconoscere il nuovo stato, bisogna aggiungere il

caso in cui la variabile assume il valore *optional_args_e*. Al suo interno sarà necessario controllare il valore che assume la variabile *arg_keyword*, e comportandosi nel seguente modo (vedi listato A.1):

- se il valore è *coordx_k* leggiamo il parametro successivo e lo memorizziamo nel campo *coordx*
- se il valore è *coordy_k* leggiamo il parametro successivo e lo memorizziamo nel campo *coordy*
- se il valore è *pin_k* leggiamo il parametro successivo e lo memorizziamo nel campo *pin*
- se il valore è *visible_k* aggiorniamo al valore *false* il campo *visible*
- se il valore è *reason_k* leggiamo il parametro successivo e lo memorizziamo nel campo *reason*
- se il valore è *output_k* impostiamo la variabile *arg_state* al valore *output_filename_e*
- se il valore non è nessuno dei precedenti, segnaliamo un errore e facciamo fallire l'esecuzione

In questo modo è possibile specificare dei comandi comprendenti tutte le possibili combinazioni dei parametri, con gli unici vincoli che il primo parametro sia il file di input, il secondo sia la parola chiave "sign"e, se specificata, l'opzione "output"seguita del nome del file di output, sia sempre l'ultima.

Si analizzerà ora più in dettaglio il significato dei vari parametri. Le variabili *coordx* e *coordy* rappresentano le coordinate del vertice in basso a sinistra del rettangolo che viene usato per contenere le informazioni testuali relative alla firma. Questo rettangolo (come definito de OpenSignPDF) ha dimensione 220 pixel come larghezza e 100 pixel come altezza. Dunque, la variabile *coordx* avrà valori ammissibili compresi tra 0 e 390, mentre la variabile *coordy* tra 0 e 691, essendo la pagina di dimensioni circa 510 in larghezze e 790 in altezza. Per posizionare il rettangolo in basso a sinistra basta specificare le coordinate (0,0), mentre le coordinate (390,691) per averlo in alto a destra. Come default, la firma viene posizionata in altro ad destra, impostando, nel costruttore, la variabile coordx al valore 390, e la variabile coordy al valore 691.

La variabile *pin* rappresenta il PIN della smartcard utilizzata per la firma. Ci sono due diversi modi per indicare al programma qual è il valore del PIN:

- passandolo come parametro usando l'opzione *pin* seguita da uno spazio e dalla stringa che rappresenta il PIN (in chiaro)

- digitandolo in modo oscurato (se non specificato tra le opzioni)

Il primo metodo, che sembra poco affidabile, è stato inserito pensando all'utilizzo di pdftk all'interno di uno script, evitando così l'interazione da parte dell'utente con pdftk e comunque non renderebbe visibile la stringa che rappresenta il PIN. Dunque, se viene aggiunta l'opzione pin tra i parametri del comando, verrà utilizzata la stringa che segue la keyword. Nel caso in cui, invece, l'opzione pin non è specificata, il programma chiede in modo oscurato l'inserimento del pin (vedi codice del listato A.2). Per fare ciò è stato necessario utilizzare la libreria ncurses, per poter sfruttare la funzione *getch()*. Questa funzione risulta molto utile, in quanto permette di leggere un carattere digitato da tastiera, senza visualizzarne l'echo. In questo modo, è possibile realizzare un comportamento simile a quello che si ha quando si fa un login e si digita una password, visualizzando un carattere '*' per ogni tasto premuto, oscurandone il valore. Sarebbe anche possibile non visualizzare nessun carattere, per non dare informazioni sulla lunghezza del PIN (aumentando quindi il grado di sicurezza), ma attualmente la maggior parte delle smart card supportano PIN di lunghezza pari a 5 o 8 caratteri. Per poter utilizzare le funzioni della libreria *curses* è necessario inizializzare l'ambiente richiamando la funzione *initscr()* prima di utilizzare le funzioni e chiudere l'ambiente chiamando la funzione *endwin()*; è necessario usare anche la funzione *noecho()* che imposta la modalità che non visualizza i caratteri digitati se si utilizza la funzione *getch()*. Il codice che legge il pin è un semplice ciclo while che legge un carattere alla volta e stampa un '*' per ognuno di essi e termina quando il carattere digitato è l'invio. I vari caratteri vengono memorizzati in ordine nella stringa pin.

La variabile booleana *visible* indica se bisogna rendere visibile oppure no il rettangolo che rappresenta la firma. Di default la variabile è impostata al valore *true* (che visualizza il rettangolo). È possibile rendere invisibile la firma specificando l'opzione *invisible* come parametro.

La variabile *reason* di tipo stringa contiene il motivo per cui è stata creata la firma sul documento. Il comportamento di default è quello di impostare, sempre nel costruttore, questa stringa al valore di stringa vuota. Per poter inserire un valore, è sufficiente utilizzare l'opzione *reason* seguita da uno spazio e dalla stringa voluta. È importante specificare che la stringa non può contenere degli spazi, in quanto pdftk

legge i parametri fino al seguente spazio. Per ovviare al problema si possono, ad esempio, utilizzare dei caratteri '_' al posto degli spazi.

L'opzione *output* viene utilizzata per specificare il nome che deve avere il file firmato. Se l'opzione non viene specificata, il programma crea il file firmato con il nome legato al file di input. In particolare prende il nome del file di input, toglie l'estensione "pdf"e il punto, appendendo la stringa "_signed.pdf". Se il file di input si chiama *esempio.pdf*, il file firmato avrà nome *esempio_signed.pdf*.

Per quanto riguarda gli altri parametri che possono essere specificati, ètato deciso di creare un file di configurazione in cui specificarli (vedi sezione 5.2.2).

Come visto nella sezione 4.2.1, il cuore del programma è la funzione create_output(). In questo metodo infatti, vengono realizzate tutte le operazioni per creare l'output corretto. Anche la firma del documento dovrà essere implementata proprio in questo metodo. Per farlo, sarà necessario aggiungere un case allo switch della variabile *m_operation* già presente, per fare riconoscere il nuovo valore *sign_k*. All'interno di questo caso, saranno inserite le operazioni descritte nella sezione 5.2.3.

Le altre modifiche necessarie, ma forse meno interessanti sono quelle relative al manuale di pdftk che si ottiene con il comandi

pdftk –help oppure pdftk -h

Sono state inserite delle voci per descrivere la nuova funzionalità di firma e tutte le opzioni che si possono specificare.

5.2.2 File di configurazione

Questo file è stato creato principalmente per definire il percorso del modulo nativo PKCS11, senza specificarlo come parametro. Infatti, per poter funzionare, il wrapper pkcs11 IAIK ha bisogno di utilizzare le funzioni definite all'interno del modulo nativo. Per quanto riguarda OpenSignPDF, il percorso viene chiesto all'utente come parametro. Per motivi di facilità d'uso del comando e per non aumentare ulteriormente il numero di opzioni da specificare, è stato deciso di introdurre un file di configurazione (a cui gli utenti dei sistemi Linux sono molto familiari) facilmente modificabile. Una volta definito, si è deciso di utilizzarlo anche per altre opzioni come i parametri necessari per la funzione di timestamp e il file in cui cercare il certificato della CA che ha rilasciato il certificato da usare per la firma. La sintassi del file è molto semplice e composta da soli due elementi:

- commenti

- parametri e rispettivi valori

I commenti vengono specificati dal carattere '# '. Tutto il testo che segue tale carattere, viene ignorato. Per quanto riguarda i parametri, nel file di configurazione si devono specificare nel seguente modo:

<nome-parametro >= <valore >

I parametri che sono supportati sono:

- *cryptoki* : percorso del modulo pkcs11 nativo (libreria)
- *certificate* : percorso del file contenente il certificato della CA che ha firmato il certificato da usare per la firma
- *serverTimestamp* : indirizzo del server che funge da TSA
- *usernameTimestamp* : nome utente dell'account per accedere al servizio di timestamp
- *passwordTimestamp* : password dell'account per accedere al servizio di timestamp

Di questi, solo i primi due sono indispensabili al funzionamento corretto di pdftk. Per quanto riguarda il servizio di timestamp, esso non è indispensabile, ma fortemente consigliato, in quanto certifica l'istante della firma. Nella sezione 5.3.1 parleremo in modo approfondito del funzionamento e dei problemi riscontrati relativamente a questa funzionalità.

Per gestire questo file, sono state scritte le seguenti funzioni:

- *static string eliminate_spaces(string)* che prende in input una stringa e la restituisce eliminando tutti gli spazi che contiene (listato A.3 a pag64)
- *static bool check_key(string, char*)* prende in input una stringa e un vettore di caratteri e restituisce *true* se la stringa contiene quel vettore di caratteri eventualmente seguita dal carattere '=' (listato A.4 a pag 64)
- *static int check_line(string)* prende in input una stringa e verifica, utilizzando la funzione *check_key* se la stringa contiene uno dei parametri supportati, in caso affermativo restituisce il valore intero corrispondente dell'enumerazione *param* (definita in seguito) (listato A.5 a pag 65)

- *static string extract_val(string)* prende in input una stringa e restituisce la sotto stringa che segue il carattere '=' (listato A.6 a pag 65)

Queste funzioni verranno poi utilizzate per leggere i parametri nella sezione di codice dedicata alla creazione della firma. A supporto della funzione *check_line* è stata dichiarata, nel file pdftk.h, la variabile *param* di tipo enumerazione che specifica delle stringhe che rappresentano i parametri riconosciuti nel file di configurazione. Questo viene fatto per avere delle costanti che ricordano i nomi dei parametri che vengono riconosciuti, cioè:

- CRYPTOKI quando viene trovata la stringa "cryptoki"
- CERTIFICATE quando viene trovata la stringa "certificate"
- TIMESTAMP_SERVER quando viene trovata la stringa "serverTimestamp"
- USERNAME_TS quando viene trovata la stringa "usernameTimestamp"
- PASSWORD_TS quando viene trovata la stringa "passwordTimestamp"
- NONE quando non viene riconosciuta nessuna delle stringhe che rappresentano un parametro supportato

Inoltre è stato necessario aggiungere ulteriori variabili alla classe Tk_Session, per memorizzare i valori dei parametri impostati nel file di configurazione, definendole in pdftk.h. In particolare si tratta di:

- *string conf_file* rappresenta il percorso del file di configurazione
- *string pkcs_module* rappresenta il percorso della libreria del modulo pkcs11 nativo
- *string cer* rappresenta il percorso del file in cui trovare il certificato della CA
- *string sT rappresenta l'indirizzo della TSA*
- *string uT rappresenta il nome utente dell'account per il servizio di timestamp*
- *string pT rappresenta la password dell'account per il servizio di timestamp*

Queste variabili non vengono inizializzate nel costruttore in quanto non è possibile definire dei valori di default, soprattutto per quanto riguarda il percorso della libreria nativa del modulo pkcs11.

5.2.3 Creazione della firma

Per realizzare concretamente la firma, è necessario utilizzare i metodi messi a disposizione dalle classi di OpenSignPDF. In pratica, si può riprodurre il codice della classe FirmaPdf (adattandolo all'ambiente C++), che è quella che rappresenta l'interfaccia utente di OpenSignPDF. In particolare, basta copiare il comportamento del metodo relativo alla gestione dell'evento pressione del bottone "firma"(metodo *public void actionPerform(ActionEvent evento)*).

Prima di creare la firma, è necessario verificare che siano disponibili in modo corretto il file di input e le informazioni relative al file di output. Per controllare il primo aspetto è sufficiente controllare che sia stato aperto un solo file di input, verificando che vettore di tipo *InputPdf* chiamato *m_input_pdf* abbia dimensione uno. In caso contrario viene segnalato un errore stampando una stringa opportuna. Come seconda cosa bisogna creare un oggetto di tipo *PdfReader*, relativo all'unico file di input specificato, che permette di leggere il contenuto del documento PDF. L'operazione successiva consiste nel verificare se è stato specificata l'opzione *output* per definire il nome del file di che conterrà la firma. Se la stringa *m_output_filename* è vuota, significa che non è stato specificato il nome del file di output e bisogna implementare il comportamento standard. Si tolgono gli ultimi 4 caratteri (che saranno ".pdf") e si appende il suffisso "_signed.pdf". Questo significa che il file firmato creato, se non si specifica un nome per il file, viene creato nella stessa cartella del file originale. Con questi controlli, siamo sicuri di avere tutte le informazioni necessarie relativamente ai file di input e ai file di output.

L'operazione successiva consiste nell'andare a leggere i parametri definiti all'interno del file di configurazione (vedi listato A.7 a pagina 65). Dopo aver aperto il file, segnalando e gestendo gli eventuali errori, è necessario implementare un ciclo per leggerne il contenuto una riga alla volta; il ciclo si interrompe non appena viene letto nel file il carattere 'EOF'. Per ogni riga, bisogna leggere e memorizzare tutti i caratteri in una stringa; non appena viene letto il carattere '# ', si ignora tutto quello che segue fino a quando non si incontra il carattere di fine riga (ASCII 10). Appena raggiunto il fine riga, si eliminano gli spazi dalla stringa ottenuta e si passa questa stringa alla funzione *check_line* per verificare se la riga letta contiene la stringa relativa ad uno dei parametri. Si fa dunque uno switch sul valore restituito dal metodo, confrontandolo con i valori definiti nella struttura *param* e, se c'è una corrispondenza, si setta il relativo parametro con il valore restituito dalla funzione *extract_val* applicato alla riga appena letta. In pratica lo switch si può riassumere

così:

- case CRYPTOKI: pkcs_module = extract_val(opt)
- case CERTIFICATE: cer = extract_val(opt)
- case SERVER_TIMESTAMP: sT = extract_val(opt)
- case USERNAME_TIMESTAMP: uT = extract_val(opt)
- case PASSWORD_TIMESTMAP: pT = extract_val(opt)
- default:

Una volta letti i parametri dal file di configurazione dobbiamo controllare se, tra i parametri con cui è stato chiamato pdftk, è stato specificato il PIN della smart card. Nel caso in cui la variabile pin sia uguale alla stringa vuota, bisogna richiedere all'utente in modo oscurato di digitare il PIN della tessera (come visto in precedenza).

Arrivati a questo punto, siamo sicuri di aver raccolto tutte le informazioni necessarie per poter firmare il documento PDF. Ora, dobbiamo solo "copiare"il comportamento della classe FirmaPdf, che abbiamo descritto in 4.2.3. Questa parte di codice è illustrata nel listato A.8. Prima di farlo, bisogna "trasformare"tutti gli oggetti di C++, in oggetti java, in particolare per quanto riguarda le stringhe. Infatti tutti i metodi che bisogna richiamare, fanno parte delle classi Java di OpenSignPDF, compilate con gcj. Ad esempio, per quanto riguarda le stringhe, quelle utilizzate nel codice fino a questo momento sono di tipo *std::string*, mentre i metodi CNI si aspettano dei parametri di tipo *java::lang::String*. C'è dunque bisogno di creare tutta una serie di nuove stringhe di questo tipo, utilizzando la funzione *JvNewStringLatin1(stringa.c_str())*. Bisogna inoltre creare gli altri parametri che ci servono, tra cui un oggetto di tipo *java::io::File* che rappresenta il file PDF di input, uno di tipo *java::io::FileOutputStream* e uno di tipo *java::io::BufferedOutputStream* per gestire il file PDF di output. Tra i parametri di cui c'è bisogno ci sarebbe anche un array di oggetti di tipo *java::security::cert::X509Certificate*, cioè oggetti che rappresentano dei certificato di tipo X.509. Come si vedrà nella sezione 5.3 è stato utilizzato, un piccolo artefatto per poter disporre di un array di questo tipo. Il resto del codice di questa parte, si comporta proprio come la classe FirmaPDF. Infatti crea un oggetto di tipo *sign::pkcs11::MyPkcs11* e ne richiama il metodo *initSession* passando il percorso del modulo pkcs11 nativo e il PIN della smart card. L'unica

differenza da notare sta nel fatto che, siccome CNI tratta gli oggetti Java tramite dei puntatori, l'invocazione dei metodi non sarà fatta con istanza.metodo(...) ma con l'operatore freccia, cioè istanza->metodo(...). Segue la costruzione di un oggetto di tipo *sign::PDFSigner* con gli opportuni parametri, per poter richiamare in seguito il metodo *signPDF* che è stato modificato (vedi sezione 5.3). Questo metodo si occupa di firmare e apporre la marcatura temporale al file, proprio come succedeva nella versione originale. Le operazioni finali consistono nella chiusura della comunicazione con il modulo di gestione della smart card e dei file e buffer aperti. L'ultima istruzione rappresenta la stampa di un messaggio di operazione di firma riuscita. La gestione degli errori infatti, viene fatta tramite la gestione delle eccezioni, già all'interno dei metodi delle classi Java di OpenSignPDF. Inoltre, anche il metodo *create_output()* di pdftk, implementa un costrutto di tipo try-catch per catturare eventuali eccezioni non gestite tramite il codice delle classi Java.

5.3 Modifiche alle classi di OpenSignPDF

Oltre alle modifiche del codice sorgente C++ di pdftk, è stato necessario apportare delle piccole modifiche al codice Java delle classi di OpenSignPDF. La prima modifica effettuata è stato modificare il nome del package delle librerie, cambiandolo da *package org.opensignature.opensignpdf* al più semplice *package sign*. Questo comporta creare una cartella di nome *sign* all'interno della cartella *java_libs* di pdftk, copiando al suo interno il contenuto della cartella *org/opensignature/opensignpdf* di OpenSignPDF. Inoltre bisogna modificare il nome del package per ognuno dei file Java contenuti in queste cartelle.

Per quanto riguarda il codice sorgente vero e proprio, grazie all'ottimo supporto per l'utilizzo del codice Java nei sorgenti C++, garantito da gcj, è stato sufficiente modificare solo le classi PDFSigner e TimeStampClient. La modifica della prima è stata necessaria, in quanto la versione 0.0.4 di OpenSignPDF non supporta la possibilità di modificare la posizione della firma all'interno del documento o di renderla invisibile. La classe TimeStampClient, invece, dopo numerose prove, sembrava non funzionare correttamente, e dunque è stata modificata per ottenere un timestamp corretto (vedi sezione 5.3.1). Si è già visto come è possibile modificare il posizionamento della firma all'interno del documento, utilizzando le opportune opzioni da passare a Pdftk. Il punto del codice in cui si rende visibile la firma, sotto forma di rettangolo, è all'interno del metodo signPDF della classe PDFSigner (come si può vedere nella sezione di codice A.9). L'istruzione fondamentale è quella che richia-

ma il metodo *setVisibleSignature* dell'oggetto di tipo *PdfSignatureAppearanceOSP*. Come si può vedere, viene creato un oggetto PDF di tipo rettangolo con le coordinate del vertice in basso a sinistra e quello in alto a destra. La versione originale utilizza dei valori fissi per le coordinate, mentre la versione modificata utilizza due variabili intere che indicano ascissa e ordinata del vertice in basso a sinistra. In base alle dimensioni originali (220x100 pixel), vengono impostate anche le coordinate del vertice in alto a destra (coordx+220, coordy+100). Le due variabili intere devono essere dei parametri della funzione signPDF, e riceve i valori al momento della chiamata all'interno di pdftk. Questi valori sono quelli che assumono le variabili *coordx* e *coordy* descritte nella sezione 5.2. Per poter rendere invisibile la firma è necessario aggiungere al metodo un nuovo parametro, questa volta di tipo booleano. Basterà aggiungere un if su questa variabile fare in modo che l'istruzione *setVisibleSignature* descritta in precedenza venga eseguita solo nel caso in cui la variabile assume valore true. Naturalmente questo parametro sarà associato, in fase di chiamata, alla variabile *visible* descritta nella sezione 5.2.

Le successive modifiche relative al codice java, rappresentano la creazione di una nuova classe, chiamata *MyX509Certificate* e inserita nel package *sign.tools*. Questa nuovo tipo di oggetti è stato definito in quanto ci sono stati dei problemi nella gestione dei vettori tramite CNI. In particolare, il costruttore della classe PDFSigner necessita di un array di certificati X509, estratti dal file CA.cer (o quello impostato nel file di configurazione). Java utilizza un meccanismo particolare per istanziare oggetti di tipo X509Certificate, che in realtà è una classe astratta. Questo meccanismo non è riprodotto da libgcj, quindi è stata definita una classe che contiene come attributo privato un vettore di questi oggetti, delegando a java la gestione dell'array. È sufficiente scrivere un metodo di tipo get, che restituisce il vettore, e richiamarlo nel codice C++ per avere un array gestito in modo corretto.

5.3.1 Modifiche alla classe TimeStampClient

La parte più sostanziosa di modifiche al codice Java, riguarda la classe TimeStampClient, responsabile di interagire con una TSA per acquisire un timestamp da collegare alla firma. La classe in versione originale, mette a disposizione tre diversi metodi per acquisire un timestamp, a seconda del tipo di protocollo di connessione che utilizza il server che l'utente utilizza e del tipo di autenticazione necessaria. In particolare, questa classe mette a disposizione i metodi:

- *getTSResponse* per connettersi ad un server che utilizza il protocollo TCP senza autenticazione dell'utente

- *getHttpTSResponse* per connettersi ad un server che utilizza il protocollo HTTP senza autenticazione dell'utente

- *getAuthenticatedHttpTSResponse* per connettersi ad un server che utilizza il protocollo HTTP e gestisce l'autenticazione degli utenti

Per poter testare il funzionamento di questa classe, è stata utilizzata una serie di indirizzi di server sperimentali per il servizio di timestamp, trovata sul sito [19] . Dopo una lunga serie di tentativi, con connessione a server con protocollo sia TCP, sia HTTP, non è stato possibile ottenere nessun risultato, nel senso che, in fase di verifica con Acrobat Reader, la data della firma risulta provenire dal computer del firmatario. Cercando in rete qualche consiglio sui possibili modi per risolvere il problema, sono stati trovati alcuni esempi di codice di un client che si connette ad un servizio di timestamping. È stato deciso di tentare di modificare, dunque, il codice del metodo *getTSResponse* utilizzando, non una connessione tramite socket, ma direttamente tramite una connessione di tipo HTTP (vedi listato A.11 a pag 70). Inoltre, la versione originale di questo metodo, utilizza un buffer di byte "standard"per creare la richiesta da inviare alla TSA. Questa richiesta deve rispettare il formato standard definito in [6] e spiegato nel capito 2.1.3, utilizzando dunque la codifica DER. La modifica consiste nella definizione della richiesta tramite le opportune classi Der definite nel package *sun.secutiry.util*, che permettono di creare una richiesta timestamp in modo corretto. Tra i campi naturalmente è presente anche un vettore di byte che rappresenta la firma che si sta apponendo al documento. A questo punto è possibile inviare la richiesta al server che funge da TSA, utilizzando un pacchetto HTTP e il metodo POST (come descritto in uno dei vari esempi). A questo punto bisogna attendere la risposta da parte del server, verificando che il pacchetto ricevuto come risposta sia di tipo TimeStampResponse. Se si ottiene una risposta di questo tipo, significa che tutto è andato a buon fine e il pacchetto contiene un token di tipo timestamp. Sarà poi il metodo *signPDF* della classe PDFStamper a utilizzare in modo corretto il timestamp, inserendolo come campo non firmato all'interno del documento.

5.4 Modifiche ai makefile

Come visto nella sezione 4.2.1, la compilazione di pdftk avviene tramite l'uso di diversi makefile, i quali servono, in particolare, per compilare i vari file java in codice macchina nativo. Per poter gestire tutte le nuove classi java aggiunte, è necessario inserire un makefile come quello presente nelle cartelle componenti il package *com.lowagie*, per ogni nuova cartella contenente file con estensione .java o .class. Inoltre è necessario modificare il makefile contenuto nella cartella java_libs, per fare in modo che vengano trovati i nuovi makefile e vengano compilate tutte le nuove classi. Questo non è ancora sufficiente, in quanto, per compilare pdftk abbiamo bisogno di tutti i file oggetto che rappresentano classi utilizzate nel progetto. Dunque, dovremo andare a modificare anche il file Makefile.Base, aggiungendo le librerie (file .a) dei package sign, sun, iaik e log4j, nello stesso modo in cui sono aggiunte le librerie del package com.lowagie.

Bisogna fare una piccola precisazione, in quanto, per alcune cartelle è stato necessario modificare leggermente il makefile e non copiarlo in modo completo. Si tratta di tutte le cartelle che contengo dei file .class già compilati in java bytecode. Infatti, java utilizza una convenzione particolare per creare i file relativi a classi definite all'interno di altre classi. Se, ad esempio, abbiamo una classe A che al suo interno dichiara una classe B, dopo la compilazione verranno creati i file A.class e A$B.class. Il carattere dollaro crea alcuni problemi perché è un carattere speciale all'interno dei makefile. Dunque, per compilare questi file contenenti il dollaro, con gcj e gcjh, invece di usare la variabile $<, abbiamo più semplicemente utilizzato l'espressione regolare *.class che comprende tutti i file con quell'estensione, anche quelli che contengono il carattere '$'.

5.5 Problemi

In questa sezione vengono descritti i principali problemi che sono stati affrontati per la realizzazione del progetto che si sta analizzando.

Il wrapper IAIK rappresenta un punto critico per il settaggio e il funzionamento del progetto OpenSignPDF, in quanto, è necessario includere il percorso della cartella che contiene il package di iaik nel classpath di gcj. Questo perché il wrapper ha bisogno di sapere dove può trovare le classi da istanziare. Anche se esistono delle opzioni di gcj che permettono di modificare il classpath, per quanto riguarda la fase di compilazione, ci si è accorti che non esiste un modo per settare il classpath a

runtime. L'unica soluzione trovata è quella di eseguire un comando per modificare la variabile d'ambiente *CLASSPATH* prima di poter eseguire pdftk. Ma esiste un altro problema che rende opinabile l'utilizzo del wrapper di IAIK. Infatti, per poter funzionare, necessita di una libreria nativa, che è necessario scaricare e installare (ulteriori informazioni su [15]). Dunque, anche se le classi sono distribuite assieme al resto del codice, l'utente del nostro progetto sarà costretto a scaricare ed installare la libreria nativa del wrapper IAIK. Oltretutto, è necessario che il percorso in cui la libreria nativa (un file .so shared object) è stata installata, sia incluso nel library path del sistema. L'unica soluzione funzionante trovata per ovviare a questo problema è quella di modificare il file di configurazione libc.conf aggiungendovi una riga con il percorso in cui la libreria è stata installata. Questa soluzione però è praticabile sole se l'utente ha i privilegi di amministratore.

Per ovviare ai diversi problemi che sono stati incontrati utilizzando il wrapper IAIK, si è cercata una soluzione alternativa. La più plausibile è da subito sembrata quella di utilizzare il wrapper pkcs11 scritto da Sun e distribuito dalla versioni di jdk 1.5. Le operazioni di sostituzione del wrapper e di modifica del codice non avrebbero dovuto essere molto 0, in quanto il funzionamento dei due wrapper è molto simile. Il wrapper pkcs11 della Sun è un Provider di sicurezza e deve quindi essere gestito tramite il meccanismo di gestione della sicurezza di java. Per istanziare un provider esistono due modi diversi, cioè farlo in modo dinamico a runtime, oppure in modo statico tramite un file specifico chiamato java.security. Il problema riscontrato è che è necessario far funzionare questo meccanismo utilizzando gcj e il suo sistema runtime, il quale non supporta tutte le funzionalità del meccanismo di sicurezza di java. Nonostante diversi tentativi, non stato possibile utilizzare il wrapper di Sun congiuntamente con gcj per diversi motivi, tra cui eccezioni di metodi non trovati e file di configurazione non contemplati.

Un'altra parte del progetto che ha causato diversi problemi, è quella relativa alla gestione del timestamp, soprattutto per quanto riguarda la classe TimeStampClient. Il funzionamento della classe risulta difforme da quanto documentato, e anche il codice appare alquanto oscuro. In particolare c'è un test che controlla se la variabile intera *port* è maggiore di zero e in caso affermativo si invoca il metodo *getTSResponse* altrimenti si sceglie, in base al fatto che siano stati indicati un nome utente ed una password oppure no, rispettivamente al metodo *getAutenticatedHttpTSResponse* oppure *getHttpResponse*. I fatto che non ci è chiaro è che questa variabile viene inizializzata al valore 80 (porta per il servizio HTTP) e non viene

mai modificata prima del controllo. Dunque qualsiasi esecuzione, con qualsiasi tipo di server per il timestamp finisce con l'invocazione del primo dei tre metodi. Comunque, anche il codice di questo metodo potrebbe essere migliorato utilizzando le apposite classi messi a disposizione da java per la creazione della richiesta di timestamp, invece di usare un array di byte. Nonostante tutto, probabilmente, il cattivo funzionamento della classe non è dovuto al codice, ma ad un problema del nostro sistema relativo alla gestione delle socket. Infatti, il metodo in versione originale, per gestire le connessioni di tipo TCP puro, crea una comunicazione con il server tramite il meccanismo delle socket. In particolare, il codice genera un'eccezione all'invocazione del metodo *shutdownOutput()*, indicando che la parte di connessione verso l'esterno era già stata chiusa. Non è stato possibile trovare informazioni o soluzioni al riguardo. Per ovviare al problema è stato deciso di creare una soluzione utilizzando una connessione Http, lasciando alle varie classi predefinite la gestione delle socket. Questa soluzione ha permesso di ottenere un buon risultato, in quanto si è riusciti a ricevere la risposta corretta da parte di almeno uno dei server testati, cioè *http://timestamping.edelweb.fr/service/tsp*. Chiaramente questa una limitazione dell'attuale implementazione, che 6 essere sopperita in versioni future.

6
Conclusioni

In questa parte conclusiva della relazione cercheremo di eseguire un'analisi critica del lavoro svolto e dei risultati ottenuti.

Come descritto nei precedenti capitoli del documento, è stato possibile produrre un tool a riga di comando per la generazione di documenti PDF legalmente validi, cioè contenenti una firma digitale e la relativa marcatura temporale. L'attualità e l'utilità del progetto, possono essere comprese meglio se si analizza la situazione relativa alla diffusione del formato PDF e alla pratica della firma digitale. L'Italia è uno dei paesi europei in cui il sistema della firma digitale è più sviluppato a livello europeo, con circa due milioni e mezzo di persone lo utilizzano per la certificazione dei documenti. Oltretutto, il riconoscimento del formato PDF come valido per la firma, ha dato un ulteriore impulso alla diffusione dei sistemi per la generazione di PDF legalmente validi, senza l'utilizzo di un formato particolare (p7m) come avveniva in passato. Il progetto non è rivoluzionario, nel senso che non è il primo che permette di firmare i documenti PDF, ma è uno dei pochi a riga di comando, per permettere l'utilizzo di queste funzionalità negli script e in altri metodi non interattivi.

Un aspetto molto interessante è che la realizzazione del progetto non è dovuta partire da zero, ma si è basata sul "riutilizzo" di codice già scritto e in parte testato. Questo, se da una parte è stato motivo di uno sviluppo abbastanza rapido, d'altro canto ha comportato un maggior sforzo nella fase di studio dei vari software e delle soluzioni migliori per farli interagire.

Inevitabilmente, si sono verificati alcuni problemi nella fase di integrazione dei vari software, soprattutto per quanto riguarda il funzionamento di *gcj*, che è in un certo senso il collante tra il codice java di *OpenSignPDF* e il codice C++ di *pdftk*. Infatti, anche se gcj supporta quasi tutte le funzionalità di jdk, sono stati riscontrati dei problemi soprattutto riguardo alla gestione del classpath e del meccanismi di

sicurezza. Questi problemi si ripercuotono sulla fase di configurazione del software (descritta nell'appendice B), rendendola macchinosa e non adatta ad utenti poco esperti.

Oltre ai problemi di integrazione dei software si sono notate anche delle mancanze, o più che altro delle scelte opinabili fatte dai realizzatori di OpenSignPDF. Ci si riferisce in particolare all'uso del wrapper IAIK, un software proprietario per la gestione del modulo pkcs11, che, anche se distribuibile liberamente, necessita dell'installazione di una libreria apposita. Questo impone a chi voglia utilizzare il progetto a scaricare e installare un software aggiuntivo, causando una forte dipendenza del nostro software da quello di IAIK. La scelta risulta opinabile, in quanto Sun mette a disposizione un proprio wrapper, che non necessita di nessun software aggiuntivo ed è sicuramente testato e documentato. In ogni caso il tentativo di sostituire il wrapper pkcs11 è fallito a causa di una mancanza di *gcj* relativamente alla gestione dei meccanismi di sicurezza di java. Questo aspetto rimane sicuramente una delle parti del software che dovrà essere migliorata.

Un altro aspetto che non è stato realizzato in modo completo è quello relativo all'acquisizione del timestamp. Infatti non è stato possibile far funzionare questa caratteristica a partire dal codice originale di OpenSignPDF. Dopo una lunga serie di prove e modifiche, si è riusciti a far funzionare il software anche per quanto riguarda la funzionalità si apposizione del timestamp della firma, anche se non in modo completo e soddisfacente. Il progetto, infatti, funziona solo utilizzando un server di marcatura con scopi di testing che lavora con protocollo HTTP e senza autenticazione degli utenti. In realtà, i server di marcatura affidabili funzionano a pagamento e con l'autenticazione da parte dell'utente. Anche in questo caso sarà necessaria una successivo miglioramento e sviluppo del codice per ottenere il comportamento corretto.

In conclusione, è possibile dire che il software generato può essere considerato una buona base da cui partire per creare un tool completo e affidabile per la creazione di PDF legalmente validi.

A

Codice sorgente

In questa appendice sono state inserite le sezioni di codice più interessanti.

Listing A.1: Case relativo al riconoscimento delle opzioni del comando sign

```
case optional_args_e : { // pin e output
        if( arg_keyword == coordx_k) {
                ii++;
                coordx = atoi(argv[ii]);
                if( coordx > 390) {
                        coordx = 390;
                } else if(coordx < 0) {
                        coordx = 0;
                }
        } else if( arg_keyword == coordy_k) {
                ii++;
                coordy = atoi(argv[ii]);
                if( coordy > 691) {
                        coordy = 691;
                } else if(coordy < 0) {
                        coordy = 0;
                }
        } else if( arg_keyword == pin_k) {
                ii++;
                card_pin = argv[ii];
        }else if( arg_keyword == visible_k) {
                visible = false;
        }else if( arg_keyword == reason_k) {
                ii++;
                reason = argv[ii];
        } else if( arg_keyword == output_k){
                arg_state = output_filename_e;
        } else { // error
                cerr << "Error: Unexpected keyword for sign operation:
                    " << endl;
                cerr << "       " << argv[ii] << endl;
                cerr << "Exiting." << endl;
                fail_b= true;
        }
}
break;
```

Listing A.2: Sezione di codice che implementa l'inserimento in modo oscurato del PIN della smart card

```
if ( card_pin == "") {
        initscr();
        char ch;
        printw("Enter card pin: ");
        noecho();
        ch = getch();
        while((int)ch != 10){
        card_pin.push_back(ch);
        printw("*");
        ch = getch();
        }
        endwin();
}
```

Listing A.3: Funzione che elimina gli spazi da una stringa

```
static std::string eliminate_spaces(std::string s) {
        std::string ret_s="";
        if( s == "") {
                return "";
        }
        for(int i=0; i<s.length(); i++) {
                if(s.at(i) != ' ') {
                        ret_s.push_back(s.at(i));
                }
        }
        return ret_s;
}
```

Listing A.4: Funzione che verifica la presenza di una particolare keyword in una stringa

```
static bool check_key(std::string s, char* key) {
        char aux[s.length()];
        if(s == "") {
                return false;
        }
        for(int i=0; i<s.length(); i++) {
                if(s.at(i) != '=') {
                        aux[i] = s.at(i);
                } else {
                        aux[i] = '\0';
                        break;
                }
        }
        if( strcmp (aux, key) == 0) {
                return true;
        } else {
                return false;
        }
}
```

Listing A.5: Funzione che controlla quale keyword, tra quelle ammesse, è presente in una stringa

```
static int check_line(std::string s) {
        if(s == "") {
                return false;
        }
        if( check_key(s, "cryptoki") ) {
                return TK_Session::CRYPTOKI;
        }else if( check_key(s, "certificate") ) {
                return TK_Session::CERTIFICATE;
        }else if( check_key(s, "serverTimestamp") ) {
                return TK_Session::TIMESTAMP_SERVER;
        }else if( check_key(s, "usernameTimestamp") ) {
                return TK_Session::USERNAME_TS;
        }else if( check_key(s, "passwordTimestamp") ) {
                return TK_Session::PASSWORD_TS;
        } else {
                return TK_Session::NONE;
        }
}
```

Listing A.6: Funzione che estrae la sottostringa che segue il carattere '=' in una stringa

```
static std::string extract_val(std::string s) {
        std::string aux;
        int i=0;
        if(s == "") {
                return "";
        }
        while(s.at(i) != '=') {
                i++;
        }
        i++;
        while(i < s.length()) {
                if((int)s.at(i) == 10){
                        break;
                }
                aux.push_back(s.at(i));
                i++;
        }
        aux.push_back('\0');
        return aux;
}
```

Listing A.7: Sezione di codice che legge e interpreta il file di configurazione di pdftk

```
ifstream ifs( conf_file.c_str() , ifstream::in );
if( ! ifs ) {
        cout << "Impossibile aprire il file " << conf_file <<" !!
            Aborted!! ";
        return;
        }
char ch;
```

```
std::string opt;
while (ifs.good()) {
        ch = (char) ifs.get();
        if(ch == '#') {
                while((int) ch != 10) {
                        ch = (char) ifs.get();
                }
        } else {
                opt.push_back(ch);
        }
        if( (int) ch == EOF ){
                break;
        } else if( (int) ch == 10 ){
                opt = eliminate_spaces(opt);
                switch( check_line( opt ) ){
                        case CRYPTOKI:
                                pkcs_module = extract_val(opt);
                                break;
                        case CERTIFICATE:
                                cer = extract_val(opt);
                                break;
                        case TIMESTAMP_SERVER:
                                sT = extract_val(opt);
                                break;
                        case USERNAME_TS:
                                uT = extract_val(opt);
                                break;
                        case PASSWORD_TS:
                                pT = extract_val(opt);
                                break;
                        default:
                                opt = "";
                        }
                opt = "";
                }
        }
ifs.close();
```

Listing A.8: Sezione di codice che crea la firma del documento PDF

```
java::String* cert = JvNewStringLatin1( cer.c_str() );
java::String* serverTimestamp = JvNewStringLatin1( sT.c_str() );
java::String* usernameTimestamp = JvNewStringLatin1( uT.c_str() );
java::String* passwordTimestamp = JvNewStringLatin1( pT.c_str() );
java::String* typeSignatureSelected = JvNewStringLatin1( tSS.c_str() );
java::String* fieldName = JvNewStringLatin1( fN.c_str() );
java::String* openOfficeSelected = JvNewStringLatin1( oOS.c_str() );
java::String* in_file = JvNewStringLatin1( m_input_pdf[0].m_filename.
    c_str() );
java::String* out_file = JvNewStringLatin1( m_output_filename.c_str() )
    ;
java::String* name = JvNewStringLatin1( reason.c_str() );
java::String* module = JvNewStringLatin1( pkcs_module.c_str() );
java::String* pin = JvNewStringLatin1( card_pin.c_str() );

sign::tools::MyX509Certificate* myCert = new sign::tools::
    MyX509Certificate(sign::tools::CertUtil::toX509Certificate( sign::
```

```
    tools::IOUtils::readBytesFromFile(cert)));

 sign::pkcs11::MyPkcs11* mySign = new sign::pkcs11::MyPkcs11();
 Session* session = mySign->initSession(module, pin);

 java::File* filePdf = new java::File(in_file);

//-- Creating the OutputStream
java::FileOutputStream* fout = new java::FileOutputStream(out_file);

java::BufferedOutputStream* stream = new java::BufferedOutputStream(
    fout);

sign::PDFSigner* signer = new sign::PDFSigner(myCert->
    getMyX509CertificateArray(), serverTimestamp, usernameTimestamp,
    passwordTimestamp, typeSignatureSelected, fieldName,
    openOfficeSelected );

signer->signPDF(mySign, session, filePdf, stream, name, coordx, coordy,
     visible);

fout->flush();
fout->close();

session->closeSession();
mySign->finalizeModule();

cout << endl << "The file '" << m_output_filename << "' had been signed
     correctly." << endl;
```

Listing A.9: Sezione di codice del metodo signPDF della classe PDFSigner che rende visibile la firma del documento PDF

```
if(visible) {
    if ("countersigner".equals(typeSignatureSelected)) {
        sap.setCertified(0);
        sap.setVisibleSignature(fieldName);
    }else {
        sap.setCertified(2);
        if (!"".equals(fieldName))
                        sap.setVisibleSignature(fieldName);
        else
                        sap.setVisibleSignature(new com.lowagie.text.
                            Rectangle(coordx, coordy, coordx+220, coordy
                            +100), 1, null);
        }
        }
```

Listing A.10: Metodo getTSResponse della classe TimeStampClient che si occupa di acquisire la marcatura temporale

```
public static byte[] getTSResponse(byte[] hash, URL serverTimestamp) {
        int CONNECT_TIMEOUT = 15000; // 15 seconds
    // The MIME type for a timestamp query
    String TS_QUERY_MIME_TYPE = "application/timestamp-query";
        // The MIME type for a timestamp reply
```

```
String TS_REPLY_MIME_TYPE = "application/timestamp-response";
    boolean DEBUG = false;
    try{
    ObjectIdentifier SHA1_OID;
    ObjectIdentifier MD5_OID;

    ObjectIdentifier sha1 = null;
    ObjectIdentifier md5 = null;
    try {
            sha1 = new ObjectIdentifier("1.3.14.3.2.26");
        md5 = new ObjectIdentifier("1.2.840.113549.2.5");
            } catch (IOException ioe) {
    // should not happen
    }
    SHA1_OID = sha1;
    MD5_OID = md5;

    int version = 1;
    ObjectIdentifier hashAlgorithmId = SHA1_OID;
    String policyId = null;
    BigInteger nonce = null;
    boolean returnCertificate = false;
    DerOutputStream Tsrequest = new DerOutputStream();
    // encode version
    Tsrequest.putInteger(version);
    // encode messageImprint
    DerOutputStream messageImprint = new DerOutputStream();
    DerOutputStream hashAlgorithm = new DerOutputStream();
    hashAlgorithm.putOID(hashAlgorithmId);
    messageImprint.write(DerValue.tag_Sequence, hashAlgorithm);
    messageImprint.putOctetString(hash);
    Tsrequest.write(DerValue.tag_Sequence, messageImprint);
    // encode optional elements
    if (policyId != null) {
        Tsrequest.putOID(new ObjectIdentifier(policyId));
    }
    if (nonce != null) {
        Tsrequest.putInteger(nonce);
    }
    if (returnCertificate) {
        Tsrequest.putBoolean(true);
    }
    DerOutputStream outReq = new DerOutputStream();
    outReq.write(DerValue.tag_Sequence, Tsrequest);
    HttpURLConnection connection = (HttpURLConnection)
        serverTimestamp.openConnection();
    connection.setDoOutput(true);
    connection.setUseCaches(false); // ignore cache
    connection.setRequestProperty("Content-Type",
        TS_QUERY_MIME_TYPE);
    connection.setRequestMethod("POST");
    // Avoids the "hang" when a proxy is required but none has been
        set.
    connection.setConnectTimeout(CONNECT_TIMEOUT);
    if (DEBUG) {
        Set headers = connection.getRequestProperties().entrySet();
```

```
        System.out.println(connection.getRequestMethod() + " " +
            serverTimestamp + " HTTP/1.1");
        for (Iterator i = headers.iterator(); i.hasNext(); ) {
            System.out.println("  " + i.next());
            }
        System.out.println();
            }
    //connection.connect(); // No HTTP authentication is performed
    // Send the request
    DataOutputStream output = null;
    try {
        output = new DataOutputStream(connection.getOutputStream())
            ;
        byte[] request = outReq.toByteArray();
            output.write(request, 0, request.length);
        output.flush();
        if (DEBUG) {
            System.out.println("sent timestamp query (length=" +
                request.length + ")");
        }
    } catch(Exception e) {
        if (output != null) {
            output.close();
        }
        e.printStackTrace();
    }

    // Receive the reply
    BufferedInputStream input = null;
    byte[] replyBuffer = null;
    try {
        input = new BufferedInputStream(connection.getInputStream()
            );
        if (DEBUG) {
            String header = connection.getHeaderField(0);
            System.out.println(header);
            int i = 1;
            while ((header = connection.getHeaderField(i)) != null)
                 {
                String key = connection.getHeaderFieldKey(i);
                System.out.println("  " + ((key==null) ? "" : key +
                    ": ") + header);
                i++;
                    }
            System.out.println();
            }
        int contentLength = connection.getContentLength();
        if (contentLength == -1) {
            contentLength = Integer.MAX_VALUE;
            }
        verifyMimeType(connection.getContentType());

        replyBuffer = new byte[contentLength];
        int total = 0;
        int count = 0;
        while (count != -1 && total < contentLength) {
```

```
                count = input.read(replyBuffer, total, replyBuffer.
                    length - total);
                total += count;
                }
            if (DEBUG) {
                System.out.println("received timestamp response (length
                    =" + replyBuffer.length + ")");
                }
                } catch(Exception e) {
                if (input != null) {
                        input.close();
                }
                e.printStackTrace();
                }

        return replyBuffer;
                }catch(Exception e) {
                        e.printStackTrace();
                        return null;
                        }

        }
```

Listing A.11: Metodo della classe TimeStampClient che si occupa di verificare che la risposta della TSA è di tipo corretto

```
  private static void verifyMimeType(String contentType) throws
      IOException {

        String TS_REPLY_MIME_TYPE ="application/timestamp-reply";

    if (! TS_REPLY_MIME_TYPE.equalsIgnoreCase(contentType)) {
        throw new IOException("MIME Content-Type is not " +
            TS_REPLY_MIME_TYPE + " but " + contentType);
        }
    }
```

B

Installazione e utilizzo PDFTK

Questa appendice, vuole essere una sorta di manuale per l'installazione e l'utilizzo di pdftk per la funzionalit di firma. Vedremo infatti, nella prima sezione, quali sono le componenti software aggiuntive da scaricare e installare e i relativi settaggi. Nella seconda parte invece presenteremo una descrizione delle varie opzioni e alcuni esempi di utilizzo del tool.

B.1 Installazione componenti

Per il corretto funzionamento di pdftk con la nuova funzionalit di firma necessario installare i seguenti componenti software:

- gcj e gcjh versione 4.2 o successiva
- g++ versione 4.2 o successiva
- libreria ncurses
- libreria nativa del wrapper pkcs11 iaik

Solitamente i compilatori gcj, gcjh e g++ sono già installati, nella maggior parte delle distribuzioni Linux. Per quanto riguarda la libreria ncurses, è possibile scaricarla da [20]. Per quanto riguarda il wrapper iaik, abbiamo già ampiamente discusso nella sezione 5.5 sul suo utilizzo, ricordiamo comunque che è possibile scaricarlo da [15]. Come già spiegato, non è sufficiente scaricare e installare la libreria (file libpkcs11wrapper.so), ma è necessario include il percorso di questo file nel library path del sistema, per fare in modo che, il sistema runtime di gcj lo trovi. Per fare ciò, una volta installata la libreria di iaik, è possibile seguire i seguenti passi:

1. Individuare il path assoluto della libreria nativa di iaik appena installata

2. Aggiungere questo path nel file libc.conf che si trova nella cartella /etc/ld.so.conf.d (in Ubuntu 8.04, qualcosa di simile per le altre distribuzioni)

3. Eseguire il comando ldconfig per far riconoscere il percorso appena aggiunto

La procedura appena descritta necessita, ai passi 2 e 3, che l'utente abbia privilegi di amministratore sul sistema.

Per quanto riguarda le classi java del wrapper iaik e di i-Text, non è necessario scaricarle o installarle, in quanto sono distribuite sotto GPL e sono già incluse nelle directory del processo. Naturalmente, per poter utilizzare la smart card e il relativo lettore, è necessario installare il modulo pkcs11 di gestione del dispositivo.

Una volta installati tutti i componenti necessari, è possibile compilare pdftk, entrando nella sotto-cartella *pdftk* del progetto ed eseguire il comando:

```
make -f Makefile.Debian
```

Il progetto di pdftk in versione originale offre il supporto anche per altri sistemi operativi, tra cui Windows, MacOSX, Solaris e altre distribuzioni Linux (RedHat e Mandrake), dunque, in via teorica, basta utilizzare il Makefile opportuno per farlo funzionare nei vari sistemi. Il funzionamento comunque è stato testato solo sul sistema descritto in 5. Il comando descritto, se va a buon fine, crea l'eseguibile di nome *pdftk* nella cartella in cui sono presenti i file sorgenti. Per installare pdftk nella cartella di sistema */usr/local/bin* e rendere disponibile il comando da qualsiasi posizione nel sistema, basta eseguire il comando (se si hanno privilegi di root):

```
sudo make -f Makefile.Debian install
```

A questo punto, è necessario impostare un altro parametro, cioè la variabile d'ambiente CLASSPATH, per permettere al wrapper iaik di trovare le classi java che gli servono. Il percorso da individuare è il path assoluto della directory java_libs del progetto, ad esempio /home/user/tesi/pdftk/java_libs. Una volta individuato il path, bisogna digitare il comando:

```
export CLASSPATH=\$CLASSPATH:path
```

In questo modo il percorso viene appeso in fondo alla variabile d'ambiente CLASSPATH, che contiene l'elenco dei percorsi in cui java e gcj vanno a cercare le classi che non trovano nel loro path standard. Questo comando deve essere eseguito non solo in fase di installazione, ma ogni qualvolta si vuole utilizzare pdftk, in quanto

la variabile CLASSPATH viene resettata ad ogni avvio di una shell. Nella sottocartella pdftk del progetto abbiamo inserito un file chiamato *class.conf* che contiene il comando descritto in precedenza. Sarà sufficiente modificare il percorso indicato dopo i ':' con il proprio path, e poi, ogni volta che si apre una nuova shell, prima di eseguire pdftk per la prima volta, eseguire il comando:

```
. class.conf
```

L'ultimo parametro da settare, riguarda, ancora, il wrapper pkcs11 iaik, che necessita del percorso della libreria del modulo pkcs11 installato sul sistema. Solo impostando questo valore il wrapper sarà in grado di interagire con il lettore della smart card e utilizzare le l'API che mette a disposizione. Per semplificare questa operazione abbiamo creato un file di configurazione, chiamato *pdftk.config* (descritto nella sezione 5.2.2) e posizionato nella stessa cartella dei sorgenti di pdftk. Uno dei parametri che è possibile settare in questo file è proprio il percorso della libreria del modulo pkcs11 nativo (chiamata anche cryptoki), semplicemente modificando il valore del parametro *cryptoki*. Per individuare la libreria corretta bisogna conoscere il nome del modulo che gestisce il lettore della smart card (ad esempio opensc) e cercare nel percorso */usr/lib/nome_del_modulo* (nel nostro caso /usr/lib/opensc). In questa cartella dovrebbe essere presente una libreria condivisa (file con estensione .so) o un link a questa libreria, con un nome simile a *nome_del_modulo-pkcs11.so* oppure *libpkcs11.so* (nel nostro caso opensc-pkcs11.so). Una volta individuata la libreria corretta, basta copiarne il path nel file pdftk.config nella voce cryptoki.

Rimane un'ultima operazione da fare per rendere completo il processo di installazione e configurazione di pdftk per la funzionalità della firma. Consiste nel procurarsi il certificato dell'autorità che ha emesso il certificato contenuto nella smart card. Solitamente questo file è scaricabile dal sito internet dell'autorità di certificazione, e deve avere estensione .cer (altrimenti è possibile convertirlo tramite appositi tool). Una volta scaricato è necessario copiarlo nella cartella del progetto contenete i sorgenti di pdftk ed possibile operare in due modalità per farlo riconoscere al nostro tool:

- rinominare il file con il nome CA.cer
- andare a modificare la voce *certificate* del file di configurazione (vedi la sezione B.2)

Seguendo passo passo queste istruzioni, dovrebbe essere possibile eseguire in modo corretto i comandi descritti nella sezione B.2

B.2 Utilizzo

In questa sezione descriveremo il significato delle opzioni e una serie di esempi di utilizzo della nuova funzionalità di firma di pdftk. Una volta installato il tool, seguendo le istruzioni indicate nella sezione install, sarà possibile utilizzare pdftk per tutte le sue funzioni di base, più la nuova funzionalità di firma. Per quanto riguarda i comandi standard, è possibile ottenere supporto visitando il sito [10], oppure digitando il comando:

```
pdftk -h
```

Vediamo invece come si può utilizzare il nuovo comando sign. Naturalmente, prima di eseguire il comando, bisogna collegare il lettore al PC ed inserire la smart card. Il comando base segue la seguente sintassi:

```
pdftk <input_pdf_file> sign [<option>  val] [<output_pdf_file>]
```

Dalla sintassi, si può capire che, le uniche parti indispensabili sono un unico file di input, e il comando sign; tutto il resto è opzionale, compreso il nome del file di output. Ma vediamo come si specificano e che significato hanno le varie opzioni :

- pin val : indica il PIN da utilizzare per accedere alla smart card. Se non viene specificato, il PIN viene richiesto in modo oscurato all'utente.
- invisible : Se presente impone che la firma non sia visibile nel documento pdf, cioè non venga creato il rettangolo contenente tutti i dati; in ogni caso il pdf contiene la firma ed è possibile vederne tutte le informazioni in un apposito menù del tool di visualizzazione. Se non specificato, la firma di default viene visualizzata.
- coordx val : modifica l'ascissa del vertice in basso a sinistra del rettangolo che rappresenta la firma; val deve essere un numero compreso tra 0 e 390. Se n on specificato il valore di default è il massimo ammesso cioè 390
- coordy val : modifica l'ordinata del vertice in basso a sinistra del rettangolo che rappresenta la firma; val deve essere un numero compreso tra 0 e 691. Se n on specificato il valore di default è il massimo ammesso cioè 691
- reason stringa : modifica la ragione della firma, cioè il motivo per cui è stata apposta la firma al documento; la stringa non deve contenere degli spazi, ma

può contenere qualsiasi altro carattere. Se non viene specificata, non viene inserita nessuna ragione della firma (stringa vuota).

- output nome_del_file_pdf : indica il nome che deve avere il file firmato; è consigliabile, ma non strettamente necessario che il file abbia estensione .pdf. Se non specificato il file di output avrà lo stesso nome di quello di input (senza l'estensione .pdf) con l'aggiunta del suffisso *_signed.pdf*.

L'ordine degli elementi indicato dalla sintassi del comando impone che il primo elemento sia uno e un solo file pdf da firmare, il secondo sia la keyword *sign*, seguito da alcune opzioni e solo alla fine l'opzione *output* che, se specificata, deve essere per forza l'ultima. L'ordine delle altre opzioni invece non è importante. Naturalmente l'opzione invisible esclude l'utilizzo delle opzioni coordx e coordy, in quanto rende inutile il posizionamento del rettangolo che non viene creato.

Oltre alle opzioni che si possono esprimere a linea di comando, è possibile modificare anche altri parametri andando a modificare il file *pdftk.config*. In particolare, le voci che si possono modificare sono:

- certificate : indica il nome del file in cui è memorizzato il certificato della Certification Authority che ha emesso il certificato contenuto nella smart card; di default è il file CA.cer presente nella cartella contenente i sorgenti di pdftk
- serverTimestamp : indica l'indirizzo del server che deve essere utilizzato come TSA
- usernemaTimestamp : indica il nome utente dell'account per accedere al servizio di timestamp
- passwordTimestamp : indica la password dell'account per accedere al servizio di timestamp

LA versione finale del progetto è stata testata e fatta funzionare con un solo server di timestamp tra quelli trovati nel sito [19]. Questo è un servizio di timestamp gratuito, senza necessità di autenticazione da parte degli utenti, utilizzato per scopi sperimentali. Le due ultime voci inserite nel file di configurazione sono state mantenute solo per compatibilità con OpenSignPDF, ma non sono attualmente utilizzate dal tool. Il file di configurazione contiene già l'indirizzo del server funzionante.

Vediamo in concreto una serie di esempi di comandi che possono essere utilizzati.

Il comando minimo, che utilizza il comportamento di default è il seguente:

```
pdftk input.pdf sign
```

Questo genererà in output il file input_signed.pdf che contiene la firma visibile posizionata nell'angolo in alto a destra, con nessuna ragione specificata; il pin viene richiesto all'utente in modo interattivo.

```
pdftk input.pdf sign output output.pdf
```

Questo genererà in output il file output.pdf che contiene la firma visibile posizionata nell'angolo in alto a destra, con nessuna ragione specificata; il pin viene richiesto all'utente in modo interattivo.

```
pdftk input.pdf sign reason Certificazione\_contenuto
```

Questo genererà in output il file input_signed.pdf che contiene la firma visibile posizionata nell'angolo in alto a destra, con ragione settata alla stringa "Certificazione_contenuto; il pin viene richiesto all'utente in modo interattivo.

```
pdftk input.pdf sign coordx 0 coordy 0
```

Questo genererà in output il file input_signed.pdf che contiene la firma visibile posizionata nell'angolo in basso a sinistra, con nessuna ragione specificata; il pin viene richiesto all'utente in modo interattivo.

```
pdftk input.pdf sign pin 55555
```

Questo genererà in output il file input_signed.pdf che contiene la firma visibile posizionata nell'angolo in alto a destra, con nessuna ragione specificata.

```
pdftk input.pdf sign invisible output output.pdf
```

Questo genererà in output il file output.pdf che contiene la firma non visibile sul documento, con nessuna ragione specificata; il pin viene richiesto all'utente in modo interattivo.

Bibliografia

[1] R. Housley, RSA Laboratories, W. Polk, NIST, W. Ford, VeriSign, D. Solo, Citigroup, *RFC3820: Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile.*, Aprile 2002 2.1.1

[2] Sito di International Telecomunication Union : *http://www.itu.int/net/home/index.aspx* 2.1.1

[3] Marino Miculan, *Asymmetric Encription and Message Authentication* 2.1

[4] D. Eastlake 3rd, Motorola, P. Jones, Cisco Systems, *RFC3174: US Secure Hash Algorithm 1 (SHA1)*, Settembre 2001 2.1.2

[5] Davide Cerri, *Funzioni di hash sicure: MD5 e SHA-1*, CEFRIEL Politecnico di Milano, http://www.cefriel.it/~cerri/ 2.2

[6] C. Adams, Entrust, P. Cain, BBN, D. Pinkas, Integris, R. Zuccherato, Entrust, *RFC3161: Internet X.509 Public Key Infrastructure Time Stamp Protocol (TSP)*, Agosto 2001 2.1.3, 5.3.1

[7] Sito internet del CNIPA *http://www.cnipa.gov.it* 2.3

[8] B. Kaliski, RSA Laboratories East, *RFC2315: PKCS # 7: Cryptographic Message Syntax Version 1.5* Marzo 1998 2.3.1, 3.5

[9] Adobe System Incorporated, *PDF REFERENCE third edition. Adobe Portable Document Format version 1.4* Addison-Wesley (Dicembre 2001) 3.1, 3.4

[10] Siti internet di PDFTK: *http://www.pdfhacks.com/pdftk/* 4.2.1, B.2

[11] Sito internet delle librerie Java i-Text: *http://www.lowagie.com/iText/* 4.2.1

[12] Sito internet di GNU Gcj: *http://gcc.gnu.org/java/* 4.2.2, 4.2.2

[13] Sito internet personale di Roberto Iacono: *http://www.blia.it* 4.2.3

[14] Sito in cui trovare informazioni su OpenSignPDF: *http://opensignature.sourceforge.net/* 4.2.3

[15] Sito di IAIK (Technical University of Graz, Austria): *http://jce.iaik.tugraz.at* 4.2.3, 5.5, B.1

[16] Sito in cui trovare informazioni su log4j: *http://logging.apache.org/log4j/1.2/index.html* 4.2.3

[17] Sito in cui trovare informazioni sul progetto OpenJDK: *http://openjdk.java.net/* 5.1

[18] Sito in cui trovare il codice sorgente di tutte le classi di OpenJDK: *http://www.docjar.com/* 5.1

[19] Sito in cui trovare informazioni sul servizio di timestamp: *http://security.polito.it/ts/* 5.3.1, B.2

[20] Sito in cui trovare informazioni sulla libreria ncurses: *http://www.gnu.org/software/ncurses/ncurses.html* B.1