PARIPARI: REINGEGNERIZZAZIONE DI UNA LIBRERIA DI CRITTOGRAFIA

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Paolo Marchezzolo

Corso di laurea in Ingegneria Informatica

A.A. 2009-2010

UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

PARIPARI: REINGEGNERIZZAZIONE DI UNA LIBRERIA DI CRITTOGRAFIA

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Paolo Marchezzolo

A.A. 2009-2010

Indice

Sommario

In questa tesi di laurea si presenta il lavoro di reingegnerizzazione e in parte di sviluppo svolto sulla libreria di crittografia *SecurityFramework* nell'ambito del progetto PariPari.

Verrà prima trattata in maniera riassuntiva la struttura del progetto PariPari, e come una libreria di crittografia si inquadri nel contesto di tale progetto; dopo qualche breve cenno teorico su hashing e algoritmi di cifratura simmetrici e asimmetrici che verranno sfruttati nel codice, si esporrà l'attività svolta su *SecurityFramework* dall'aprile 2010 (data nel quale sono entrato a far parte del team di sviluppo di PariPari) ad oggi.

Capitolo 1

Il progetto PariPari

Lo scopo del progetto PariPari è di progettare un'applicazione peer-to-peer multifunzionale. L'aspetto caratteristico che lo contraddistingue da altre reti già diffuse tra gli utenti di Internet è la volontà di rendere disponibili attraverso un unico client la gran parte dei servizi che al momento richiedono l'uso di client e reti diverse. In questo modo l'utente, avendo a disposizione soltanto il client PariPari, potrà sfruttare sia le possibilità degli ormai diffusi software di file sharing quali eMule, Vuze[1] e altri, sia comunicare tramite Internet tramite i protocolli IRC[2], IM[3] e VoIP[4], ma anche usufruire di altri servizi quali web hosting, file hosting, DBMS[5], DHT[6], DNS[7].

In questo capitolo verrà fatta una panoramica sul progetto, a partire dall'architettura della rete fino a accennare alla struttura del client.

[1] meglio conosciuto come Azureus
[2] Internet Relay Chat
[3] Instant Messaging
[4] Voice over IP
[5] DataBase Management System
[6] Distributed Hash Table
[7] Domain Name System

Figura 1.1: Logo di PariPari

1.1 L'architettura della rete

L'architettura impiegata dalla rete PariPari è di tipo *serverless*. Questa potrebbe sembrare una scelta controproducente ad una analisi superficiale, visto che tutti i principali client peer-to-peer più affermati al giorno d'oggi (quelli basati sulla rete ED2K e quelli che sfruttano il protocollo BitTorrent) fanno uso di server. Tuttavia questa scelta architetturale porta a una serie di problemi:

- L'indisponibilità dei server blocca in gran parte le funzionalità della rete;
- I server sono vulnerabili a attacchi del tipo *Denial of Service*[8];
- Un eventuale aumento repentino o comunque non previsto degli utenti della rete può provocare un sovraccarico dei server e un conseguente malfunzionamento degli stessi.

Tutte queste problematiche sono evitate con la scelta di un'architettura priva di server: infatti, ogni nodo della rete può rimanere connesso per un periodo variabile di tempo senza compromettere l'integrità della rete (è estremamente improbabile che tutti i nodi o una gran parte di essi si disconnettano nello stesso

[8]Un attacco di tipo *Denial of Service* mira a rendere inutilizzabile un servizio offerto da un server inviando allo stesso un numero molto elevato di richieste, saturando così la banda e/o la capacità di elaborazione della macchina.

periodo di tempo); anche i problemi di scalabilità sono più facilmente gestibili, visto che non è necessario che sia un unico server a gestire tutte le richieste, ma il carico di lavoro è suddiviso su tutti i peer della rete.

PariPari si basa su una variante dell'algoritmo *Kademlia*, attualmente già utilizzata da software come eMule. Ne diamo una descrizione sommaria, trascurando i dettagli che non riguardano troppo da vicino l'argomento trattato nella tesina.

Kademlia si basa sul concetto di distanza in metrica XOR. Ogni host e ogni risorsa condivisa (ad esempio, i file di un sistema di file sharing) sono identificati da un ID a 160 bit. Per calcolare la distanza tra due ID, viene effettuato lo XOR dei due, e il risultato viene interpretato come un numero intero. E' facile verificare che la distanza così definita è una metrica in quanto rispetta le seguenti proprietà:

- $d(x, y) \geq 0 \quad \forall\ x, y$
- $d(x, y) = 0 \iff x = y$
- $d(x, y) = d(y, x) \quad \forall\ x, y$
- $d(x, y) + d(y, z) \geq d(x, z) \quad \forall\ x, y, z$

Possiamo individuare varie fasi nel funzionamento della rete:

- **Inizializzazione**: ad ogni nodo viene assegnato un ID univoco, come ad ogni risorsa presente sulla rete (in genere, per queste untime, viene usato l'hash SHA-1). Ogni nodo mantiene i riferimenti agli altri nodi di cui è venuto a conoscenza (come terne `<ID, IP address, port>`) ordinandoli secondo la distanza.
- **Ingresso di un nuovo nodo**: l'ingresso di un nuovo nodo nella rete avviene con una proceduta detta di *bootstrap*. Dopo aver calcolato l'hash delle proprie risorse e un ID casuale non utilizzato, avendo a disposizione l'indirizzo IP di almeno un altro nodo già facente parte della rete, il nuovo nodo lo aggiunge ai riferimenti già conosciuti, dopodichè procede a auto-ricercarsi

(la ricerca è spiegata nel punto successivo). Questa procedura consentirà al nuovo nodo di ricevere contatti da molti altri nodi sulla rete, e così facendo popolare l'elenco di nodi conosciuti.

- **Ricerca**: il nodo che effettua la ricerca invia la richiesta ai tre nodi conosciuti con distanza minima dall'ID cercato. Ogni nodo restituisce al nodo di partenza al più 20 ID scelti ancora tra quelli conosciuti con distanza minima dall'ID bersaglio della ricerca. La sorgente sceglie ancora una volta i tre più vicini all'ID cercato e inoltra la richiesta. Questo processo è ripetuto finchè non viene restituito il nodo cercato.

Si può verificare facilmente che non solo l'algoritmo di ricerca converge, ma che la sua complessità è logaritmica rispetto al numero dei nodi, caratteristica molto importante se si pensa che reti di questo genere possono arrivare a contenere migliaia o milioni di nodi.
La versione di *Kademlia* usata da PariPari presenta qualche leggera differenza con quella qui esposta, che è la versione originale del protocollo, ma non è opportuno soffermarsi ad analizzarle visto che non riguardano da vicino il lavoro svolto.

1.2 La struttura del client

Passiamo ora ad analizzare la struttura di ogni nodo della rete PariPari, cioè il client vero e proprio che sarà in esecuzione sulla macchina dell'utente.

L'approccio scelto e seguito nella progettazione dell'applicazione PariPari è quello a plugin. Un plugin è un programma non autonomo che dialoga e collabora con un altro programma per ampliarne le funzionalità. Questa struttura presenta degli enormi vantaggi in termini di modularità: infatti, i plugin comunicano tra di loro attraverso interfacce ben definite, e questo consente a modifiche ingenti nel codice di ripercuotersi in minima parte (nel caso ideale, di non ripercuotersi affatto) sul funzionamento di tutti gli altri plugin. Questa caratteristica rende anche lo sviluppo del codice molto più agevole: infatti il lavoro può essere

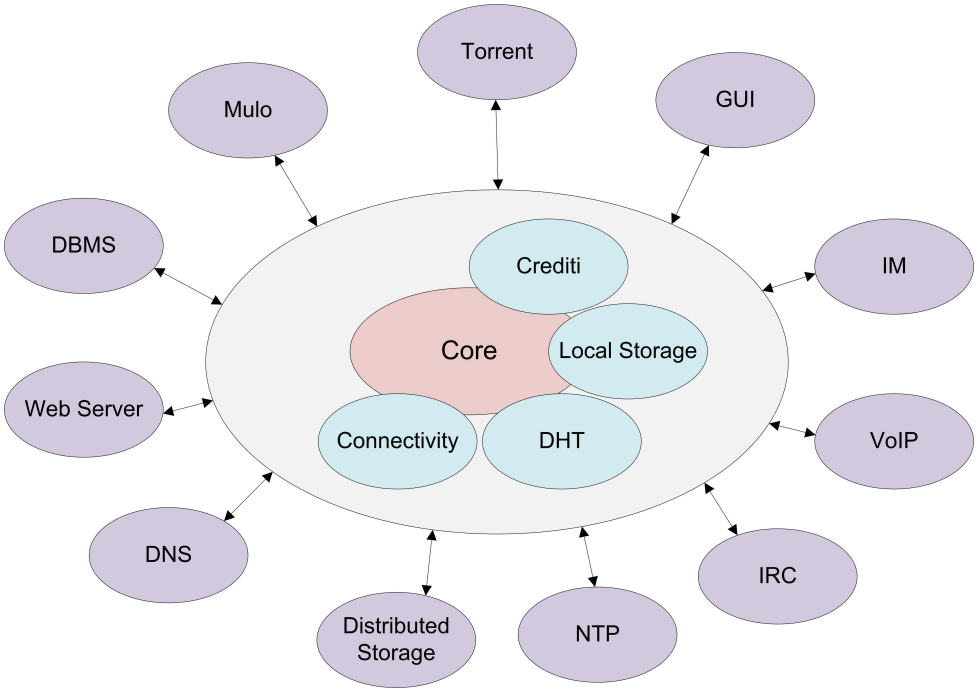

Figura 1.2: Struttura del progetto PariPari

suddiviso e assegnato a più gruppi di sviluppo separati, con interazione minima; ogni gruppo si occupa dello sviluppo/testing di un plugin, con poche figure di coordinamento al di sopra di essi.

Un'altra caratteristica fondamentale che deriva dall'architettura a plugin è la facilità nell'aggiungere funzionalità nuove al client PariPari, non necessariamente sviluppate dal team di sviluppo originale ma anche da terze parti Questa possibilità rende a tutti gli effetti il client più simile a una piattaforma per l'erogazione di servizi, non limitata a quelli per cui era stata pensata in un primo momento.

Come possiamo vedere dalla figura 1.2, i plugin si dividono in tre categorie principali. Il core, colorato in rosso, ha la funzione di ricevere e gestire le richieste da tutti gli altri plugin. Ogni plugin dialoga solo e soltanto con il core, ed ogni interazione tra due plugin diversi è mediata nello stesso modo.

I plugin colorati in azzurro fanno parte del cosiddetto *inner circle*, ossia quelli più in stretto contatto con il core. Si occupano di rendere disponibili le risorse: Local Storage per la gestione del disco, Connectivity e DHT per le connessioni

ad altri nodi PariPari. Fa eccezione il plugin Credits, che è di ausilio al core nel decidere con che priorità servire le varie richieste. Ogni plugin deve "pagare" una certa quantità di crediti ogni volta che richiede al core l'uso di una determinata risorsa. Questo sistema assicura che un plugin malevolo, oppure semplicemente progettato male, non metta in crisi l'intero client PariPari, negandogli l'accesso alle risorse nel caso le sue richieste siano troppo frequenti: il plugin si troverebbe nella situazione di non poter far fronte alle proprie stesse richieste con i crediti da esso posseuti.

Invece, i plugin colorati in viola fanno parte del cosiddetto *outer circle*. Sono la parte del client PariPari più a contatto con l'utilizzatore finale: in gran parte, infatti, sono l'implementazione dei vari servizi erogati dalla rete. Ad esempio, Mulo e Torrent si occupano di rendere disponibili all'utente i protocolli ED2K e BitTorrent per il file sharing, IM si occupa di tutti i protocolli di messaggistica istantanea, e così via. Fa eccezione il plugin GUI[9], il cui scopo è di rendere disponibile un'interfaccia chiara e di semplcie utilizzo all'utente, caratteristica essenziale in un software moderno.

Un'altra scelta interessante nello sviluppo di PariPari è l'uso del linguaggio di programmazione Java™. Oltre a rendere il software estremamente portabile, Java™ ci permette di utilizzare la tecnologia Java™ Web Start, consentendo di avviare PariPari dal proprio browser in maniera estremamente facile. Java™ presenta degli indubbi svantaggi rispetto ai suoi concorrenti (in primo luogo C++), specialmente a riguardo delle prestazioni, ma essi risultano difficilmente percepibili dall'utilizzatore finale e quindi la validità della scelta fatta non è inficiata da tali problemi.

1.3 Crediti

PariPari non usa i crediti soltanto per amministrare le risorse disponibili, come accennato in 1.2, ma anche, come ogni altro software peer-to-peer, per stabilire le

[9]Graphic User Interface

priorità di accesso alle risorse da aprte dei vari utenti. Questo potrebbe sembrare un argomento di secondaria importanza, visto che non migliora le prestazioni nè rende disponibili nuove funzionalità, tuttavia si rivela di fondamentale importanza perchè incide su come i nodi collaborano tra di loro.

Il sistema di crediti usato da altri software come eMule è abbastanza rudimentale: se Bob ha scaricato un file da Alice, avrà un debito nei suoi confronti; tuttavia, se Alice decidesse di scaricare un file da Celeste, non avrebbe alcun vantaggio dall'aver fatto scaricare Bob in precedenza. Invece, il sistema di crediti implementato da PariPari è ispirato al libero mercato: se Alice ha un credito verso Bob, e Bob ha un credito verso Celeste, Alice deve essere in grado di far valere il proprio credito anche nei confronti di Celeste, con l'interazione di Bob. Questo accorgimento rende il sistema di crediti più transitivo, incentivando maggiormente gli utenti a condividere una parte consistente delle proprie risorse per trarne vantaggio quando dovranno richiedere a loro volta risorse alla rete. Come si può notare dall'esempio precedente, ciascuna istanza di PariPari cerca di guadagnare da ogni transazione, proprio come nell'economia di mercato. Con questo sistema, un nuovo nodo connesso alla rete è costretto a "sprecare" una certa quantità di risorse, non troppo elevata, al fine di maturare crediti verso gli altri nodi e quindi poter cominciare a sfruttare le risorse altrui: questa fase iniziale scoraggia ancora di più i nodi *leechers*[10], che non possono più connettersi alla rete e liberamente scaricare contenuti e/o richiedere l'uso di altre risorse senza aver dato nulla in cambio.

1.4 *eXtreme Programming*

Per quanto riguarda l'approccio alla progettazione e allo sviluppo del codice di PariPari, si è scelta la tecnica dell'*eXtreme Programming* (detto anche XP). Questo

[10]Il nodo *leecher* punta soltanto al proprio tornaconto, senza preoccuparsi di contribuire al buon funzionamento della rete: tende quindi a scaricare files e poi non condividerli, o usare risorse altrui senza condividerne di proprie.

modello, classificato tra le strategie di sviluppo agili nell'ingegneria del software, prevede una particolare attenzione a alcuni punti nello sviluppo dell'applicazione in questione:

- Utilizzo di un ciclo di sviluppo molto breve, suddividendo il lavoro in molti, piccoli sottoproblemi;
- Particolare attenzione alla comunicazione e alla collaborazione nel team di sviluppo;
- *Feedback* il prima possibile in caso di problemi o malfunzionamenti;
- Frequenti riunioni del team di sviluppo (in PariPari, almeno una al mese);
- Test-driven Development: si procede alla scrittura di test che aiutano a definire i requisiti del codice prima del codice stesso; così facendo, è immediato verificare se il codice rispetta oppure no le specifiche richieste.

Questo modello si adatta perfettamente a un software molto modulare, in quanto risulta molto facile dividere il lavoro in piccoli compiti e tenere il ciclo di sviluppo il più breve possibile e garantendo così in feedback più veloce possibile. Visto l'approccio a plugin di PariPari, questa metodologia risulta ideale: ogni plugin gestisce un piccolo team di sviluppatori (in genere meno di 7-8) con il quale ci sono frequenti riunioni, e ci si tiene spesso in contatto utilizzando e-mail e lo strumento del *Google Group*[11].

[11]Si veda http://groups.google.com/ per ulteriori informazioni.

Capitolo 2

Hashing e crittografia

Come si sarà potuto notare dalla breve introduzione a PariPari al capitolo 1, l'uso di tecniche di hashing e crittografia è molto diffuso tra i vari plugin: le reti ED2K e BitTorrent sfruttano in maniera intensiva tali algoritmi, e anche per servizi quali la messaggistica o lo storage distribuito, il loro utilizzo è di essenziale importanza sia per il funzionamento stesso dei plugin, sia per garantire un livello più alto possibile di *privacy* per i dati che si andrà a trasmettere su Internet (notoriamente un canale altamente insicuro).

In questo capitolo vengono presentati brevemente i concetti e gli algoritmi di hashing e crittografia che poi verranno implementati in *SecurityFramework*.

2.1 Hashing

In crittografia, per funzione di hashing si intende una funzione deterministica che permetta di mappare un messaggio, di dimensione arbitraria in quello che spesso viene definito il *Message Digest* del messaggio stesso, cioè una sequenza di bit di dimensione prefissata, che abbia delle proprietà ben definite:

- Dato un messaggio m, è semplice calcolare $h(m)$[1].

[1]Con la notazione $h(m)$ si intende la funzione di hashing applicata al messaggio m.

- Dato $h(m)$, è "difficile"[2] risalire a m. Questa caratteristica viene spesso espressa dicendo che la funzione di hash è una funzione "a una via"[3].

- Dato un messaggio m_1, è "difficile" trovare m_2, con $m_2 \neq m_1$ tale che $h(m_1) = h(m_2)$[4].

- Cambiando anche un solo bit del messaggio iniziale, il suo *Message Digest* deve mutare completamente; in altre parole, la funzione di hash, pur rimanendo deterministica, deve comportarsi in maniera simile a una funzione casuale.

In altre parole, un malintenzionato non deve essere in grado di alterare il messaggio iniziale senza far modificare anche il *Message Digest* del messaggio stesso.

Funzioni che soddisfano tali proprietà vengono impiegate per vari scopi, legati alle firme digitali e all'autenticazione dei soggetti coinvolti in uno scambio di messaggi più in generale. Oltre a questo, vengono anche sfruttate come *checksum*, oppure come identificativi per i file condivisi nei software di file sharing.

Nel capitolo 1 si sarà già potuto notare che le funzioni di hash sono utilizzate in più punti nei vari plugin di PariPari: ad esempio, viene sfruttato l'hash SHA-1 per calcolare gli ID delle risorse condivise su una DHT; inoltre, come detto poco fa, la rete ED2K e in genere i protocolli di file sharing sfruttano in maniera intensiva queste tecniche.

2.2 Altri algoritmi crittografici

Oltre alle funzioni di hashing, è necessario disporre di strumenti per cifrare dati sensibili per farli transitare su un canale non sicuro quale è Internet. Ci sono due approcci differenti a questo problema: la crittografia simmetrica e la crittografia

[2] Più che difficile, forse sarebbe più opportuno parlare di "computazionalmente intrattabile": deve essere necessario un tempo talmente ampio per risolvere il problema tale da vanificare ogni utilità della soluzione.

[3] "One-way function" in inglese.

[4] Nel caso si riesca a trovare un m_2 con tali proprietà, si ha individuato una *collisione*.

asimmetrica. La differenza tra i due consiste nel fatto che la crittografia simmetrica fa uso di una sola chiave[5], mentre la crittografia asimmetrica richiede l'uso di due chiavi differenti per crittare e decrittare il messaggio.

Il principale algoritmo simmetrico utilizzato da PariPari è RC4, mentre come algoritmo asimmetrico viene utilizzato RSA. Non ne verrà trattato in dettaglio il funzionamento, tuttavia si ritiene oppurtuna una ulteriore, breve trattazione a riguardo del funzionamento della crittografia asimmetrica.

Cenni teorici sull'algoritmo RSA

Nella crittografia simmetrica, l'unico modo per essere certi che i propri dati non verranno decrittati (in un tempo ragionevole, vedi nota a pagina 12) è mantenere segreta la chiave di cifratura/decifratura utilizzata. Purtroppo questo vincolo è molto scomodo in quanto costringe a utilizzare un canale alternativo (sicuro) per lo scambio della chiave. La crittografia asimmetrica nasce proprio per ovviare a questo problema. Se Alice deve inviare un messaggio a Bob utilizzando un algoritmo asimmetrico, andrà a cercare la chiave pubblica di Bob (che sarà pubblicata, ad esempio, su una base di dati reperibile online) e la utilizzerà per cifrare il proprio messaggio. Bob utilizzerà quindi la propria chiave privata (nota soltanto a lui) per decifrare il messaggio. Nonostante la chiave utilizzata in fase di cifratura sia conosciuta da chiunque, soltanto Bob può risalire al messaggio iniziale di Alice, in quanto soltanto lui è in possesso della chiave privata necessaria per decrittarlo.

L'algoritmo principale che rende possibile questo genere di crittografia è detto RSA[6]; ne forniamo qui una breve descrizione.

La generazione della coppia di chiavi pubblica e privata avviene nel seguente modo:

[5]Una chiave è un parametro dell'algoritmo di crittografia che, insieme al messaggio da cifrare, influisce sull messaggio cifrato.

[6]da Rivest, Shamir e Adleman, gli inventori dell'algoritmo.

1. Si scelgono due numeri primi p e q abbastanza grandi per garantire il livello di sicurezza desiderato.

2. Si calcolano $n = p \cdot q$ e $z = (p-1) \cdot (q-1)$

3. Si sceglie un numero d coprimo con z

4. si trova e tale che $e \cdot d \equiv 1 \pmod{z}$

Completati questi passaggi preliminari, si può procedere alla cifratura del messaggio. Esso deve essere diviso in blocchi di al massimo k bit , dove k è il massimo intero tale che sia soddisfatta la disuguaglianza $2^k < n$ (chiameremo P questi blocchi). A questo punto, per ogni blocco, si calcola $C = P^e \pmod{n}$. L'insieme di tutti i blocchi cifrati in questo modo rappresentano il nostro messaggio cifrato. Il destinatario calcolerà, per ogni blocco, $P = C^d \pmod{n}$. In questo modo vengono riottenuti tutti i blocchi di partenza. Come si è potuto intuire, la coppia (e, n) rappresenta la chiave pubblica del destinatario, mentre la coppia (d, n) rappresenta la sua chiave privata.

La robustezza del metodo è basata sul fatto che, pur un malintenzionato conoscendo n (che è pubblico), fattorizzare questo numero (e quindi trovare e e d che gli permetterebbero di decifrare correttamente la comunicazione) è estremamente oneroso dal punto di vista computazionale, e non esiste un algoritmo efficiente per effettuale tale operazione.

Purtroppo, per raggiungere un livello apprezzabile di sicurezza, le chiavi utilizzate dall'algoritmo sono considerevolmente più lunghe di quelle usate da altri algoritmi simmetrici. Questo rende RSA piuttosto lento nell'elaborare grosse quantità di dati. Proprio per questo, in generale (e anche all'interno di PariPari) viene utilizzato per scambiare in maniera sicura una chiave simmetrica, e poi utilizzare algoritmi meno onerosi come RC4 o AES per scambiare le informazioni.

Capitolo 3

SecurityFramework

In questo capitolo verrà esposta l'effettiva implementazione di quanto trattato nel capitolo 2, nonchè il processo di sviluppo, revisione e riscrittura di parti del *SecurityFramework* già presente in PariPari quando sono entrato a far parte del progetto.

3.1 Perchè *SecurityFramework?*

La domanda può sembrare banale ad una analisi superficiale, ma in realtà non lo è affatto. In effetti, come è già stato detto, era già presente un *SecurityFramework* quando sono entrato a far parte del team di sviluppo di PariPari, ma veniva utilizzato sporadicamente dai plugin, che spesso preferivano implementare metodi per il calcolo di hash o altre funzioni simili direttamente al loro interno piuttosto che chiamare i metodi del *Framework*.

Le caratteristiche che si vogliono ottenere da *SecurityFramework* sono le seguenti:

- Deve mettere a disposizione tutti i più comuni ed utilizzati metodi per l'hashing e la crittografia utilizzati nel progetto;
- Deve essere ben documentato;
- Deve essere il più semplice possibile da utilizzare.

L'intento di queste richieste è far sì che ogni plugin che necessiti di tali funzionalità non abbia convenienza a scriverle da zero, ma richiami i metodi del *Framework*. Questo, oltre ad alleviare il lavoro degli altri sviluppatori, dà la possibilità di avere un maggiore controllo sulle librerie esterne usate. Come vedremo nelle prossime sezioni, spesso ci si appoggia a tali librerie per implementare metodi di crittografia, in particolare a *BouncyCastle*[1]; questo a causa dei limiti delle implementazioni fornite da Java™ di tali algoritmi. Se si riuscisse a raggiungere l'obiettivo di far utilizzare a tutti i plugin le implementazioni del *SecurityFramework*, sarebbe possibile modificare i metodi (ad esempio usando una libreria differente, o implementandoli direttamente all'interno della libreria) senza che sia necessaria alcuna modifica al codice, al di fuori della libreria stessa.

3.2 Raccolta dei requisiti

Per far rispettare le richieste elencate nella sezione 3.1, è stata indispensabile una attività di raccolta dei requisiti, per capire quali algortimi venivano effettivamente utilizzati e sarebbe stato il caso di implementare, e quali erano superflui. E' stata condotta una "intervista" (non dal vivo ma tramite lo strumento del *Google Group*, utilizzato come strumento di comunicazione e coordinazione all'interno del progetto) ai vari Plugin Leaders e Team Leaders per reperire queste informazioni.

In un primo momento, erano emerse le seguenti necessità:

1. Implementanzione dei seguenti algoritmi di hashing:
   - MD4,
   - MD5,
   - SHA-1;
2. Implementazione di RSA che supporti anche chiavi a 384 bit;
3. Implementazione di RC4.

[1]Si veda la sezione 3.4.

I requisiti si sono andati man mano ampliando e definendo in maniera più precisa durante la stesura del codice, come vedremo più avanti.

3.3 La situazione iniziale

Come già più volte accennato, era già presente un *SecurityFramework* all'interno di PariPari, tuttavia, come si è potuto appurare dall'analisi dei requisiti, le funzionalità offerte non rispondevano completamente ai bisogni degli altri plugin. Oltre a questo, pur essendo la libreria ben documentata, non era presente alcuna pagina sulla pagina wiki[2] del progetto, e forse anche questo contribuiva al suo scarso utilizzo.

Le funzionalità offerte erano le seguenti:

1. Algoritmi di hashing:
   - MD5,
   - SHA-1,
   - SHA-256,
   - SHA-384,
   - SHA-512;
2. Implementazione di RSA (utilizzando l'implementazione di Java™);
3. Metodi per la firma digitale di blocchi di dati;
4. Implementazione di AES[3].

Come possiamo vedere, alcuni di queste funzionalità rispondono alle esigenze espresse in precedenza, ma mancano alcuni requisiti, e sono presenti caratteristiche non richieste che, pur potenzialmente essendo utili, rendono la libreria di più difficile utilizzo.

[2] Si veda http://paripari.it/mediawiki/index.php/Main_Page

[3] Advanced Encryption Standard

3.4 Implementazione

Completata la fase di raccolta dei requisiti, si è proceduto con l'implementazione delle funzionalità richieste.

Per quanto riguarda il calcolo delle funzioni di hashing, non si sono incontrate grosse difficoltà; per quanto riguarda tutti gli hash richiesti, eccetto MD4, Java™ mette a disposizione un'implementazione perfettamente funzionante, cosa che già era sfruttata dal precedente *SecurityFramework*. L'unica novità è stata l'introduzione dell'hasinh MD4, come richiesto; è stato fatto ricorso alla libreria esterna *BouncyCastle*, in quanto Java™ non implementa tale algoritmo. Il plugin Mulo aveva implementato la stessa funzione da zero, ma si è scelto di usare *BouncyCastle* sia per la semplicità che ovviamente ne deriva, sia perchè tale libreria sarebbe comunque stata sfruttata più avanti, quindi non era richiesto di importare nessuna classe addizionale per sfruttare quell'implementazione. Sono state previsti tre metodi per gli hasing più comuni (MD4, MD5 e SHA-1), mentre per i restanti membri della famiglia SHA è stata creata una procedura parametrica che ne pemette l'utilizzo. Tutti i metodi richiedono in ingresso un array di bytes (`byte[]`) e restituiscono l'hash richiesto con un analogo `byte[]`.

Anche per quanto riguarda l'algoritmo RSA si è dovuti ricorrere all'uso di *BouncyCastle*: infatti, pur Java™ mettendo a disposizione una implementazione di tale algoritmo, essa prevede chiavi di dimensione almeno 512 bit, e quindi non sarebbe stato possibile ottenere una codifica con chiavi di soli 384 bit. Si è previsto un metodo per generare un oggetto di tipo `KeyPair`, contenente una coppia di chiavi pubblica e privata abbinate della lunghezza richiesta; dopodichè sono stati implementati due semplici metodi che consentono la cifratura e la decifratura di un array di byte, come nelle funzioni di hashing, fornita una opportuna chiave pubblica/privata a seconda dell'operazione richiesta.

Sempre per quanto riguarda la crittografia asimmetrica, si è anche provveduto a realizzare alcuni metodi (richiesti dal plugin Distributed Storage, di cui faccio parte, proprio durante lo sviluppo del codice) che, dato in input uno stream di

input/output (una istanza di `InputStream` o `OutputStream`), provvedono a restituire una analoga istanza (in questo caso si tratterà di un `CipherInputStream` o `CipherOutputStream`) che si occupi di cifrare o decifrare i dati con RSA e con la chiave assegnata prima di restituirli con il metodo `read()` per quanto riguarda gli stream di input, oppure prima di restituirli in output nell'altro caso.

Per quanto riguarda l'implementazione dell'algoritmo RC4, non presente nella versione iniziale di *SecurityFramework*, si è potuto ricorrere all'implementazione nativa di Java™ , che non presenta particolari limitazioni in questo caso. Si sono previsti tre metodi, analogamente a quanto fatto per RSA: il primo consente di generare una chiave RC4 della lunghezza desiderata, mentre il secondo e il terzo provvedono a crittare e decrittare un array di byte (`byte[]`) una volta fornita la chiave opportuna.

BouncyCastle

Come già menzionato poco fa, *BouncyCastle*[4], spesso abbreviato BC, è una raccolta di API[5] per le principali funzioni di crittografia. Nella libreria è anche incluso un `provider`, cioè una implementazione di tutte le funzionalità contenute nella Java™ Cryptography Extension; è quindi possibile sia sfruttare direttamente le classi di *BouncyCastle* che, con una semplice istruzione, impostare come predefinito il `provider` di *BouncyCastle* e poter così invocare gli stessi metodi o comunque non dover modificare il codice rispetto a quando si faceva uso di un altro `provider`.

In *SecurityFramework* abbiamo dovuto far ricorso a al supporto di *BouncyCastle* non tanto per problemi di prestazioni o di altro genere, ma semplicemente perchè alcune funzionalità richieste alla libreria non sono offerte dalla Java™ Cryptography Extension, oppure hanno delle pesanti limitazioni (ho già citato in precedenza la dimensione minima della chiave RSA, 512 bit, che impediva ai metodi di avere la flessibilità richiesta, oppure la mancanza dell'algoritmo RC4).

[4] http://www.bouncycastle.org/java.html

[5] Application Programming Interface

3.5 Testing

Come già citato in precedenza (sezione 1.4), in PariPari si adotta una strategia, detta Test-driven Development, che prevede la scrittura di alcuni test prima della scrittura del codice vero e proprio, in modo che quando si andrà a sviluppare quest'ultimo, i requisiti siano già ben cristallizzati all'interno della classe di test. Durante lo sviluppo, comunque, è necessario continuare a aggiornare e ampliare la classe di test, in maniera che questa possa essere lanciata in qualsiasi momento e individui tempstivamente anomalie o malfunzionamenti di quanto si sta sviluppando.

Nel caso di *SecurityFramework*, questa tecnica è stata utilizzata in parte. Essendo le specifiche di tutti gli algoritmi in questione già ben definite in appositi documenti[6], si è utilizzato il testing come strumento prezioso per poter individuare malfunzionamenti in maniera tempestiva.

Va notato che questa pratica è parecchio onerosa, in quanto la lunghezza della classe di test sviluppata spesso supera di diverse volte il codice (nel caso in questione, risulterà di circa 1000 righe di codice, mentre la libreria ne conta poco più di 300), e spesso non semplice, visto che il test di alcuni metodi può risultare più difficile da progettare e implementare dei metodi stessi. Tuttavia, questo approccio è stato prezioso in alcune occasioni per identificare e risolvere piccole anomalie che sarebbero probabilmente passate inosservate, e avrebbero potuto generare non pochi problemi in futuro, quando fossero state riscontrate senza capire la loro origine.

3.6 Verifica dei requisiti

Terminata la stesura del codice e il suo testing, eliminati i principali difetti e completata una prima fase di documentazione, sono stati contattati i Team e Plugin Leaders che avevano richiesto le varie funzionalità (si veda quanto riportato

[6] A titolo di esempio, si veda http://tools.ietf.org/html/rfc1321 per MD5.

in sezione 3.2), ed è stato loro sottoposto il codice, chiedendo se effettivamente rispondeva alle loro esigenze, se ne erano emerse di nuove o se comunque avevano altre richieste da effettuare a *SecurityFramework*.

Quello che è emerso da questa verifica è che le funzionalità già implementate erano proprio quelle necessarie ai plugin; tuttavia, è emersa la necessità di aggiungere due metodi per quanto riguarda l'algoritmo RSA, che si occupassero di generare una chiave pubblica o privata a partire da modulo e esponente. Il plugin Mulo, infatti, sfrutta al proprio interno questo tipo di generazione delle chiavi, e quindi era necessario fornire metodi per gestire anche questa esigenza.

L'implementazione di quanto richiesto è stata, pur nella sua semplicità, più complessa di quella del resto della libreria. Infatti, non è possibile generare un oggetto di tipo `PublicKey` o `PrivateKey` partendo da moldulo e esplonente, almeno non in maniera diretta. E' necessario generare prima un oggetto di tipo `RSAPublicKeySpec` o `RSAPrivateKeySpec` con il modulo e l'esponente assegnati, dopodichè usare un `KeyFactory` che provvede a convertirli in chiavi pubbliche e private, come richiesto.

3.7 Documentazione

A contorno del codice, va sempre fornita una adeguata documentazione. Questa necessità è particolarmente sentita per librerie, come quella in questione, che dovranno essere poi riutilizzate da un numero piuttosto cospicuo di persone. Una documentazione non adeguata può facilmente portare a un uso improprio delle funzionalità, oppure a un abbandono della libreria per una implementazione nativa dei metodi, che, come già riportato nella sezione 3.1, è proprio quello che si vuole evitare.

Per soddisfare queste esigenze, oltre alla normale documentazione JavaDoc, che è standard nel progetto PariPari, è stata prodotta una pagina web inserita nel progetto wiki di PariPari, che illustra l'uso dei metodi forniti da *SecurityFramework* riportandone le interfacce e facendo alcuni semplici esempi.

Capitolo 4

Conclusioni

In questo lavoro è stato descritta l'attività reingegnerizzazione e sviluppo svolta sulla libreria *SecurityFramework*. Il compito svolto non può che essere considerato positivo. Infatti, oltre a aver incontrato molti strumenti nuovi per la mia esperienza sullo sviluppo di codice (l'uso di *Eclipse* come ambiente di sviluppo, o *JUnit* per il testing) e l'esperienza di lavorare con un team di sviluppo, che hanno sicuramente favorito una crescita personale, il risultato della reingegnerizzazione è sicuramente soddisfacente: si è prodotta una libreria ben documentata, semplice da leggere e da utilizzare, che risponde ai requisiti richiesti nella quasi totalità.

Le tecniche crittografiche con cui sono entrato in contatto e che ho iniziato a conoscere sono di vitale importanza in molti ambiti in quanto l'esigenza di tenere segrete le comunicazioni su un canale pubblico è sempre esistita, ben prima dell'avvento dell'infomatica. In particolare, per quanto riguarda le comunicazioni sulla Rete, è indispensabile poter proteggere dati riservati come il contenuto del traffico che si invia, oppure avere la certezza dell'identità del proprio interlocutore tramite tecniche di firma digitale. Da queste considerazioni emerge ancora una volta l'utilità di avere un'unica libreria condivisa in un progetto ampio e multifunzionale come PariPari. Gli algoritmi crittografici implementati, infatti, sono indispensabili per garantire le funzionalità appena descritte: l'hashing per quanto riguarda le firme digitali e l'integrità dei dati (nonchè altri utilizzi, non meno im-

portanti, negli algoritmi quali *Kademlia* o le reti ED2K/BitTorrent), mentre la crittografia simmetrica e asimmetrica per la riservatezza dei dati inviati.

Il processo di sviluppo e revisione del *SecurityFramework* probabilmente non termina con questa tesi: infatti, più il progetto PariPari evolverà e diventerà più complesso e articolato, più è probabile che siano necessarie nuove funzionalità e modifiche a quelle già implementate; tuttavia il codice di partenza che verrà trovato da un futuro sviluppatore renderà il suo compito più agevole, avendo già una buona linea guida da seguire.

Un'altra nota positiva incontrata durante lo svolgimento del lavoro appena documentato è stata la collaborazione e la disponibilità dei (seppur pochi) PariParisti (come amiamo definirci all'interno del progetto) con cui sono venuto in contatto. Nonostante le mie difficioltà iniziali nell'uso degli strumenti e nelle prime fasi di revisione e stesura del codice, sono sempre stato facilitato dal loro supporto e non mi sono quasi mai trovato in difficoltà. Di sicuro il progetto PariPari rappresenta un cospicuo investimento a livello di tempo ed energie, tuttavia ribadisco come il fare esperienza, spesso per la prima volta, con il lavoro di squadra sia estremamente positivo e avvicini a ciò che ci aspetterà un domani nel mondo del lavoro.

Appendice A

Codice sorgente di *SecurityFramework*

A.1 HashStandards.java

```
*/
package Security;


/**
 * @author Fabio Dominio - PariPari Group
 * @version 0.0.1<br><br>
 *
 * This class provides hashing standard algorithm names and digest
     sizes.
 *
 */
public enum HashStandards {
  MD5{public String getAlgorithm(){return "MD5";}
        public short digestSize(){return 128;}},

  SHA1{public String getAlgorithm(){return "SHA-1";}
        public short digestSize(){return 160;}},
```

```
  SHA256{public String getAlgorithm(){return "SHA-256";}
        public short digestSize(){return 256;}},

  SHA384{public String getAlgorithm(){return "SHA-384";}
        public short digestSize(){return 384;}},

  SHA512{public String getAlgorithm(){return "SHA-512";}
        public short digestSize(){return 512;}};

/**
 * Method that returns digester standard algorithm name
 * @return digester algorithm name
 */
public abstract String getAlgorithm();

/**
 * Method that returns digester standard algorithm digest size [bit]
 * @return digester algorithm digest size
 */
public abstract short digestSize();

}
```

A.2 ParipariSecurity.java

```
package Security;

import java.io.InputStream;
import java.io.OutputStream;
import java.math.BigInteger;
import java.security.GeneralSecurityException;
import java.security.InvalidKeyException;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
```

```
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.RSAPrivateKeySpec;
import java.security.spec.RSAPublicKeySpec;

import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

import org.bouncycastle.jce.provider.JDKKeyPairGenerator;
import org.bouncycastle.jce.provider.JDKMessageDigest;
/**
 *
 * @author Paolo Marchezzolo
 * @version 0.1
 * <br><br>
 * This class provides an easy-to-use implementation of:
 * <li>message digest - hashing algorithms (MD4, MD5 and SHA-X
     provided),
 * <li>key generation , encryption and decryption using RC4
     algorithm,
 * <li>key pair generation, encryption and decryption of messages
     using RSA algorithm.
 */
public class ParipariSecurity {
  /**
   * Calculates the hash code of an array of bytes.
   * @param in The array to be processed.
   * @param algorithm :<ul>
   * <li>MD5 (128 bit --> 16 byte hash);</li>
   * <li>SHA-1 (160 bit --> 20 byte hash);</li>
   * <li>SHA-256 (256 bit --> 32 byte hash);</li>
   * <li>SHA-384 (384 bit --> 48 byte hash);</li>
   * <li>SHA-512 (512 bit --> 64 byte hash);</li></ul>
```

```
     * @return The hash code required or <code>null</code> if an
         exception is thrown.
     * @throws NoSuchAlgorithmException if the algorithm passed is
         invalid.
     * @throws IllegalArgumentException if you passed a null algorithm
          or input array.
     */
    public static byte[] getHash (byte[] in, HashStandards algorithm)
        throws NoSuchAlgorithmException{
      if (algorithm == null || in == null){
        throw new IllegalArgumentException("[ParipariSecurity]: Null
            input or null algorithm");
      }
        return MessageDigest.getInstance(algorithm.getAlgorithm()).
            digest(in);
    }


    /**
     * Calculates the message digest of the array in input using SHA-1
          algorithm.
     * @param in The array to be processed
     * @throws IllegalArgumentException if you passed a null input
         array.
     */
    public static byte[] getSHA1Hash(byte[] in) {

      try {
        return getHash(in, HashStandards.SHA1);
      }
      catch (NoSuchAlgorithmException e) {
        //we don't throw anything here because we know that the passed
            algorithm IS valid
        //so this catch block should never be executed.

        System.out.println("[ParipariSecurity] Unsupported Algorithm
```

```
            (? this should never happen) "+e.getMessage());
        return null;
    }
}

/**
 * Calculates the message digest of the array in input using MD5
     algorithm.
 * @param in The array to be processed
 * @throws IllegalArgumentException if you passed a null input
     array.
 */
public static byte[] getMD5Hash(byte[] in) {
    try {
        return getHash(in, HashStandards.MD5);
    }
    catch (NoSuchAlgorithmException e) {
        //we don't throw anything here because we know that the passed
            algorithm IS valid
        //so this catch block should never be executed.

        System.out.println("[ParipariSecurity] Unsupported Algorithm
            (? this should never happen) "+e.getMessage());
        return null;
    }
}

/**
 * Calculates the message digest of the array in input using MD4
     algorithm.<br>
 * BouncyCastle implementation of MD4 algorithm is used here
     because Java lacks an implementation of such algorithm.
 * @param in The array to be processed .
 * @throws IllegalArgumentException if you passed a null input
     array.
 */
```

```
public static byte[] getMD4Hash(byte[] in){
  if (in == null) {
    throw new IllegalArgumentException("[ParipariSecurity]: Null
        input or null algorithm");
  }
  JDKMessageDigest.MD4 messagedigest = new JDKMessageDigest.MD4();
  return messagedigest.digest(in);
}

/**
 * Generates a key pair (public key and private key) for the RSA
    algorithm.
 * @param keysize The keysize.
 * @return A KeyPair object containing the pair generated.
 */
public static KeyPair generateRSAKeys(int keysize){
  if (keysize<=0) {
    throw new IllegalArgumentException("Cannot generate a zero-
        length key");
  }
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(keysize);
  return keygen.genKeyPair();
}

public static PublicKey generateRSAPublicKey(BigInteger mod,
    BigInteger exp){

  if (mod==null||exp==null){
    throw new IllegalArgumentException("Cannot generate keys with
        a null modulus or exponent");
  }

  try {
    java.security.Security.addProvider(new org.bouncycastle.jce.
        provider.BouncyCastleProvider());
```

```
      KeyFactory kf = KeyFactory.getInstance("RSA");
      RSAPublicKeySpec spec = new RSAPublicKeySpec(mod, exp);
      return kf.generatePublic(spec);

    }
    catch(GeneralSecurityException e){
      // unsupported algorithm or bad padding
      //this should not happen...
      System.out.println("[ParipariSecurity]: Unsupported algorithm
          or bad padding "+e.getMessage());
      return null;
    }

  }

  public static PrivateKey generateRSAPrivateKey (BigInteger mod,
      BigInteger exp){

    if (mod==null||exp==null){
      throw new IllegalArgumentException("Cannot generate keys with
          a null modulus or exponent");
    }

    try {
      java.security.Security.addProvider(new org.bouncycastle.jce.
          provider.BouncyCastleProvider());
      KeyFactory kf = KeyFactory.getInstance("RSA");
      RSAPrivateKeySpec spec = new RSAPrivateKeySpec(mod, exp);
      return kf.generatePrivate(spec);

    }
    catch(GeneralSecurityException e){
      // unsupported algorithm or bad padding
      //this should not happen...
      System.out.println("[ParipariSecurity]: Unsupported algorithm
          or bad padding "+e.getMessage());
```

```
      return null;
    }
  }

  /**
   * Encrypts a block of data using RSA algorithm.
   * @param in The data to be encrypted
   * @param pk The public key
   * @return The encrypted data required, or <code>null</code> if an
       exception is thrown.
   * @throws InvalidKeyException If the key is not valid
   */
  public static byte[] encryptRSA (byte[] in, PublicKey pk) throws
      InvalidKeyException{
    if (in == null) {
      throw new IllegalArgumentException("Cannot encrypt a null
          block");
    }
    if (pk == null){
      throw new IllegalArgumentException("Cannot encrypt with a null
          key");
    }
    try{
      java.security.Security.addProvider(new org.bouncycastle.jce.
          provider.BouncyCastleProvider());
      Cipher cipher = Cipher.getInstance("RSA", "BC");
      cipher.init(Cipher.ENCRYPT_MODE, pk);
      return cipher.doFinal(in);
    }
    catch (InvalidKeyException e){
      throw new InvalidKeyException("[ParipariSecurity]: Invalid key
          ");
    }
    catch(GeneralSecurityException e){
      // unsupported algorithm or bad padding
      //this should not happen...
```

```
            System.out.println("[ParipariSecurity]: Unsupported algorithm
                or bad padding "+e.getMessage());
            return null;
        }
    }

    /** Decrypts a block of data using RSA algorithm.
     * @param in The data to be decrypted
     * @param pk The private key
     * @return The plain data resulting from the decryption, or <code>
         null</code> if an exception is thrown.
     * @throws InvalidKeyException If the key is not valid
     */
    public static byte[] decryptRSA (byte[] in, PrivateKey pk) throws
        InvalidKeyException{
        if (in == null) {
            throw new IllegalArgumentException("Cannot encrypt a null
                block");
        }
        if (pk == null){
            throw new IllegalArgumentException("Cannot encrypt with a null
                 key");
        }
        try{
            java.security.Security.addProvider(new org.bouncycastle.jce.
                provider.BouncyCastleProvider());
            Cipher cipher = Cipher.getInstance("RSA", "BC");
            cipher.init(Cipher.DECRYPT_MODE, pk);
            return cipher.doFinal(in);
        }
        catch (InvalidKeyException e){
            throw new InvalidKeyException("[ParipariSecurity]: Invalid key
                ");
              }
        catch(GeneralSecurityException e){
            // unsupported algorithm or bad padding
```

```
      System.out.println("[ParipariSecurity]: Unsupported algorithm
          or bad padding "+e.getMessage());
      return null;
    }
  }

  /**
   * Returns a CipherInputStream that is the RSA encryption of the
       given InputStream.
   * @param in The InputStream to be encrypted.
   * @param pk The PublicKey used for the RSA encryption.
   * @throws InvalidKeyException If the key is not valid
   */
  public static CipherInputStream encryptInputStreamRSA(InputStream
      in, PublicKey pk) throws InvalidKeyException{

    if (in == null) {
      throw new IllegalArgumentException("Cannot encrypt a null
          stream");
    }
    if (pk == null){
      throw new IllegalArgumentException("Cannot encrypt with a null
           key");
    }

    try{
      java.security.Security.addProvider(new org.bouncycastle.jce.
          provider.BouncyCastleProvider());
      Cipher cipher = Cipher.getInstance("RSA", "BC");
      cipher.init(Cipher.ENCRYPT_MODE, pk);
      return new CipherInputStream(in, cipher);
    }
    catch (InvalidKeyException e){
      throw new InvalidKeyException("[ParipariSecurity]: Invalid key
          ");
    }
```

```
    catch(GeneralSecurityException e){
      // unsupported algorithm or bad padding
      //this should not happen...
      System.out.println("[ParipariSecurity]: Unsupported algorithm
          or bad padding "+e.getMessage());
      return null;
    }
  }

  /**
   * Returns a CipherInputStream that is the RSA decryption of the
       given InputStream.
   * @param in The InputStream to be decrypted.
   * @param pk The PrivateKey used for the RSA decryption.
   * @throws InvalidKeyException If the key is not valid
   */
  public static CipherInputStream decryptInputStreamRSA(InputStream
      in, PrivateKey pk) throws InvalidKeyException{
    if (in == null) {
      throw new IllegalArgumentException("Cannot decrypt a null
          stream");
    }
    if (pk == null){
      throw new IllegalArgumentException("Cannot decrypt with a null
          key");
    }
    try{
      java.security.Security.addProvider(new org.bouncycastle.jce.
          provider.BouncyCastleProvider());
      Cipher cipher = Cipher.getInstance("RSA", "BC");
      cipher.init(Cipher.DECRYPT_MODE, pk);
      return new CipherInputStream(in, cipher);
    }
    catch (InvalidKeyException e){
      throw new InvalidKeyException("[ParipariSecurity]: Invalid key
          ");
```

```
    }
    catch(GeneralSecurityException e){
      // unsupported algorithm or bad padding
      //this should not happen...
      System.out.println("[ParipariSecurity]: Unsupported algorithm
          or bad padding "+e.getMessage());
      return null;
    }
  }

  /**
   * Returns a CipherOutputStream that is the RSA encryption of the
       given OutputStream.
   * @param in The OutputStream to be encrypted.
   * @param pk The PublicKey used for the RSA encryption.
   * @throws InvalidKeyException If the key is not valid
   */
  public static CipherOutputStream encryptOutputStreamRSA(
      OutputStream out, PublicKey pk) throws InvalidKeyException{
    if (out == null) {
      throw new IllegalArgumentException("Cannot encrypt a null
          stream");
    }
    if (pk == null){
      throw new IllegalArgumentException("Cannot encrypt with a null
          key");
    }
    try{
      java.security.Security.addProvider(new org.bouncycastle.jce.
          provider.BouncyCastleProvider());
      Cipher cipher = Cipher.getInstance("RSA", "BC");
      cipher.init(Cipher.ENCRYPT_MODE, pk);
      return new CipherOutputStream(out, cipher);
    }
    catch (InvalidKeyException e){
      throw new InvalidKeyException("[ParipariSecurity]: Invalid key
```

```
            ");
        }
        catch(GeneralSecurityException e){
            // unsupported algorithm or bad padding
            //this should not happen...
            System.out.println("[ParipariSecurity]: Unsupported algorithm
                or bad padding "+e.getMessage());
            return null;
        }
    }

    /**
     * Returns a CipherOutputStream that is the RSA decryption of the
         given OutputStream.
     * @param in The OutputStream to be decrypted.
     * @param pk The PrivateKey used for the RSA decryption.
     * @throws InvalidKeyException If the key is not valid
     */
    public static CipherOutputStream decryptOutputStreamRSA(
        OutputStream out, PrivateKey pk) throws InvalidKeyException{
        if (out == null) {
            throw new IllegalArgumentException("Cannot decrypt a null
                stream");
        }
        if (pk == null){
            throw new IllegalArgumentException("Cannot decrypt with a null
                 key");
        }
        try{
            java.security.Security.addProvider(new org.bouncycastle.jce.
                provider.BouncyCastleProvider());
            Cipher cipher = Cipher.getInstance("RSA", "BC");
            cipher.init(Cipher.DECRYPT_MODE, pk);
            return new CipherOutputStream(out, cipher);
        }
        catch (InvalidKeyException e){
```

```
      throw new InvalidKeyException("[ParipariSecurity]: Invalid key
          ");
    }
    catch(GeneralSecurityException e){
      // unsupported algorithm or bad padding
      //this should not happen...
      System.out.println("[ParipariSecurity]: Unsupported algorithm
          or bad padding "+e.getMessage());
      return null;
    }
  }

  /**
   * Generates a SecretKey to be used in an RC4 encryption algorithm
   * @param keysize The keysize
   */
  public static SecretKey generateRC4Key(int keysize){
    if (keysize == 0)
      throw new IllegalArgumentException("[ParipariSecurity]: Cannot
          generate a zero-length key.");
    try{
      KeyGenerator keygen = KeyGenerator.getInstance("ARCFOUR");
      keygen.init(keysize);
      return keygen.generateKey();
    }
    catch (NoSuchAlgorithmException e){
      System.out.println("[ParipariSecurity] Unsupported Algorithm
          (? this should never happen) "+e.getMessage());
      return null;
    }

  }

  /**
   * Encrypts the block with the given key, using RC4 algorithm.
   * @param in the block
```

```
 * @param k the key
 * @throws InvalidKeyException if the key is invalid
 */

public static byte[] encryptRC4(byte[] in, SecretKey k) throws
    InvalidKeyException{
  if (k == null){
    throw new IllegalArgumentException("Cannot encrypt with a null
        key");
  }
  if (in == null){
    throw new IllegalArgumentException("Cannot encrypt a null
        block");
  }
  try{
    Cipher cipher = Cipher.getInstance("ARCFOUR");
    cipher.init(Cipher.ENCRYPT_MODE, k);
    return cipher.doFinal(in);
  }
  catch (InvalidKeyException e){
    throw new InvalidKeyException("[ParipariSecurity]: Invalid key
        ");
  }
  catch(GeneralSecurityException e){
    // unsupported algorithm or bad padding
    //this should not happen...
    System.out.println("[ParipariSecurity]: Unsupported algorithm
        or bad padding "+e.getMessage());
    return null;
  }
}

/**
 * Decrypts the block with the given key, using RC4 algorithm.
 * @param in the block
 * @param k the key
```

```
     * @throws InvalidKeyException if the key is invalid
     */

    public static byte[] decryptRC4(byte[] in, SecretKey k) throws
        InvalidKeyException{
      if (k == null){
        throw new IllegalArgumentException("Cannot decrypt with a null
            key");
      }
      if (in == null){
        throw new IllegalArgumentException("Cannot decrypt a null
            block");
      }
      try{
        Cipher cipher = Cipher.getInstance("ARCFOUR");
        cipher.init(Cipher.DECRYPT_MODE, k);
        return cipher.doFinal(in);
      }
      catch (InvalidKeyException e){
        throw new InvalidKeyException("[ParipariSecurity]: Invalid key
            ");
      }
      catch(GeneralSecurityException e){
        // unsupported algorithm or bad padding
        //this should not happen...
        System.out.println("[ParipariSecurity]: Unsupported algorithm
            or bad padding "+e.getMessage());
        return null;
      }
    }

}
```

A.3 ParipariSecurityTest.java

```
package Security;
```

```
import Security.ParipariSecurity;
import Security.HashStandards;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.security.GeneralSecurityException;
import java.security.InvalidKeyException;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.RSAPrivateKeySpec;
import java.security.spec.RSAPublicKeySpec;
import java.util.Arrays;

import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;

import org.bouncycastle.jce.provider.JDKKeyPairGenerator;
import org.bouncycastle.jce.provider.JDKMessageDigest;

import org.junit.*;
```

```
import static org.junit.Assert.*;

/**
 * This class provides tests for the methods in ParipariSecurity
 * @author Paolo Marchezzolo
 *
 */

//this is not completed yet
public class ParipariSecurityTest {

  private final int RSA_TEST_KEYSIZE = 384;
  private final int RC4_TEST_KEYSIZE = 128;
  private final int TEST_ARRAYLENGTH = 20;

  @Before
  public void setUp(){

  }

  /**
   * Method for testing null byte block hash; must return null
   * because we pass a null block
   */
  @Test
  public void testHashNullInput() throws NoSuchAlgorithmException{

    byte[] temp = null;
    ParipariSecurity.getHash(temp, HashStandards.MD5);
    fail("Should throw an exception when called with a null block");
  }

  /**
   * Method for testing byte block hash; must return null
   * because we pass a null algorithm
   * */
```

```
@Test
public void testHashNullAlgorithm() throws
    NoSuchAlgorithmException{
  //generating a random byte[20] to be processed
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);

  HashStandards temp = null;
  ParipariSecurity.getHash(block, temp);
  fail("Should throw an exception when called with a null block");
}

/**
 * Method to test the correct generation of hashes
 */
@Test
public void testCorrectHash() throws NoSuchAlgorithmException{
  //generating a random byte[20] to be processed
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);

  //retrieving MD5 Hash and testing its correctness
  byte[] correctHash = MessageDigest.getInstance("MD5").digest(
      block);
  byte[] retrievedHash = ParipariSecurity.getMD5Hash(block);
  assertTrue("Must return the correct hash", Arrays.equals(
      correctHash, retrievedHash));

  //retrieving MD4 Hash and testing its correctness
  JDKMessageDigest.MD4 messagedigest = new JDKMessageDigest.MD4();
  correctHash = messagedigest.digest(block);
  retrievedHash = ParipariSecurity.getMD4Hash(block);
```

```
        assertTrue("Must return the correct hash", Arrays.equals(
            correctHash, retrievedHash));

        //retrieving SHA1 Hash and testing its correctness
        correctHash = MessageDigest.getInstance("SHA-1").digest(block);
        retrievedHash = ParipariSecurity.getSHA1Hash(block);
        assertTrue("Must return the correct hash", Arrays.equals(
            correctHash, retrievedHash));

        //retrieving SHA256 Hash and testing its correctness
        correctHash = MessageDigest.getInstance("SHA-256").digest(block)
            ;
        retrievedHash = ParipariSecurity.getHash(block, HashStandards.
            SHA256);
        assertTrue("Must return the correct hash", Arrays.equals(
            correctHash, retrievedHash));

        //retrieving SHA384 Hash and testing its correctness
        correctHash = MessageDigest.getInstance("SHA-384").digest(block)
            ;
        retrievedHash = ParipariSecurity.getHash(block, HashStandards.
            SHA384);
        assertTrue("Must return the correct hash", Arrays.equals(
            correctHash, retrievedHash));

        //retrieving SHA512 Hash and testing its correctness
        correctHash = MessageDigest.getInstance("SHA-512").digest(block)
            ;
        retrievedHash = ParipariSecurity.getHash(block, HashStandards.
            SHA512);
        assertTrue("Must return the correct hash", Arrays.equals(
            correctHash, retrievedHash));

    }

    /**
```

```
 * Testing the consistency of hashes: two different blocks must
    have
 * different hashes.
 */

@Test
public void testHashConsistency() throws NoSuchAlgorithmException{
  //generating a random byte[20] to be processed; the first byte
      is forced to 0
  //so we can alter the array afterwards
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block1 = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block1);
  block1[0] = 0;

  //generating a second random byte[TEST_ARRAYLENGTH]; this time,
      the first byte is set to 1
  byte[] block2 = new byte[TEST_ARRAYLENGTH];
  System.arraycopy(block1, 0, block2, 0, block1.length);
  block2[0] = 1;

  //retrieving MD5 hashes of the two arrays; they must be
      different
  byte[] hash1 = ParipariSecurity.getMD5Hash(block1);
  byte[] hash2 = ParipariSecurity.getMD5Hash(block2);
  assertFalse("Different inputs must result in different hashes",
      Arrays.equals(block1, block2));

  //retrieving MD4 hashes of the two arrays; they must be
      different
  hash1 = ParipariSecurity.getMD4Hash(block1);
  hash2 = ParipariSecurity.getMD4Hash(block2);
  assertFalse("Different inputs must result in different hashes",
      Arrays.equals(block1, block2));
```

```
        //retrieving SHA1 hashes of the two arrays; they must be
            different
        hash1 = ParipariSecurity.getSHA1Hash(block1);
        hash2 = ParipariSecurity.getSHA1Hash(block2);
        assertFalse("Different inputs must result in different hashes",
            Arrays.equals(block1, block2));

        //retrieving SHA256 hashes of the two arrays; they must be
            different
        hash1 = ParipariSecurity.getHash(block1, HashStandards.SHA256);
        hash2 = ParipariSecurity.getHash(block2, HashStandards.SHA256);
        assertFalse("Different inputs must result in different hashes",
            Arrays.equals(block1, block2));

        //retrieving SHA384 hashes of the two arrays; they must be
            different
        hash1 = ParipariSecurity.getHash(block1, HashStandards.SHA384);
        hash2 = ParipariSecurity.getHash(block2, HashStandards.SHA384);
        assertFalse("Different inputs must result in different hashes",
            Arrays.equals(block1, block2));

        //retrieving SHA512 hashes of the two arrays; they must be
            different
        hash1 = ParipariSecurity.getHash(block1, HashStandards.SHA512);
        hash2 = ParipariSecurity.getHash(block2, HashStandards.SHA512);
        assertFalse("Different inputs must result in different hashes",
            Arrays.equals(block1, block2));

    }
    /**
     * Trying to generate a RSA key with 0 length should throw an
        exception
     */

    @Test
    public void testRSAzeroLengthKey(){
```

```
    KeyPair p = ParipariSecurity.generateRSAKeys(0);
    fail("Fail to throw an exception when trying to generate zero-
        length keys");
}

/**
 * Trying to generate a key pair with a valid key length.
 * Should return a not null value.
 */
@Test
public void testRSAnotNullKey(){
    assertNotNull("Must return a not null key", ParipariSecurity.
        generateRSAKeys(RSA_TEST_KEYSIZE));
}

/**
 * Trying to generate a RSA key with null modulus should throw an
     exception
 */

@Test
public void testRSAnullModPublicKey() throws
    InvalidKeySpecException, NoSuchAlgorithmException{
    java.security.Security.addProvider(new org.bouncycastle.jce.
        provider.BouncyCastleProvider());

    JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
    keygen.initialize(RSA_TEST_KEYSIZE);
    KeyPair p = keygen.genKeyPair();

    RSAPublicKeySpec spec = KeyFactory.getInstance("RSA").getKeySpec
        (p.getPublic(), RSAPublicKeySpec.class);
    PublicKey x = ParipariSecurity.generateRSAPublicKey(null, spec.
        getPublicExponent());
    fail("Fail to throw an exception when trying to generate key
        with null modulus");
```

```
}

/**
 * Trying to generate a RSA key with null exponent should throw an
     exception
 */

@Test
public void testRSAnullExpPublicKey() throws
    InvalidKeySpecException, NoSuchAlgorithmException{
  java.security.Security.addProvider(new org.bouncycastle.jce.
      provider.BouncyCastleProvider());

  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  RSAPublicKeySpec spec = KeyFactory.getInstance("RSA").getKeySpec
      (p.getPublic(), RSAPublicKeySpec.class);
  PublicKey x = ParipariSecurity.generateRSAPublicKey(spec.
      getModulus(), null);
  fail("Fail to throw an exception when trying to generate key
      with null exponent");
}

/**
 * Trying to generate a key with a valid modulus and exponent.
 * Should return a not null value.
 */
@Test
public void testRSAnotNullPublicKey() throws
    InvalidKeySpecException, NoSuchAlgorithmException{
  java.security.Security.addProvider(new org.bouncycastle.jce.
      provider.BouncyCastleProvider());

  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
```

```
    keygen.initialize(RSA_TEST_KEYSIZE);
    KeyPair p = keygen.genKeyPair();

    RSAPublicKeySpec spec = KeyFactory.getInstance("RSA").getKeySpec
        (p.getPublic(), RSAPublicKeySpec.class);
    assertNotNull("Must return a not null key", ParipariSecurity.
        generateRSAPublicKey(spec.getModulus(),spec.getPublicExponent
        ()));
}

/**
 * Trying to generate a RSA key with null modulus should throw an
     exception
 */

@Test
public void testRSAnullModPrivateKey() throws
    InvalidKeySpecException, NoSuchAlgorithmException{
    java.security.Security.addProvider(new org.bouncycastle.jce.
        provider.BouncyCastleProvider());

    JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
    keygen.initialize(RSA_TEST_KEYSIZE);
    KeyPair p = keygen.genKeyPair();

    RSAPrivateKeySpec spec = KeyFactory.getInstance("RSA").
        getKeySpec(p.getPrivate(), RSAPrivateKeySpec.class);
    PrivateKey x = ParipariSecurity.generateRSAPrivateKey(null, spec
        .getPrivateExponent());
    fail("Fail to throw an exception when trying to generate key
        with null modulus");
}

/**
 * Trying to generate a RSA key with null exponent should throw an
     exception
```

```
 */

@Test
public void testRSAnullExpPrivateKey() throws
    InvalidKeySpecException, NoSuchAlgorithmException{
  java.security.Security.addProvider(new org.bouncycastle.jce.
      provider.BouncyCastleProvider());

  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  RSAPrivateKeySpec spec = KeyFactory.getInstance("RSA").
      getKeySpec(p.getPrivate(), RSAPrivateKeySpec.class);
  PrivateKey x = ParipariSecurity.generateRSAPrivateKey(spec.
      getModulus(), null);
  fail("Fail to throw an exception when trying to generate key
      with null exponent");
}

/**
 * Trying to generate a key with a valid modulus and exponent.
 * Should return a not null value.
 */
@Test
public void testRSAnotNullPrivateKey() throws
    InvalidKeySpecException, NoSuchAlgorithmException{
  java.security.Security.addProvider(new org.bouncycastle.jce.
      provider.BouncyCastleProvider());

  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  RSAPrivateKeySpec spec = KeyFactory.getInstance("RSA").
      getKeySpec(p.getPrivate(), RSAPrivateKeySpec.class);
```

```
    assertNotNull("Must return a not null key", ParipariSecurity.
        generateRSAPublicKey(spec.getModulus(),spec.
        getPrivateExponent()));
}

/**
 * Testing the encryption of a block with a null key;
 * must return null.
 */
@Test
public void testRSAencryptNullKey() throws InvalidKeyException {
    //generating a random input
    SecureRandom random = new SecureRandom();
    random.setSeed(System.currentTimeMillis());
    byte[] block = new byte[TEST_ARRAYLENGTH];
    random.nextBytes(block);

    ParipariSecurity.encryptRSA(block, null);
    fail ("Should throw an exception when called with a null key");
}

/**
 * Testing the encryption of a null block; must return null.
 * @throws NoSuchAlgorithmException
 */
@Test
public void testRSAencryptNull() throws NoSuchAlgorithmException,
    InvalidKeyException{
    //generating keys
    JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
    keygen.initialize(RSA_TEST_KEYSIZE);
    KeyPair p = keygen.genKeyPair();

    ParipariSecurity.encryptRSA(null, p.getPublic());
    fail ("Should throw an exception when called with a null block")
        ;
```

```
    }

  /**
   * Testing the correct encryption of the block
   * @throws GeneralSecurityException
   * @throws NoSuchAlgorithmException
   */
  @Test
  public void testRSAencrypt() throws NoSuchAlgorithmException,
      GeneralSecurityException{
    //generating a random input
    SecureRandom random = new SecureRandom();
    random.setSeed(System.currentTimeMillis());
    byte[] block = new byte[TEST_ARRAYLENGTH];
    random.nextBytes(block);

    //generating keys
    JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
    keygen.initialize(RSA_TEST_KEYSIZE);
    KeyPair p = keygen.genKeyPair();


    //encrypting the block
    byte[] encrypted = ParipariSecurity.encryptRSA(block, p.
        getPublic());

    //decrypting
    java.security.Security.addProvider(new org.bouncycastle.jce.
        provider.BouncyCastleProvider());
    Cipher cipher = Cipher.getInstance("RSA", "BC");
    cipher.init(Cipher.DECRYPT_MODE, p.getPrivate());
    byte[] decrypted = cipher.doFinal(encrypted);

    assertTrue("Decrypted block must be equal to the initial one",
        Arrays.equals(decrypted, block));
  }
```

```
/**
 * Testing the decryption of a block with a null key;
 * must return null.
 */
@Test
public void testRSAdecryptNullKey() throws InvalidKeyException{
  //generating a random input
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);

  ParipariSecurity.decryptRSA(block, null);
  fail ("Should throw an exception when called with a null key");
}

/**
 * Testing the decryption of a null block; must return null.
 */
@Test
public void testRSAdecryptNull() throws GeneralSecurityException{
  //generating keys
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  ParipariSecurity.decryptRSA(null, p.getPrivate());
  fail ("Should throw an exception when called with a null block")
      ;

}

/**
 * Testing the correct decryption of the block
 * @throws InvalidKeyException
```

```
     * @throws GeneralSecurityException
     */
    @Test
    public void testRSAdecrypt() throws InvalidKeyException,
        GeneralSecurityException{
      //generating a random input
      SecureRandom random = new SecureRandom();
      random.setSeed(System.currentTimeMillis());
      byte[] block = new byte[TEST_ARRAYLENGTH];
      random.nextBytes(block);

      //generating keys
      JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
      keygen.initialize(RSA_TEST_KEYSIZE);
      KeyPair p = keygen.genKeyPair();

      //encrypting the block
      java.security.Security.addProvider(new org.bouncycastle.jce.
          provider.BouncyCastleProvider());
      Cipher cipher = Cipher.getInstance("RSA", "BC");
      cipher.init(Cipher.ENCRYPT_MODE, p.getPublic());
      byte[] encrypted = cipher.doFinal(block);

      //decrypting
      byte[] decrypted = ParipariSecurity.decryptRSA(encrypted, p.
          getPrivate());

      assertTrue("Decrypted block must be equal to the initial one",
          Arrays.equals(decrypted, block));
    }

    /**
     * Testing the encryption of a stream with a null key; must return
         null.
     */
    @Test
```

```
public void testRSAencryptInputStreamNullKey() throws
    InvalidKeyException{
  //generating a random input
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);
  InputStream stream = new ByteArrayInputStream(block);

  ParipariSecurity.encryptInputStreamRSA(stream, null);
  fail("Should throw an exception when called on a null key");
}

/**
 * Testing the encryption of a null stream; must return null.
 */
@Test
public void testRSAencryptInputStreamNull() throws
    InvalidKeyException {
  //generating keys
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  ParipariSecurity.encryptInputStreamRSA(null, p.getPublic());
  fail("Should throw an exception when called on a null stream");
}

/**
 * Testing the correct encryption of the stream by
    encryptInputStreamRSA
 * @throws GeneralSecurityException
 * @throws IOException
 *
 */
@Test
```

```
public void testRSAencryptInputStream() throws
    GeneralSecurityException, IOException{
  //to be checked

  //generating keys
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  //generating a random input
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);

  //generating the stream to be tested
  byte[] blockcopy = new byte[TEST_ARRAYLENGTH];
  System.arraycopy(block, 0, blockcopy, 0, block.length);
  CipherInputStream encryptStream = ParipariSecurity.
      encryptInputStreamRSA(new ByteArrayInputStream(block), p.
      getPublic());

  //generating the decrypt stream
  java.security.Security.addProvider(new org.bouncycastle.jce.
      provider.BouncyCastleProvider());
  Cipher cipher = Cipher.getInstance("RSA", "BC");
  cipher.init(Cipher.DECRYPT_MODE, p.getPrivate());
  CipherInputStream decryptStream = new CipherInputStream(
      encryptStream, cipher);

  //verifying that the decrypted stream is equal to the original
      one
  int count = 0;
  byte current = 0;
  while (decryptStream.available()>0)
  {
```

```
      current = (byte) decryptStream.read(); //????
      assertEquals("Decrypted stream must be equal to the original
          one", current, blockcopy[count]);
      count++;
    }


}

/**
 * Testing the decryption of a stream with a null key; must return
     null.
 */
@Test
public void testRSAdecryptInputStreamNullKey() throws
    InvalidKeyException{
  //generating a random input
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);
  InputStream stream = new ByteArrayInputStream(block);

  ParipariSecurity.decryptInputStreamRSA(stream, null);
  fail("Should return null when called on a null key");
}

/**
 * Testing the decryption of a null stream; must return null.
 */
@Test
public void testRSAdecryptInputStreamNull() throws
    InvalidKeyException {
  //generating keys
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
```

```
    KeyPair p = keygen.genKeyPair();

    ParipariSecurity.decryptInputStreamRSA(null, p.getPrivate());
    fail ("Should return null when called on a null stream");
  }

  /**
   * Testing the correct decryption of the stream by
       decryptInputStreamRSA
   * @throws GeneralSecurityException
   * @throws IOException
   *
   */
  @Test
  public void testRSAdecryptInputStream() throws
      GeneralSecurityException, IOException{
    //to be checked

    //generating keys
    JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
    keygen.initialize(RSA_TEST_KEYSIZE);
    KeyPair p = keygen.genKeyPair();

    //generating a random input
    SecureRandom random = new SecureRandom();
    random.setSeed(System.currentTimeMillis());
    byte[] block = new byte[TEST_ARRAYLENGTH];
    random.nextBytes(block);

    //generating the stream to be tested
    byte[] blockcopy = new byte[TEST_ARRAYLENGTH];
    System.arraycopy(block, 0, blockcopy, 0, block.length);

    java.security.Security.addProvider(new org.bouncycastle.jce.
        provider.BouncyCastleProvider());
    Cipher cipher = Cipher.getInstance("RSA", "BC");
```

```
    cipher.init(Cipher.ENCRYPT_MODE, p.getPublic());
    CipherInputStream encryptStream = new CipherInputStream(new
        ByteArrayInputStream(block), cipher);

    //generating the decrypt stream
    CipherInputStream decryptStream = ParipariSecurity.
        decryptInputStreamRSA(encryptStream, p.getPrivate());

    //verifying that the decrypted stream is equal to the original
        one
    int count = 0;
    byte current = 0;
    while (decryptStream.available()>0)
    {
      current = (byte) decryptStream.read(); //?????
      assertEquals("Decrypted stream must be equal to the original
          one", current, blockcopy[count]);
      count++;
    }


}

/**
 * Testing the encryption of a stream with a null key; must return
     null.
 */
@Test
public void testRSAencryptOutputStreamNullKey() throws
    InvalidKeyException{

    OutputStream stream = new ByteArrayOutputStream();

    ParipariSecurity.encryptOutputStreamRSA(stream, null);
    fail("Should throw an exception when called with a null key");
}
```

```
/**
 * Testing the encryption of a null stream; must return null.
 */
@Test
public void testRSAencryptOutputStreamNull() throws
    InvalidKeyException {
  //generating keys
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  ParipariSecurity.encryptOutputStreamRSA(null, p.getPublic());
  fail("Should throw an exception when called on a null block");
}

/**
 * Testing the correct encryption of the stream by
    encryptOutputStreamRSA
 *
 */
@Test
public void testRSAencryptOutputStream() throws
    InvalidKeyException, NoSuchAlgorithmException,
    NoSuchProviderException, NoSuchPaddingException, IOException {
  //generating keys
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  //generating a random input
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);
```

```
    //creating the output stream
    ByteArrayOutputStream result = new ByteArrayOutputStream();

    //decrypting the stream

    java.security.Security.addProvider(new org.bouncycastle.jce.
        provider.BouncyCastleProvider());
    Cipher cipher = Cipher.getInstance("RSA", "BC");
    cipher.init(Cipher.DECRYPT_MODE, p.getPrivate());
    CipherOutputStream decryptedStream = new CipherOutputStream(
        result, cipher);

    //encrypting it (when we call write, the data is first encrypted
    //by encryptOutputStreamRSA, then decrypted by the
        CipherOutputStream
    //and eventually written on the ByteArrayOutputStream)
    CipherOutputStream toEncrypt = ParipariSecurity.
        encryptOutputStreamRSA(decryptedStream, p.getPublic());

    //verifying that the decrypted stream is equal to the original
        one
    for (byte b: block)
    {
      toEncrypt.write(b);
    }
    toEncrypt.close();    //<--we need to close the stream for it to
        properly flush

    assertTrue("Decrypted stream must be equal to the original one",
        Arrays.equals(result.toByteArray(), block));

}

/**
 * Testing the encryption of a stream with a null key; must return
     null.
```

```
 */
@Test
public void testRSAdecryptOutputStreamNullKey() throws
    InvalidKeyException{

  OutputStream stream = new ByteArrayOutputStream();

  ParipariSecurity.decryptOutputStreamRSA(stream, null);
  fail("Should throw an exception when called with a null key");

}

/**
 * Testing the encryption of a null stream; must return null.
 */
@Test
public void testRSAdecryptOutputStreamNull() throws
    InvalidKeyException {
  //generating keys
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  ParipariSecurity.decryptOutputStreamRSA(null, p.getPrivate());
  fail("Should throw an exception when called with a null block");
}



/**
 * Testing the correct decryption of the stream by
    decryptOutputStreamRSA
 * @throws GeneralSecurityException
 * @throws IOException
 *
 */
@Test
```

```
public void testRSAdecryptOutputStream() throws
    GeneralSecurityException, IOException{
  //generating keys
  JDKKeyPairGenerator.RSA keygen = new JDKKeyPairGenerator.RSA();
  keygen.initialize(RSA_TEST_KEYSIZE);
  KeyPair p = keygen.genKeyPair();

  //generating a random input
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);

  //creating the output stream
  ByteArrayOutputStream result = new ByteArrayOutputStream();

  //decrypting the stream
  CipherOutputStream decryptedStream = ParipariSecurity.
      decryptOutputStreamRSA(result, p.getPrivate());

  //encrypting it (when we call write, the data is first encrypted
  //by the CipherOutputStream, then decrypted by
      decryptOutputStreamRSA
  //and eventually written on the ByteArrayOutputStream)
  java.security.Security.addProvider(new org.bouncycastle.jce.
      provider.BouncyCastleProvider());
  Cipher cipher = Cipher.getInstance("RSA", "BC");
  cipher.init(Cipher.ENCRYPT_MODE, p.getPublic());
  CipherOutputStream toEncrypt = new CipherOutputStream(
      decryptedStream, cipher);

  //verifying that the decrypted stream is equal to the original
      one
  for (byte b: block)
  {
    toEncrypt.write(b);
```

```
    }
    toEncrypt.close(); //<--we need to close the stream for it to
        properly flush

    assertTrue("Decrypted stream must be equal to the original one",
        Arrays.equals(result.toByteArray(), block));

  }



  /**
   * Trying to generate a RC4 key with 0 length should throw an
      exception
   */

  @Test
  public void testRC4zeroLengthKey(){
    SecretKey p = ParipariSecurity.generateRC4Key(0);
    fail("Fail to throw an exception when trying to generate zero-
        length keys");
  }

  /**
   * Trying to generate a key with a valid key length.
   * Should return a not null value.
   */
  @Test
  public void testRC4notNullKey(){
    assertNotNull("Must return a not null key", ParipariSecurity.
        generateRC4Key(RC4_TEST_KEYSIZE));
  }

  /**
   * Testing the encryption of a block with a null key;
   * must return null.
```

```
 */
@Test
public void testRC4encryptNullKey() throws InvalidKeyException{
  //generating a random input
  SecureRandom random = new SecureRandom();
  random.setSeed(System.currentTimeMillis());
  byte[] block = new byte[TEST_ARRAYLENGTH];
  random.nextBytes(block);

  ParipariSecurity.encryptRC4(block, null);
  fail("Should throw an exception when called with a null key");
}

/**
 * Testing the encryption of a null block; must return null.
 * @throws GeneralSecurityException
 */
@Test
public void testRC4encryptNull() throws GeneralSecurityException{
  //generating keys
  KeyGenerator k = KeyGenerator.getInstance("ARCFOUR");
  k.init(RC4_TEST_KEYSIZE);
  SecretKey key = k.generateKey();

  ParipariSecurity.encryptRC4(null, key);
  fail("Should throw an exception when called with a null block");
}

/**
 * Testing the correct encryption of the block
 * @throws GeneralSecurityException
 */
@Test
public void testRC4encrypt() throws GeneralSecurityException{
```

```
    //generating a random input
    SecureRandom random = new SecureRandom();
    random.setSeed(System.currentTimeMillis());
    byte[] block = new byte[TEST_ARRAYLENGTH];
    random.nextBytes(block);

    //generating key
    KeyGenerator k = KeyGenerator.getInstance("ARCFOUR");
    k.init(RC4_TEST_KEYSIZE);
    SecretKey key = k.generateKey();

    //encrypting the block
    byte[] encrypted = ParipariSecurity.encryptRC4(block, key);

    //decrypting
    Cipher c = Cipher.getInstance("ARCFOUR");
    c.init(Cipher.DECRYPT_MODE, key);
    byte[] decrypted = c.doFinal(encrypted);

    assertTrue("Decrypted block must be equal to the initial one",
        Arrays.equals(decrypted, block));
}

/**
 * Testing the decryption of a block with a null key;
 * must return null.
 */
@Test
public void testRC4decryptNullKey() throws InvalidKeyException{
    //generating a random input
    SecureRandom random = new SecureRandom();
    random.setSeed(System.currentTimeMillis());
    byte[] block = new byte[TEST_ARRAYLENGTH];
    random.nextBytes(block);

    ParipariSecurity.decryptRC4(block, null);
```

```
    fail("Should throw an exception when called with a null key");
  }


  /**
   * Testing the decryption of a null block; must return null.
   * @throws GeneralSecurityException
   */
  @Test
  public void testRC4decryptNull() throws GeneralSecurityException{
    //generating keys
    KeyGenerator k = KeyGenerator.getInstance("ARCFOUR");
    k.init(RC4_TEST_KEYSIZE);
    SecretKey key = k.generateKey();

    ParipariSecurity.decryptRC4(null, key);
    fail("Should throw an exception when called with a null block");
  }

  /**
   * Testing the correct decryption of the block
   * @throws GeneralSecurityException
   * @throws NoSuchAlgorithmException
   */
  @Test
  public void testRC4decrypt() throws NoSuchAlgorithmException,
      GeneralSecurityException{
    //generating a random input
    SecureRandom random = new SecureRandom();
    random.setSeed(System.currentTimeMillis());
    byte[] block = new byte[TEST_ARRAYLENGTH];
    random.nextBytes(block);

    //generating key
    KeyGenerator k = KeyGenerator.getInstance("ARCFOUR");
    k.init(RC4_TEST_KEYSIZE);
```

```
    SecretKey key = k.generateKey();

    //encrypting the block
    Cipher cipher = Cipher.getInstance("ARCFOUR");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    byte[] encrypted = cipher.doFinal(block);

    //decrypting
    byte[] decrypted = ParipariSecurity.decryptRC4(encrypted, key);

    assertTrue("Decrypted block must be equal to the initial one",
        Arrays.equals(decrypted, block));
  }


}
```

Bibliografia

[1] Paolo Bertasi. *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*. Dipartimento di Ingegneria dell'Informazione, Università di Padova, 2004.

[2] Andrew S. Tanenbaum. *Reti di Calcolatori*. Pearson Paravia Bruno Mondadori, 2003.