Informatikprojekt

Java API für PGP

Thomas Jampen
jampen@iam.unibe.ch

Institut für Informatik und angewandte Mathematik
Universität Bern
Neubrückstrasse 10
CH-3012 Bern
Schweiz

19. April 2001

Zusammenfassung
Dieses Dokument beschreibt ein Java API, welches mittels *Java Native Interface (JNI)* auf eine C-Implementation von PGP (**P**retty **G**ood **P**rivacy) zugreift. Dadurch ist es möglich, die Vorteile der schnellen C-Implementation und der einfachen, weitverbreiteten Programmiersprache Java zu nutzen.

Der Quellcode dieses Java APIs sowie kompilierte Versionen für Linux und Windows können von der Cryptography-Homepage [5] heruntergeladen werden. Diese Dokumentation steht ebenfalls dort in diversen Formaten zum Download bereit. Zudem ist der Javadoc-Output online verfügbar.

Inhaltsverzeichnis

Abbildungsverzeichnis **5**

Tabellenverzeichnis **6**

1. Einleitung **7**
- 1.1. Der Vorteil eines PGP-Interfaces für Java . . . 7
- 1.2. Die Grundlagen dieses Projekts . . . 7

2. Übersicht über PGP **8**
- 2.1. Was ist PGP? . . . 8
- 2.2. Grobe Übersicht über PGP . . . 8
- 2.3. Kryptographie in PGP . . . 9
  - 2.3.1. IDEA . . . 9
  - 2.3.2. RSA . . . 10
  - 2.3.3. MD5 . . . 11
- 2.4. Wo ist PGP erhältlich? . . . 11
- 2.5. Plugins und Utilities für PGP . . . 11

3. Übersicht über JNI **12**
- 3.1. Was ist JNI? . . . 12
  - 3.1.1. Wann ist es sinnvoll, JNI einzusetzen? . . . 12
  - 3.1.2. Wie können Native-Methoden genutzt werden? . . . 13
- 3.2. JNI konkret . . . 13

4. Implementation der Schnittstelle **16**
- 4.1. Architektur . . . 16
- 4.2. Verwendete Tools . . . 16
- 4.3. Probleme/Erfahrungen . . . 16
  - 4.3.1. PGP-Bibliotheken und Betriebssysteme . . . 16
  - 4.3.2. JNI . . . 17
- 4.4. Das PGP API (Java Seite) . . . 17
  - 4.4.1. Konstruktoren . . . 18
  - 4.4.2. PGP-Methoden . . . 18
  - 4.4.3. Steuermethoden . . . 18
  - 4.4.4. Statusmethoden . . . 19
  - 4.4.5. Fehlerbehandlung . . . 20
- 4.5. Das PGP API (C Seite) . . . 21
  - 4.5.1. Native Methoden . . . 22

4.5.2. Hilfsmethoden . . . 25

5. Evaluation des Ansatzes **27**
5.1. Leistungsvergleich . . . 27
5.1.1. Skript vs. JNI . . . 27
5.1.2. Die Geschwindigkeit der reinen Java Implementation . . . 28
5.2. Sicherheit . . . 28
5.2.1. Was hat man gewonnen? . . . 28
5.2.2. Welche neuen Lücken sind entstanden? . . . 29
5.3. Offene Aspekte . . . 29
5.3.1. Schlüsselverwaltung . . . 29
5.3.2. Portabilität . . . 29

A. Quellcode des PGP Java API **32**
A.1. Die Java Klasse . . . 32

Literaturverzeichnis **39**

Abbildungsverzeichnis

2.1. Struktur des IDEA-Algorithmus [9] 9

3.1. Das Java Native Interface [11] 12

4.1. Änderungen der C-Implementation 21

Tabellenverzeichnis

4.1. Fehlercodes des Java PGP API. 20
4.2. Bezeichner für Java Typen. 25

5.1. Geschwindigkeitsvergleich: Skript - JNI. 28
5.2. Geschwindigkeit der reinen Java-Implementation. 29

1. Einleitung

1.1. Der Vorteil eines PGP-Interfaces für Java

Warum sollte man etwas programmieren, das schon existiert? Aus diesem und aus folgenden Gründen ist es meiner Ansicht nach nicht sinnvoll, PGP vollständig in Java zu implementieren:

- Java ist wesentlich langsamer als C.
- Im Gegensatz zu C kann der Programmierer in Java nicht in das Speichermanagement eingreifen, d.h. Passwörter und geheime Schlüssel gezielt aus dem Speicher löschen.
- Es ist sehr zeitaufwändig, eine eigene Implementation auf Fehler und Sicherheitslöcher zu testen, während PGP schon sehr lange und zuverlässig als Verschlüsselungssoftware dient.

Somit wäre eine optimale Lösung ein Java API, welches den Zugriff auf eine bestehende PGP Implementation zur Verfügung stellt. Genau das war das Ziel dieses Informatikprojekts.

1.2. Die Grundlagen dieses Projekts

Fürs *CATI-Projekt* [3, 4] implementierte Manuel Günter bereits eine Java-Klasse, welche PGP via Shell-Skripte aufruft, um Nachrichten zu verschlüsseln, signieren und verifizieren. Softwareagenten (serialisierte Java-Objekte) werden als Nachrichten an andere Computer versandt. Dieser Java-Bytecode wird später ausgeführt. Aus diesem Grund ist es wichtig, dass solche Agenten - und auch andere Nachrichten - authentisiert sind. Ausserdem ist eine Verschlüsselung unter Umständen auch angebracht [1].

Uns interessierte vor allem auch, ob die elegantere Lösung - das Java API - Geschwindigkeitsvorteile bietet, oder ob der Shell-Skript Ansatz schneller ist.

Mein API musste vollständig kompatibel sein zu dieser bereits implementierten Java-Klasse, damit man am Schluss - falls dieses Projekt ein Erfolg wird - die beiden Klassen nur noch auszutauschen braucht.

2. Übersicht über PGP

2.1. Was ist PGP?

PGP ist die Abkürzung für **P**retty **G**ood **P**rivacy und heisst auf deutsch *ziemlich gute Privatsphäre.* Hinter diesem Namen versteckt sich ein Software-Produkt, welches Daten verschlüsseln, entschlüsseln und signieren kann. PGP ist im Jahre 1991 von Phil Zimmermann entwickelt worden und wurde seither vom Massachusetts Institute of Technology (MIT), ViaCrypt, PGP Inc. und momentan von Network Associated Inc. (NAI) weiterentwickelt und vertrieben. PGP ist der de-facto Standart für Emailverschlüsselung und wird von Millionen Benutzern weltweit verwendet. PGP kann aber auch zur Verschlüsselung von Dateien, ganzen Dateisystemen und Kommunikationskanälen verwendet werden.

Für privaten Gebrauch ist PGP gratis, der Quellcode ist öffentlich. Somit kann jeder im Code nach Sicherheitslücken und Fehlern suchen und sogar überprüfen, ob irgendwelche Hintertüren (engl. backdoors) eingebaut worden sind. Das erklärt auch PGPs enorme Beliebtheit.

2.2. Grobe Übersicht über PGP

Jeder PGP-Benutzer generiert ein eigenes **Schlüsselpaar**. Dieses Schlüsselpaar (engl. *key pair*) besteht aus einem **privaten** und dem dazu gehörenden **öffentlichen Schlüssel**.

Der **private Schlüssel** - oft auch *private key* genannt - ist geheim und sollte immer geschützt aufbewahrt werden. Falls er in fremde Hände gelangt, ist die Verschlüsselung gebrochen.

Der **öffentliche Schlüssel** - *public key* genannt - jedoch wird an Kommunikationspartner verteilt und ist nicht geheim. Er wird dazu verwendet, eine Nachricht speziell für jemanden zu verschlüsseln. Nur wer den entsprechenden privaten Schlüssel hat, kann diese Nachricht wieder entschlüsseln.

Beispiel 1 (Verschlüsselung) *Falls Alice eine verschlüsselte Nachricht an Bob senden will, benötigt Alice den öffentlichen Schlüssel von Bob. Bob wiederum muss im Besitz seines privaten Schlüssels sein, um die Nachricht lesen zu können.*

Beispiel 2 (Signatur) *Wenn Alice die Nachricht für Bob signieren möchte, kann sie das mit ihrem privaten Schlüssel tun, Bob kann die Signatur mit ihrem öffentlichen Schlüssel überprüfen.*

Selbstverständlich kann man Signatur und Verschlüsselung auch kombinieren, um authentisierbare und verschlüsselte Botschaften herzustellen. Hier wird üblicherweise zuerst signiert und dann erst verschlüsselt, damit erstens ein möglicher Angreifer nicht sehen kann, um wessen Signatur es sich handelt, und zweitens - was viel wichtiger ist - damit die Signatur nicht entfernt werden kann.

2.3. Kryptographie in PGP

PGP verwendet konventionelle symmetrische Algorithmen sowie auch neuere asymmetrische Verfahren. In PGP 2.3 wird *IDEA* als symmetrischer und *RSA* als asymmetrischer Algorithmus verwendet. Ferner werden auch sogenannte sichere Hashfunktionen, wie z.B. *MD5*, verwendet.

2.3.1. IDEA

IDEA heisst **I**nternational **D**ata **E**ncryption **A**lgorithm und wurde an der ETH Zürich in Zusammenarbeit mit der Firma Ascom entwickelt. IDEA muss für kommerzielle Zwecke lizenziert werden, ist aber für private Zwecke gratis [9].

IDEA ist ein 64 bit Block-Cipher, der einen 128 bit Schlüssel verwendet. IDEA arbeitet in 8 Runden mit Operationen aus drei verschiedenen algebraischen Strukturen: bitweises XOR, Addition modulo 2^{16} und Multiplikation modulo $2^{16}+1$ (siehe Abbildung 2.1).

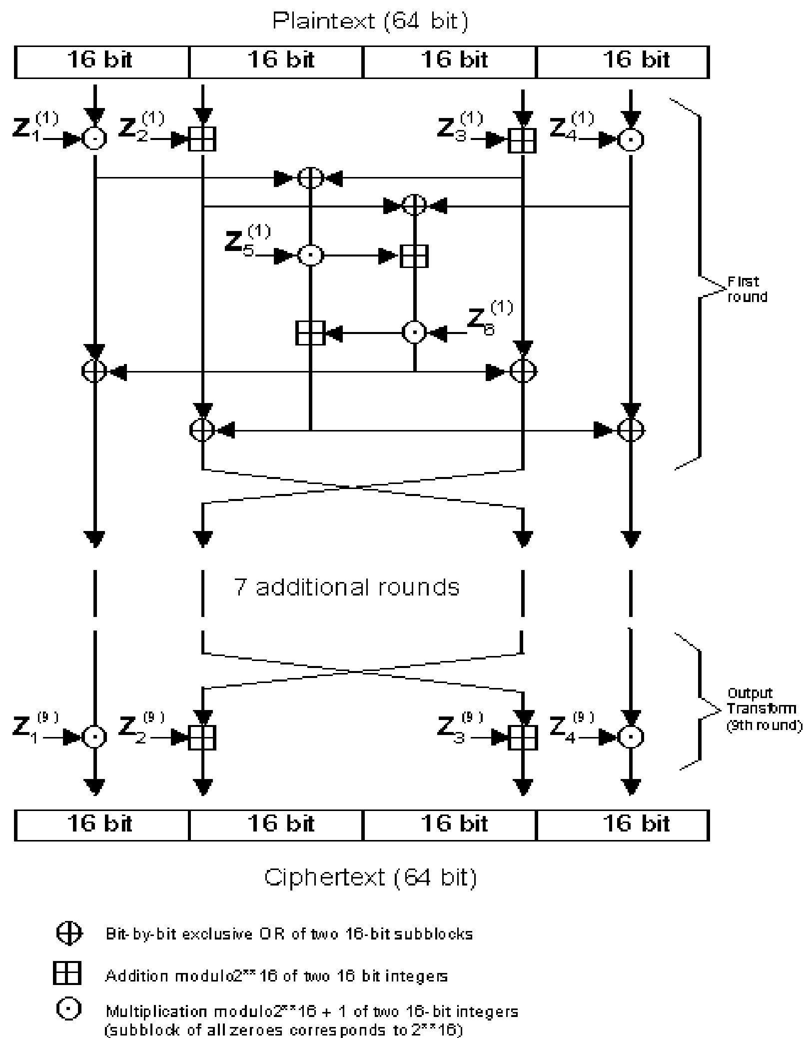

Abbildung 2.1.: Struktur des IDEA-Algorithmus [9]

Symmetrisch bedeutet, dass zum Verschlüsseln und Entschlüsseln derselbe Schlüssel verwendet wird. Das bereitet oft erhebliche Probleme beim Schlüsselaustausch, denn es stellt sich die Frage, wie über einen unsicheren Kanal ein geheimer Schlüssel ausgetauscht werden kann. Zur Lösung dieses Problems wurden die asymmetrischen Algorithmen entwickelt. Sie werden weiter unten genauer unter die Lupe genommen. Wieso werden aber symmetrische Algorithmen immer noch verwendet, wenn asymmetrische das Problem lösen könnten? Die Antwort ist ganz einfach, der Geschwindigkeitsvorteil von symmetrischen Algorithmen gegenüber asymmetrischen beträgt etwa einen Faktor 1000. Aus diesem Grund wird die eigentliche Information mit einem symmetrischen, der dazu gehörende Schlüssel jedoch mit einem asymmetrischen Algorithmus verschlüsselt. Das ist wesentlich sicherer als die herkömmliche symmetrische Verschlüsselung, da statt eines passwort-basierten symmetrischen Schlüssels ein zufälliger generiert werden kann. So ist das Erraten des Passwortes (der symmetrischen Verschlüsselung) nicht mehr effizienter als eine brute-force Attacke (brute-force bedeutet, dass jeder einzelne Schlüssel ausprobiert werden muss, was bei grossen Schlüsselräumen schlicht zu aufwändig ist[1]).

2.3.2. RSA

Der Name **RSA** deutet auf dessen Erfinder Ronald **R**ivest, Adi **S**hamir und Leonard **A**dleman hin. Entstanden ist RSA zwischen 1976 und 1977 am Massachusetts Institute of Technology (MIT). Das Patent lief im September 2000 ab, d.h. RSA kann nun frei verwendet werden.

RSA funktioniert wie folgt:

grosse Primzahlen (geheim)	p, q	
Modulus	m	$m = pq$
Euler'sche ϕ-Funktion	$\phi()$	$\phi(m) = (p-1)(q-1)$
öffentlicher Exponent	e	$ggT(e, \phi(m)) = 1$
privater Exponent	d	$d = e^{-1} \ mod \ \phi(m)$
Nachricht (engl. *plaintext*)	n	
verschlüsselte Nachricht (engl. *ciphertext*)	c	$c = n^e \ mod \ m$
entschlüsselte Nachricht		$n = c^d \ mod \ m$
signierte Nachricht	s	$s = n^d \ mod \ m$
verifizierte Nachricht	v	$v = s^e \ mod \ m$

Der öffentliche Exponent e und der Modulus m bilden zusammen den *öffentlichen Schlüssel*, der private Exponent d den *privaten Schlüssel*. Für weitere Informationen zu RSA, zu den Entwicklern und zu deren Publikationen zu asymmetrischer Verschlüsselung siehe [16].

[1] Bei einer Schlüssellänge von 128-bit gibt es 2^{128} mögliche Schlüssel. Angenommen, jemand besitzt eine Milliarde Prozessoren, welche je eine Milliarde Schlüssel pro Sekunde ausprobieren können, dann bräuchte er immer noch etwa 10^{13} Jahre, um alle Schlüssel zu testen.

2.3.3. MD5

MD5 heisst **M**essage **D**igest und bedeutet soviel wie Kurzfassung einer Nachricht. Dahinter versteckt sich eine mathematische Funktion, welche als Input einen beliebig langen Text nimmt, und als Output einen 128 bit langen Fingerabdruck dieses Textes liefert. Entwickelt wurde dieser Hashalgorithmus im Jahre 1992 von Ronald Rivest am MIT [12].

Die Schwierigkeit, eine solche Hashfunktion zu entwickeln, besteht darin, dass es unmöglich sein soll,

1. zu einer existierenden Nachricht eine weitere zu konstruieren, die denselben Hashwert besitzt wie die ursprüngliche Nachricht.
2. zwei verschiedene Texte zu erzeugen, welche auf denselben Wert gehasht werden.

Doch bis heute ist noch von keiner Hashfunktion bewiesen worden, dass sie diese Eigenschaften besitzt. Das Ziel ist es also, eine sichere Hashfunktion zu finden, für welche es zu aufwändig ist, eine solche sogenannte Kollision zu finden.

2.4. Wo ist PGP erhältlich?

Da starke Kryptographie bis vor kurzem in den Vereinigten Staaten von Amerika unter das Waffenexport-Gesetz fiel, wird im Internet eine internationale Version angeboten[2]. Der Quelltext und die kompilierten Versionen für diverse Betriebssysteme können auf der PGP International Homepage [14] heruntergeladen werden.

2.5. Plugins und Utilities für PGP

Unter Windows gibt es Plugins für Outlook und Outlook Express, Eudora, Microsoft Exchange, Lotus Notes, Calypso und Pegasus Mail. Neben Plugins für Emailprogramme gibt es auch diverse Tools, welche die Arbeit mit PGP erleichtern, so zum Beispiel in der neusten Version ein Plugin für ICQ, das es erlaubt, an Kollegen, die zum selben Zeitpunkt online sind, verschlüsselte Kurznachrichten zu schicken.

Auch unter Unix/Linux existieren Plugins für diverse Mailclients wie zum Beispiel Pine, Mutt, Zmail, Elm und ebenso für den beliebten Allrounder Emacs.

Seit kurzem sind sogar schon PGP Versionen für Handhelds wie Palm und Psion erhältlich. Und selbstverstänlich gibt es auch Versionen für Dos, Amiga, Atari, OS/2 und Mac.

[2] Der Quelletext von PGP wurde in Amerika als Buch gedruckt und konnte in dieser Form legal exportiert werden. In Europa wurde der Quelltext wieder gescannt, angepasst und kompiliert. Die Version 6.5.1 z.B. wurde bei CNLab in der Schweiz gescannt!

3. Übersicht über JNI

3.1. Was ist JNI?

JNI heisst **J**ava **N**ative **I**nterface, wird von Sun Microsystems, Inc. entwickelt und ist eine Weiterentwicklung des im Java Development Kit (JDK) 1.0 eingeführten Native Method Interfaces (NMI). Das JNI ist ab JDK 1.1 verfügbar und gehört seither zu dessen Standartlieferumfang.

Das JNI erlaubt es Java-Code, der in einer virtuellen Maschine läuft, mit Applikationen oder Bibliotheken zu interagieren, welche in C, C++ oder Assembler geschrieben wurden (siehe Abbildung 3.1). Ein grosser Vorteil von JNI ist, dass es keine Einschränkungen bezüglich der Implementation der virtuellen Maschine gibt. Das heisst, dass alles, was in einer VM einer bestimmten Platform läuft, problemlos auch in anderen VM Implementationen derselben Platform laufen wird, sofern diese JNI unterstützt.

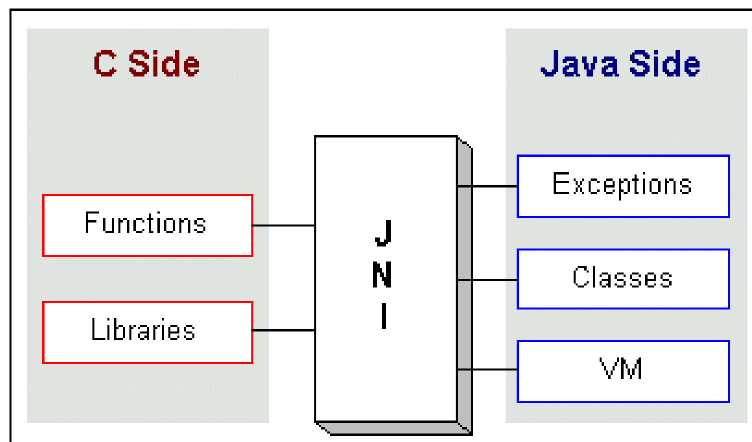

Abbildung 3.1.: Das Java Native Interface [11]

Das JNI ist ein zwei-Weg Interface und unterstützt deshalb zwei Arten von Native-Code: Bibliotheken und Applikationen. Das heisst, neben dem Einsatz von Native-Methoden in Java-Applikationen ist es auch möglich, eine Java VM in einer Native-Anwendung einzubinden. Genauere Informationen dazu findet man bei Sheng [2] oder auf der JNI Homepage [11].

3.1.1. Wann ist es sinnvoll, JNI einzusetzen?

Es gibt Situationen, in denen es nicht möglich ist (oder keinen Sinn macht) eine ganze Applikation in Java zu programmieren. Zum Beispiel könnte das in folgenden Szenarien der Fall sein:

- Java unterstützt bestimmte platformabhängige Features nicht, welche von einer Anwendung benötigt werden.
- Es existiert bereits eine Implementation, welche man gerne einer Java Applikation zugänglich machen würde.
- Eine Anwendung benötigt zeitkritischen Code, der besser nicht in Java implementiert wird (Echtzeit-Andwendungen).
- Das Programm beinhaltet aufwändige Berechnungen, welche in Java langsamer durchgeführt werden als in C.

3.1.2. Wie können Native-Methoden genutzt werden?

In Native-Methoden kann man Java-Objekte (z.B. `Strings`, aber auch `Arrays`) kreieren und verändern, Java-Methoden aufrufen und Java-Objekte zugreifen. Es ist möglich, Klassen zu laden und Klasseninformationen zu erhalten oder Typenprüfungen vorzunehmen. Weiter erlaubt es das JNI, Exceptions abzufangen oder auszulösen. Die meisten der obgenannten Punkte werden in Kapitel 4 ausführlicher behandelt.

3.2. JNI konkret

Ich möchte an dieser Stelle das einfachste aller JNI-Beispiele vorstellen: Ein *Hello World* Programm.

Zuerst einmal die Java-Klasse:

Datei: **Hello.java**

```
public class Hello {

        private native void print();

        static {
                System.loadLibrary("Hello");
        }

        public static void main(String[] args) {
                new Hello().print();
        }
}
```

So kompiliert man die Java-Klasse:

```
zadeh:~$ javac Hello.java
```

Anschliessend generiert man das C-Headerfile `Hello.h` mit folgendem Befehl:

```
zadeh:~$ javah -jni Hello
```

Das automatisch erstellte Headerfile sieht folgendermassen aus:

Datei: **Hello.h**

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Hello */

#ifndef _Included_Hello
#define _Included_Hello
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Hello
 * Method:    print
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Hello_print
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Anhand dieses Files erstellen wir nun den C-Code. Wichtig ist vor allem die Zeile mit der Methodendeklaration. Der Methodenname setzt sich zusammen aus dem Präfix **Java_**, dem vollständigen Klassennamen, einem **Underscore** und dem in Java verwendeten Methodennamen.

Datei: **Hello.c**

```
#include <jni.h>
#include <stdio.h>
#include "Hello.h"

JNIEXPORT void JNICALL
Java_Hello_print(JNIEnv *env, jobject jobj) {
  printf("Hello World!\n");
}
```

Kompiliert wird `Hello.c` schliesslich mit Hilfe der folgenden Anweisung:

```
zadeh:~$ cc -shared -I/path/to/jdk/include \
         -I/path/to/jdk/include/linux Hello.c -o libHello.so
```

So wird also ein *Shared Object* mit dem Namen `libHello.so` erstellt, auf welches die Java Anwendung zugreifen kann.

Gestartet wird das Programm mit:

```
zadeh:~$ java Hello
Hello World!
zadeh:~$
```

Dabei kann es zu einem `UnsatisfieldLinkError` kommen, was darauf hindeutet, dass der `LD_LIBRARY_PATH` nicht richtig gesetzt wurde und deshalb das *Shared Object* nicht gefunden werden konnte.

Der Pfad muss das aktuelle Verzeichnis enthalten. Wer diesen Pfad nicht anpassen möchte, kann das Java-Programm mit der folgenden Option starten, um das Problem zu beheben:

```
zadeh:~$ java -Djava.library.path=. Hello
Hello World!
zadeh:~$
```

4. Implementation der Schnittstelle

4.1. Architektur

Java ist bekanntlich platformunabhängig, C jedoch nicht. Da an der Universität vorwiegend Unix zur Verfügung steht, sollte das Projekt ursprünglich unter Unix laufen.

Im Internet kursiert eine leicht modifizierte Implementation von PGP 2.3a, die **PGPTools** [15]. Die Version 2.3 von PGP ist die letzte, welche unter der GPL (**G**NU **P**ublic **L**icense) vertrieben wird und somit frei in eigenen Programmen verwendet werden kann.

PGPTools liess sich nach Langem unter Unix zwar kompilieren, erzeugte jedoch zur Laufzeit korrupte RSA-Signaturen. Unter Linux kompilierte PGPTools sofort problemlos und funktionierte auch fehlerfrei, weshalb dieses Projekt unter Linux weiterentwickelt wurde.

Alle platformabhängigen Angaben und Vergleiche beziehen sich auf einen Dell Inspiron Laptop, auf dem *Debian GNU/Linux "Woody"* mit Kernel 2.4.2-ac9 installiert ist. Der Laptop hat 256MB RAM und einen Intel Pentium III Prozessor mit 750MHz Taktfrequenz. Kompiliert wurde das Projekt mit dem Java 2 SDK v1.3 [10] und dem Gnu C Compiler gcc v2.95.3 [7].

4.2. Verwendete Tools

Wie oben erwähnt werden **PGPTools** [15] für Linux verwendet, welche auf PGP 2.3a basieren. Um die korrekte Funktionsweise und Kompatibilität zu **PGP 2.6.3 International** [14] zu überprüfen, wurden etliche Output-Dateien von PGPTools mit PGPi entschlüsselt und umgekehrt. Das bedeutet, dass sowohl jede mit PGPTools verschlüsselte oder/und signierte Nachricht mit PGPi entschlüsselt und verifiziert werden kann, als auch umgekehrt jede mit PGPi verschlüsselte oder/und signierte Mitteilung mit PGPTools entschlüsselt und überprüft werden kann.

Ferner können bestehende Schlüsseldateien, `pubring.pgp` und `secring.pgp`, welche mit PGPi verwendet wurden, problemlos auch mit meinem Java API benutzt werden.

4.3. Probleme/Erfahrungen

4.3.1. PGP-Bibliotheken und Betriebssysteme

Probleme eröffneten sich vor allem in der Startphase des Projekts, nämlich während der Entwicklung unter Unix.

Auf der PGP-Homepage [13] ist ersichtlich, dass aktuelle Versionen von PGP zwar für nichtkommerzielle Zwecke frei verfügbar sind, jedoch nicht abgeändert werden dürfen. Deshalb musste für dieses Projekt die Version 2.3 oder eine frühere verwendet werden. Das stellte eigentlich kein Problem dar, denn die Version 2.3 ist, soweit sie hier verwendet wird, kompatibel zu PGP 2.6.3i, der auch heute noch am meisten verbreiteten Version von PGP.

Die Suche nach bereits vorhandenen PGP Bibliotheken lieferte zwei Resultate: *PGPlib* und *PGPTools*. PGPlib liess sich unter Unix nicht kompilieren. Bei PGPTools eröffneten sich ebenfalls Probleme beim Kompilieren. Schliesslich schloss dieser Vorganz zwar ohne Fehlermeldung ab, leider traten aber zur Laufzeit Fehler auf. Nicht, dass PGPTools abgestürzt wäre oder sich unerwartet beendet hätte. Nein, es schien zu funktionieren, genauere Überprüfungen ergaben aber, dass PGPi die von PGPTools erzeugten, verschlüsselten und signierten Nachrichten nicht entschlüsseln konnte, da PGPTools korrupte RSA-Signaturen zu generieren schien.

Unter Linux konnte PGPTools sofort erfolgreich kompiliert werden. Hier traten - im Gegensatz zu Unix - keine korrupten RSA Signaturen auf. Aus diesem Grund wurde dieses Java API unter Linux weiter entwickelt.

Die weiteren Probleme waren eher kleinerer Natur, zum Beispiel herauszufinden, welche Änderungen des *Makefiles* erforderlich waren, um PGPTools nicht als Anwendung, sondern als *Shared Object* mit JNI-Unterstützung zu kompilieren.

4.3.2. JNI

Das JNI ist sehr gut verständlich und bereitet keine grossen Schwierigkeiten. Die Implementation einfacher Testprogramme, welche verdeutlichten, wie Variablen und Objekte übergeben und wie Rückgabewerte zurück an Java geliefert werden, erleichterte den Einstieg. Dabei trat das Problem auf, dass je nach Compiler die von Sun vorgeschlagene Option **-G** beim Kompilieren der C-Dateien nicht funktionierte. Anhand der Man-Pages stellte sich heraus, dass besser die Option **-shared** verwendet wird, da dies beim *cc* sowie beim *gcc* auf Unix und Linux funktioniert.

Nun galt es, die bestehende Implementation von PGPTools abzuändern und mittels Java API aufzurufen. Hier traten eigentlich nur kleinere Probleme auf, welche nach eingehendem Studium der JNI Dokumentation [2, 11] und einer Einarbeitungsphase in C gut gelöst werden konnten.

Eine etwas schwierigere Angelegenheit war, ein Java String-Array aus dem C-Code an Java zurück zu geben. Warum das notwendig war, und wie die Implementation aussieht, wird im Kapitel 4.5.1 genauer behandelt. Auch hier halfen Literatur und Newsgroups weiter.

4.4. Das PGP API (Java Seite)

Das PGP API ist in der Klasse `PGPi` implementiert worden, sie stellt ein Interface zwischen Java-Programmen und der C-Bibliothek dar. `PGPi` bietet fünf verschiedene Arten von Methoden an:

- Konstruktoren
- PGP-Methoden
- Steuermethoden
- Statusmethoden
- Fehlerbehandlung

4.4.1. Konstruktoren

Es gibt zwei Konstruktoren, welche zur Objekterzeugung aufgerufen werden können.

public **PGPi**();

public **PGPi**(boolean *armor*, boolean *zip*, boolean *text*);

Der Standartkonstruktor setzt die Membervariablen so, dass die Ausführungszeit minimiert wird. Der zweite Konstruktor erlaubt eine gezielte Zuweisung. Für eine Beschreibung dieser Parameter wird auf 4.4.3 verwiesen.

4.4.2. PGP-Methoden

Zu dieser Gruppe von Methoden gehören die Native-Methoden. Sie rufen die PGP-Bibliothek auf und führen die kryptographischen Aktionen sowie das Schlüsselmanagement durch.

public native int **showKeys**(boolean *sigson*);

public native int **encrypt**(String *uid*,String *infile*);

public native int **sign**(String *uid*, String *infile*, String *passphrase*);

public native int **encryptAndSign**(String *suid*, String *euid*, String *infile*, String *passphrase*);

private native String[] **decryptc**(String *infile*, String *passphrase*);

public int **decrypt**(String *infile*, String *passphrase*);

public native int **deleteKey**(String *uid*);

public native int **addPubKey**(String *infile*);

public native int **signKey**(String *toBeUid*, String *suid*, String *passphrase*);

public native int **generateKey**(String *ran*, int *length*, String *uid*, String *passphrase*);

public native int **keyMaint**();

public native int **extractKey**(String *uid*, String *outfile*);

public native int **examine**(String *infile*);

Alle Methoden ausser `decrypt` sind Native-Methoden, das heisst, sie sind im C-Code implementiert. `decrypt` ist eine Ausnahme, da PGP nach erfolgter Entschlüsselung nicht nur einen `int` zurück gibt, sondern neben diesem Fehlercode noch weitere Informationen liefert, welche mit den Statusmethoden abgefragt werden können. Deshalb wird zusätzlich noch die Methode `decryptc` benötiget, welche den Zugriff auf den C-Code realisiert.

In `decrypt` werden die Rückgabewerte von `decryptc`, vier Elemente eines String-Arrays, analysiert, und die Membervariablen `sigOK`, `wasEncrypted` und `signer` entsprechend gesetzt. Der Fehlercode (siehe Kapitel 4.4.5) wird – wie bei allen anderen *PGP-Methoden* – als Rückgabewert geliefert.

4.4.3. Steuermethoden

Die folgenden Steuermethoden können dazu gebraucht werden, Eigenschaften wie z.B. Kompression oder ASCII-Ausgabe abzufragen, ein- und auszuschalten.

```
public boolean getArmor();

public boolean getText();

public boolean getZip();

public void setArmor(boolean on);

public void setText(boolean on);

public void setZip(boolean on);
```

Mit den `get`-Methoden können die Zustände der Flags `armor`, `text` und `zip` abgefragt werden. Die `set`-Methoden können dazu benutzt werden, die folgenden Eigenschaften ein- und auszuschalten.

- **armor**
  Falls `armor` auf *true* gesetzt wird, wird die verschlüsselte und/oder signierte Nachricht in ausdruckbare (nicht-binäre) Zeichen verwandelt und eingerahmt durch die folgenden Zeilen:

  ```
  ---BEGIN PGP MESSAGE---
  Version: 2.3a
  ```

  Hier steht die verschlüsselte Nachricht im ASCII-Format.

  ```
  ---END PGP MESSAGE---
  ```

  Falls `armor` *false* ist, enthält die Output-Datei lediglich die verschlüsselten Informationen in binärem Format ohne Hinweis auf PGP. Default-Wert ist *False*.

- **text**
  Mit Hilfe dieser `boolean`-Variablen kann PGP mitgeteilt werden, dass es sich beim Klartext um ASCII-Text handelt und nicht um eine binäre Datei. Default-Wert ist *False*.

- **zip**
  Falls gewünscht, kann die Kompression des Klartextes unterbunden werden, standartmässig ist sie jedoch eingeschalten.

4.4.4. Statusmethoden

Will man nach erfolgter Entschlüsselung wissen, ob die Nachricht unterschrieben und verschlüsselt oder nur unterschrieben war, möchte man die Identität des Unterzeichnenden erhalten und sicher sein, dass dessen Signatur gültig ist, dann kann man die Statusmethoden verwenden, um diese Informationen zu erhalten.

Die Identität besteht - wie in PGP üblich - meist aus Namen und Mailadresse, also z.B. `Thomas Jampen <jampen@iam.unibe.ch>`.

```
public boolean wasEncrypted();

public boolean signatureValid();

public String getSigningParty();
```

Um zuverlässige Auskunft über die Art des Ciphertextes zu erhalten, dürfen diese Methoden natürlich erst nach erfolgter Entschlüsselung ausgeführt werden.

Falls `wasEncrypted` *false* ist, wurde die Nachricht nur signiert, sonst signiert und verschlüsselt. Die Methode `signatureValid` gibt nur *true* zurück, falls die Signatur gültig ist. Die Methode `getSigningParty` gibt, falls die Signatur gültig ist, den Namen des Unterzeichnenden zurück, ansonsten den String *"no signature"*.

4.4.5. Fehlerbehandlung

Jede *PGP-Methode* gibt einen Fehlercode zurück. Falls dieser gleich 0 ist, ist die Aktion fehlerfrei durchgeführt worden, falls der Code grösser als 0 ist, ist ein Fehler aufgetreten. Um zu erfahren, um welchen Fehler es sich handelt, kann nach gescheiterter Ver- resp. Entschlüsselung die folgende Methode aufgerufen werden.

```
public String getErrorMsg(int errCode);
```

In Tabelle 4.1 sind die möglichen Fehlercodes und ihre Bedeutung aufgelistet.

Fehlercode	Bedeutung
0	Es ist *kein* Fehler aufgetreten!
1	`pubring.pgp` kann nicht geöffnet werden
2	`secring.pgp` kann nicht geöffnet werden
3	Input-Datei kann nicht geöffnet werden
4	Output-Datei kann nicht geöffnet werden
5	Neuer Keyring kann nicht angelegt werden
6	Öffentlicher Keyring kann nicht aktualisiert werden
7	Privater Keyring kann nicht aktualisiert werden
10	Schlüssel kann nicht gefunden werden
11	Öffentlicher Schlüssel kann nicht gefunden werden
12	Privater Schlüssel kann nicht gefunden werden
13	Falscher IDEA Schlüssel
20	Datei ist keine PGP Nachricht
21	Unerwartetes Dateiende
22	Paket mit Ascii-Armor ist korrupt
23	Paket-Fehler
24	Fehlerhaftes RSA-Paket
25	Fehlerhaftes Signatur-Paket
26	Unbekannter Paketfehler
30	Ungültiges Passwort

Tabelle 4.1.: Fehlercodes des Java PGP API.

4.5. Das PGP API (C Seite)

Die bestehende Implementation von PGPTools wurde wie folgt abgeändert:

- PGPTools enthielt eine umfangreiche `main` Methode, welche in mehrere kleine Methoden unterteilt wurde. Zum Teil musste JNI-Funktionalität hinzugefügt werden, um Abfragen, wie z.B. die der `init` Methode (siehe 4.5.2) durchführen zu können. Abbildung 4.1 verdeutlicht diese Änderungen.
- Die Native-Methoden rufen die oben erwähnten Methoden auf und ersetzen das Parsen der Kommandozeilenargumente, da der Aufruf von PGP nicht mehr aus der Shell erfolgt, sondern via JNI.
- Diverse `exit 1;` Aufrufe wurden ersetzt durch `return x;`, wobei x für den entsprechenden Fehlercode steht (siehe Tabelle 4.1).
- Bei Aktionen, welche eine Interaktion mit dem Benutzer erfordern, wurden bei der bestehenden Implementation Abfragen an STDOUT geschrieben und die vom Benutzer gelieferten Daten von STDIN gelesen. Das war für dieses Projekt nicht sinnvoll, deshalb stellt das Java API diese Informationen als Argumente zur Verfügung. Eine Abfrage auf der C Seite entfällt also.

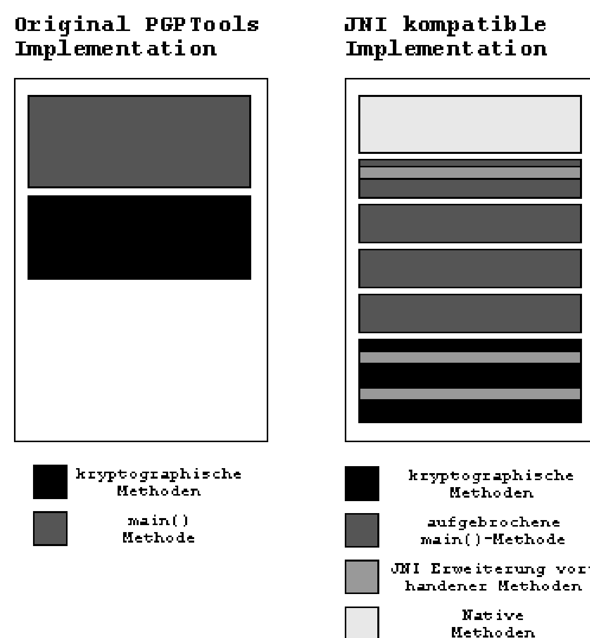

Abbildung 4.1.: Änderungen der C-Implementation

Auch hier lassen sich die Methoden in Gruppen unterteilen und besser getrennt betrachten. Es wird nicht jede Methode beschrieben, sondern nur auf einige wichtige Dinge hingewiesen, die im Zusammenhang mit dem JNI stehen.

- Native Methoden
- Hilfsmethoden

Zuerst sei jedoch ein allgemeiner Punkt zur Implementation erwähnt. Um überhaupt mit dem JNI arbeiten zu können, müssen sowohl `jni.h` als auch die von Java generierten Header-Dateien - in diesem Fall `PGPi.h` - eingebunden werden. Das heisst, die C-Datei `PGPi.c`, welche Native-Methoden implementiert, braucht die folgende Zeilen:

```
#incude <jni.h>
#inclde "PGPi.h"
```

4.5.1. Native Methoden

In diese Gruppe gehören die Implementationen der in Kapitel 4.4.2 besprochenen *PGP-Methoden*. Die Deklarationen dieser Methoden sind komplizierter als diejenigen normaler C-Methoden:

```
JNIEXPORT jint JNICALL
 Java_PGPi_showKeys(JNIEnv *env, jobject jobj, jboolean sigson);

JNIEXPORT jint JNICALL
 Java_PGPi_encrypt(JNIEnv *env, jobject jobj, jstring euid, jstring infile);

JNIEXPORT jint JNICALL
 Java_PGPi_sign(JNIEnv *env, jobject jobj, jstring suid, jstring infile, jstring passphrase);

JNIEXPORT jint JNICALL
 Java_PGPi_encryptAndSign(JNIEnv *env, jobject jobj, jstring suid, jstring euid,
                          jstring infile, jstring passphrase);

JNIEXPORT jobjectArray JNICALL
 Java_PGPi_decryptc(JNIEnv *env, jobject jobj, jstring infile, jstring passphrase);

JNIEXPORT jint JNICALL
 Java_PGPi_deleteKey(JNIEnv *env, jobject jobj, jstring uid);

JNIEXPORT jint JNICALL
 Java_PGPi_addPubKey(JNIEnv *env, jobject jobj, jstring infile);

JNIEXPORT jint JNICALL
 Java_PGPi_signKey(JNIEnv *env, jobject jobj, jstring uid, jstring signing,
                   jstring passphrase);

JNIEXPORT jint JNICALL
 Java_PGPi_generateKey(JNIEnv *env, jobject jobj, jstring ran, jint size, jstring uname,
                       jstring passphrase);

JNIEXPORT jint JNICALL
 Java_PGPi_keyMaint(JNIEnv *env, jobject jobj);

JNIEXPORT jint JNICALL
 Java_PGPi_extractKey(JNIEnv *env, jobject jobj, jstring uid, jstring infile);

JNIEXPORT jint JNICALL
 Java_PGPi_examine(JNIEnv *env, jobject jobj, jstring infile);
```

Auch hier fällt wieder die Methode `decrpytc` auf, sie gibt als einzige nicht einen `jint` zurück, sondern ein `jobjectArray`, also ein Array, das aus Java-Objekten - in diesem Fall aus Strings - besteht.

Die Signatur jeder Methode enthält die Makros `JNIEXPORT` und `JNICALL`, welche in `jni.h` definiert sind. Diese stellen sicher, dass die Methode aus der Native-Bibliothek exportiert wird, und dass der C-Compiler den Call-Konventionen entsprechenden Code generiert. Jede Methode hat mindestens zwei Argumente, ein `JNIEnv` und ein `jobject`. Der JNIEnv-Pointer zeigt an eine Stelle, welche einen Pointer zur Funktionentabelle, bestehend aus allen JNI-Funktionen, enthält. Auf Datenstrukturen der Java VM wird immer via JNI-Funktionen zugegriffen. Das zweite Argument ist eine Referenz auf das Object, auf welches die Funktion angewendet wird (ähnlich wie *this* in C++).

Exemplarisch werden die beiden Funktionen `sign` und `decrpytc` betrachet.

```
JNIEXPORT jint JNICALL
 Java_PGPi_sign(JNIEnv *env, jobject jobj, jstring suid, jstring infile, jstring passphrase) {

    signuid = (*env)-> GetStringUTFChars(env, suid, 0);
    infilename = (*env)-> GetStringUTFChars(env, infile, 0);
    pass = (*env)-> GetStringUTFChars(env, passphrase, 0);

    init(env, jobj);

    errcode = openKeyring();
    if (errcode)
        return errcode;

    errcode = prepareSign();
    if (errcode)
        return errcode;

    errcode = openInFile();
    if (errcode)
        return errcode;

    signOrEncrypt();
    justSign();

    errcode = endSignOrEncrypt();
    if (errcode)
        return errcode;

    finish();
    return 0;
}
```

Am Anfang werden die drei `jstring` Argumente umgewandelt mittels `GetStringUTFChars` Funktion in einen `jbyte*`, der einem `byte*` in C zugewiesen werden kann. Anschliessend werden die kryptographischen Algorithmen durchlaufen und eventuelle Fehlermeldungen sofort an Java zurückgegeben. Falls nirgends ein Fehler aufgetreten ist, wird am Schluss der `int` Wert 0 zurückgegeben, was dem Rückgabetypen `jint` zugewiesen werden kann.

```
1   JNIEXPORT jobjectArray JNICALL
2    Java_PGPi_decryptc(JNIEnv *env, jobject jobj, jstring infile, jstring passphrase) {
3       jclass strCls;
4       jobjectArray jarr;
5       char errc[2];
6       jbyteArray jerrArr;
7
8
9       errc[0] = (char)errcode;
10      errc[1] = 0;
11
12      ec = (*env)-> NewStringUTF(env, errc);          // Error ?
13      so = (*env)-> NewStringUTF(env, "0");           // SignatureOk ?
```

```
14          sp = (*env)-> NewStringUTF(env, "no signature"); // SigningParty ?
15          we = (*env)-> NewStringUTF(env, "0");            // WasEncrypted ?
16
17          strCls = (*env)-> FindClass(env, "Ljava/lang/String;");
18          jarr = (*env)-> NewObjectArray(env, 4, strCls, NULL);
19
20          (*env)-> SetObjectArrayElement(env, jarr, 0, ec);
21          (*env)-> SetObjectArrayElement(env, jarr, 1, so);
22          (*env)-> SetObjectArrayElement(env, jarr, 2, sp);
23          (*env)-> SetObjectArrayElement(env, jarr, 3, we);
24
25          infilename = (*env)->GetStringUTFChars(env, infile, 0);
26          pass = (*env)->GetStringUTFChars(env, passphrase, 0);
27
28          init(env, jobj);
29
30          errcode = openKeyring();
31          if (errcode) {
32              errc[0] = (char)errcode;
33              ec = (*env)-> NewStringUTF(env, errc);
34              (*env)-> SetObjectArrayElement(env, jarr, 0, ec);
35              return jarr;
36          }
37
38          errcode = openInFile();
39          if (errcode) {
40              errc[0] = (char)errcode;
41              ec = (*env)-> NewStringUTF(env, errc);
42              (*env)-> SetObjectArrayElement(env, jarr, 0, ec);
43              return jarr;
44          }
45
46          errcode = doDecrypt();
47          if (errcode) {
48              errc[0] = (char)errcode;
49              ec = (*env)-> NewStringUTF(env, errc);
50              (*env)-> SetObjectArrayElement(env, jarr, 0, ec);
51              return jarr;
52          }
53
54          finish();
55
56          errc[0] = (char)errcode;
57          (*env)-> SetObjectArrayElement(env, jarr, 0, ec);
58          (*env)-> SetObjectArrayElement(env, jarr, 1, so);
59          (*env)-> SetObjectArrayElement(env, jarr, 2, sp);
60          (*env)-> SetObjectArrayElement(env, jarr, 3, we);
61
62          return jarr;
63      }
```

In den Zeilen 12 - 15 werden `jstring`s erzeugt und mit den Standartwerten gefüllt. In Zeile 17 wird mit Hilfe von `FindClass` eine Referenz zur Java-Klasse `java.lang.String` zugewiesen und in Zeile 18 ein neues `jobjectArray` dieser String-Klasse erzeugt. Die Zeilen 20 - 23 füllen das Array mit den vorher zugewiesenen `jstring`s.

Falls nun ein Schlüsselring oder die Input-Datei nicht geöffnet werden kann, oder wenn während der Entschlüsselung ein Fehler auftritt, wird der entsprechende Fehlercode in einen `jstring` umgewandelt und dem ersten Array-Element zugewiesen. Schliesslich wird das Array als Rückgabewert an Java geliefert.

Wenn die Entschlüsselung ohne Fehler durchgeführt werden konnte, werden in den Zeilen 57 - 60 die in der Hilfsfunktion `doDecrypt` gewonnenen Informationen ins Array codiert und an Java zurückgegeben.

4.5.2. Hilfsmethoden

Als Hilfsmethoden werden die restlichen C-Methoden zusammengefasst, da sie nicht direkt von Java aus aufgerufen werden können und mit Ausnahme von `init` und `doDecrypt` keine JNI-Funktionalität aufweisen. Deshalb werden hier auch nur diese beiden Methoden genauer betrachten.

Die `init` Funktion wird von den *Native-Methoden* aufgerufen, um unter anderem den Zustand der Membervariablen der `PGPi` Klasse abzufragen. Der folgende Code-Abschnitt zeigt, wie dieser Zugriff konkret durchgeführt wird.

```
jcl = (*env)-> GetObjectClass(env, jobj);

aid = (*env)-> GetMethodID(env, jcl, "getArmor", "()Z");
tid = (*env)-> GetMethodID(env, jcl, "getText", "()Z");
cid = (*env)-> GetMethodID(env, jcl, "getZip", "()Z");

armor = (*env)-> CallBooleanMethod(env, jobj, aid);
text = (*env)-> CallBooleanMethod(env, jobj, tid);
compress = (*env)-> CallBooleanMethod(env, jobj, cid);
```

Mit Hilfe der Funktion `GetObjectClass` wird eine Referenz zur Klasse `PGPi` erzeugt. Diese Referenz, zusammen mit dem Interface-Pointer `env`, dem Java-Methodennamen (z.B. `getArmor`) und der Signatur dieser Methode, ergibt die Argumentenliste für die Funktion `GetMethodID`, welche eine eindeutige Identifikation der Java-Methode liefert.

Die Signatur ist wie folgt zu verstehen. In der runden Klammer stehen die Argumenttypen (in diesem Beispiel sind keine vorhanden). Hinter der Klammer steht der Rückgabetyp. Die möglichen Bezeichner der Standart-Datentypen können aus der Tabelle 4.2 entnommen werden. Die Signatur eines Objektes besteht aus einem `L`, dem vollständigen Klassenamen und einem Strichpunkt (z.B. `Ljava/lang/String;`).

Schliesslich ruft die Funktion `CallBooleanMethod` die entsprechende Java-Methode auf und gibt deren Returnwert zurück.

Bezeichner	Java Typ
Z	`boolean`
B	`byte`
C	`char`
S	`short`
I	`int`
J	`long`
F	`float`
D	`double`
L	Java Objekt

Tabelle 4.2.: Bezeichner für Java Typen.

Die `doDecrypt` Funktion entschlüsselt und verifiziert Ciphertexte und setzt die Variablen, die in Java mit Hilfe der *Statusmethoden* abgefragt werden können. Das sind die Variablen, die dem `jobjectArray` vor der Rückkehr zu Java zugewiesen werden.

```
so = (*env)-> NewStringUTF(env, "1");

sp = (*env)-> NewStringUTF(env, userid);

we = (*env)-> NewStringUTF(env, "1");
```

so (signature ok) wird auf 1 gesetzt, falls der Hashwert des unterzeichnenden Schlüssels mit dem des gespeicherten Schlüssels übereinstimmt. Der Variablen sp (signing party) wird der Name des Unterzeichnenden zugewiesen, sofern die Signatur gültig ist und die Nachricht überhaupt unterzeichnet wurde. we (was encrypted) wird auf 1 gesetzt, wenn die Nachricht verschlüsselt - nicht nur signiert - war.

5. Evaluation des Ansatzes

5.1. Leistungsvergleich

Um einen Vergleich zum skript-basierten Ansatz von Manuel Günter zu erhalten, sollte nun bei beiden Versionen die Zeit für Verschlüsselung, Signatur und Entschlüsselung gemessen werden. Ausserdem schliesst dieser Vergleich noch eine eigene Java-Implementation ein, um abschätzen zu können, wie viel langsamer eine reine Java-Implementation ist. Dieser zweite Vergleich ist allerdings nicht so aussagekräftig wie der erste, da die Implementationen nicht den genau gleichen Voraussetzungen genügen.

5.1.1. Skript vs. JNI

Als Basis für den Geschwindigkeitsvergleich dienen verschieden grosse Input-Dateien, welche mit Hilfe der Java-Klasse `Compare` von beiden Implementationen verarbeitet werden. Die RSA-Schlüssel haben eine Länge von 1024 bit. Der Vergleich umfasst die folgenden Operationen:

- Plaintext-File signieren
- signiertes File decodieren
- decodiertes File überprüfen
- Signer identifizieren
- verifizierte Daten auslesen

- Plaintext-File verschlüsseln **und** signieren
- Ciphertext-File decodieren
- decodiertes File überprüfen
- Signer identifizieren
- entschlüsselte und verifizierte Daten auslesen

Die Tabelle 5.1 listet die Resultate auf. Um Resultatverfälschungen zu vermeiden, welche aufgrund temporärer Mehrbelastung der CPU auftreten könnten, wurde die Testroutine meist mehrmals durchlaufen.

Es wird deutlich, dass der Skript-Ansatz vor allem für kleine Dateien ungeeignet und wesentlich langsamer ist, als die JNI-basierte Lösung. Für grössere Dateien, ab 100 kB, wird der Skript-Ansatz schneller, resp. der JNI-Ansatz langsamer.

Dateigrösse	Iterationen	Skript	JNI
500 B	10	2.3 s	1.2 s
500 B	100	22.6 s	12.4 s
1 kB	10	2.6 s	1.5 s
1 kB	100	25.0 s	14.6 s
10 kB	10	4.5 s	3.6 s
10 kB	100	44.1 s	35.5 s
100 kB	1	2.6 s	2.6 s
100 kB	10	25.7 s	25.9 s
500 kB	1	16.0 s	16.8 s
1 MB	1	31.8 s	33.4 s

Tabelle 5.1.: Geschwindigkeitsvergleich: Skript - JNI.

5.1.2. Die Geschwindigkeit der reinen Java Implementation

Für diese Messungen konnte auf eine bestehende Java-Implementation kryptographischer Algorithmen zurück gegriffen werden [6]. Nun war es natürlich interessant, einen Leistungsvergleich durchführen zu können. Das war nicht ganz einfach, denn diese Krypto-Bibliothek arbeitet nicht mit Dateien, sondern nur mit Strings. Sie ist vorwiegend dazu gedacht, in Applets die Funktionsweise einiger Algorithmen zu demonstrieren. Da keine aufwändigen Tests durchgeführt werden sollten, sondern nur die Grössenordnung von Interesse war, wurde diese Bibliothek nicht verändert, sondern nur zur gemessenen Zeit diejenige addiert, die von Java benötigt wird, um Dateien entsprechender Grösse zu lesen resp. zu schreiben. Was ebenfalls fehlte, war die Implementation eines Hash-Algorithmus wie z.B. MD5, der in PGP verwendet wird. Und natürlich ist die Krypto-Bibliothek nicht vollständig PGP-kompatibel. Es wurden lediglich RSA-Schlüssel derselben Länge erzeugt und damit 128 bit Strings verschlüsselt (das entspricht dem IDEA-Session-Key, der in PGP asymmetrisch verschlüsselt wird). Danach wurde dieser IDEA-Schlüssel verwendet, um einen String, der die entsprechende Datei enthielt, zu verschlüsseln. Anschliessend wurden diese beiden verschlüsselten (resp. signierten) Strings wieder entschlüsselt.

Es geht also wirklich nur darum zu zeigen, dass eine reine Java-Implementation wesentlich langsamer ist als eine C-Implementation (siehe Tabelle 5.2).

5.2. Sicherheit

5.2.1. Was hat man gewonnen?

Es ist schwierig, bezüglich Sicherheit vollkommen klare und richtige Angaben zu machen. Sicher ist jedoch, dass man gegenüber einer reinen Java-Implementation die Sicherheit gewonnen hat, welche durch die jahrelange Erfahrung der PGP-Programmierer aufgebaut wurde. Der PGP Source-Code ist öffentlich, also konnten ihn viele Leute überprüfen, und gefundene Fehler konnten korrigiert werden.

Dateigrösse	Iterationen	Java
500 B	10	5.8 s
500 B	100	58.7 s
1 kB	10	6.3 s
1 kB	100	63.4 s
10 kB	10	8.3 s
10 kB	100	81.7 s
100 kB	1	4.2 s
100 kB	10	40.0 s
500 kB	1	21.5 s
1 MB	1	40.2 s

Tabelle 5.2.: Geschwindigkeit der reinen Java-Implementation.

5.2.2. Welche neuen Lücken sind entstanden?

Wenn man dieses API mit einer herkömmlichen PGP-Version vergleicht, ist sicher zu beachten, dass in Java Passwörter weder direkt aus dem Speicher gelöscht noch überschrieben werden können. Versucht man zum Beispiel einen String buchstabenweise (mit `replace`) zu ersetzen, wird die Operation nicht auf dem originalen String ausgeführt, sondern eine Kopie mit der entsprechenden Änderung erstellt. Es bleibt nur die Möglichkeit, der Variablen sofort nach Gebrauch `null` zuzuweisen und `System.gc()` aufzurufen. Somit sollte eigentlich der Garbage Collector die nicht mehr referenzierten Objekte aus dem Speicher entfernen. Java garantiert jedoch nicht, dass der Garbage Collector nach dem Aufruf auch sofort ausgeführt wird.

Vergleicht man das Java API mit der skript-basierten Lösung, ist sicherlich die Passwortübergabe ein Thema. Ob die Parameterübergabe mittels JNI vor Angriffen von aussen genügend geschützt wird, hängt von der Implementation der Virtual Machine ab.

5.3. Offene Aspekte

5.3.1. Schlüsselverwaltung

Es ist möglich, mit meinem API neue Schlüssel zu erzeugen, zu löschen, zu unterschreiben, öffentliche Schlüssel zu importieren und zu exportieren, aber es fehlt noch die Möglichkeit, die Trust-Levels zu verändern und das Passwort des eigenen privaten Schüssels zu ändern.

Die Methode `keyMaint` ist zwar vorhanden, sie aufzurufen bringt jedoch nichts, da eine vollständige Implementation noch fehlt.

5.3.2. Portabilität

Bis jetzt wurde das API nur auf dem Dell Laptop unter Linux und Windows 98 erfolgreich kompiliert und getestet. Unter Windows 98 wurde der GCC Mingw32 v2.95.2 (msvcrt) [8] und das J2SDK von Sun [10] verwendet. Die in Windows 98 kompilierte Library lief ebenfalls unter Windows NT/2000. Wie oben erwähnt hat es bisher unter Unix noch nicht geklappt. Auf anderen Linux Distributionen dürfte dieses API jedoch problemlos funktionieren.

Anhang

A. Quellcode des PGP Java API

A.1. Die Java Klasse

Datei: **PGPi.java**

```
/**
 * @(#)PGPi.java 1.0 2001/04/03
 *
 * Copyright 2001 Thomas Jampen, University of Berne, Switzerland
 */


/**
 * <br>The <code>PGPi</code> is a PGP interface class.
 * <br>This means it offers methods to access the C implementation
 *     of Phil Zimmermann's PGP (Pretty Good Privacy).
 * <br>It uses the java native interface JNI to interact with PGP
 *     compiled as a shared object.
 *
 *
 * @author Thomas Jampen
 * @version 1.0
 */
public class PGPi {



    /** True if ascii armor is desired. */
    private boolean armor = false;

    /** Uses compression if enabled. */
    private boolean zip = true;

    /** Input is text. */
    private boolean text = false;

    /** Indicates whether a signature is valid or not. */
    private boolean sigOK = false;

    /** True if the current file was encrypted (not just signed). */
    private boolean wasEncrypted = false;

    /** Contains the signer of a message or "no signature". */
    private String signer = "no signature";



    /**
     * Initializes a newly created <code>PGPi</code> object.
     * <br>The flags are set to provide the fastest possible
     * encryption.
     */
    public PGPi() {
```

```
        armor = false;
        zip = true;
        text = false;
    }



    /**
     * Initializes a newly created <code>PGPi</code> object.
     * <br>The flags are set corresponding to the arguments passed.
     *
     * @param armor True if ascii armor is desired.
     * @param zip True to enable compression.
     * @param text True if input is ascii.
     */
    public PGPi(boolean armor, boolean zip, boolean text) {
        this.armor = armor;
        this.zip = zip;
        this.text = text;
    }



    /**
     * Displays the contents of the keyring files on standard out.
     * With signatures if desired.
     *
     * @param sigson indicates whether signatures of keys are shown, too.
     * @return the error code as an <code>int</code> value.
     */
    public native int showKeys(boolean sigson);



    /**
     * Calls PGP to encrypt the contents of the specified file with the
     * public key of the given user.
     *
     * @param uid the user id to use for encryption.
     * @param infile the file to encrypt
     * @return the error code as an <code>int</code> value.
     */
    public native int encrypt(String uid, String infile);



    /**
     * Calls PGP to sign the contents of the specified file with the private
     * key of the given user.
     *
     * @param uid the userid to use for the signature.
     * @param infile the file to sign.
     * @param passphase the passphrase for uid's private key.
     * @return the error code as an <code>int</code> value.
     */
    public native int sign(String uid, String infile, String passphrase);



    /**
     * Calls PGP to sign and then encrypt the specified file.
     *
     * @param suid the userid to use for the signature.
     * @param euid the userid to use for the encryption.
     * @param infile the file to encrypt and sign.
     * @param passphrase the passphrase for suid's private key.
```

```
     * @return the error code as an <code>int</code> value.
     */
    public native int encryptAndSign(String suid, String euid, String infile,
                                     String passphrase);



    /**
     * Calls PGP to decrypt and/or verify the specified file.
     * <br>This method is called by the <code>decrypt</code> method and
     * cannot be used directly by other classes.
     *
     * @param infile the file to decrypt.
     * @param passphrase the passphrase for decryption.
     * @return the error code as an <code>int</code> value.
     */
    private native String[] decryptc(String infile, String passphrase);



    /**
     * Calls PGP to decrypt and/or verify the specified file.
     *
     * @param infile the file to decrypt.
     * @param passphrase the passphrase for decryption.
     * @return the error code as an <code>int</code> value.
     */
    public int decrypt(String infile, String passphrase) {
        String arr[] = decryptc(infile, passphrase);
        int error;

        if (arr[0].length() == 0)
            error = 0;
        else
            error = (int)arr[0].charAt(0);

        if (arr[1].equals("1"))
            sigOK = true;

        signer = arr[2];

        if (arr[3].equals("1"))
            wasEncrypted = true;

        return error;
    }



    /**
     * Calls PGP to delete the specified key from the keyring file.
     *
     * @param uid the uid of the key to be deleted.
     * @return the error code as an <code>int</code> value.
     */
    public native int deleteKey(String uid);



    /**
     * Calls PGP to add a public key to the public keyring.
     *
     * @param infile the file with the public key to be added.
     * @return the error code as an <code>int</code> value.
     */
    public native int addPubKey(String infile);
```

```
/**
 * Calls PGP to sign an existing public key.
 *
 * @param toBeUid the userid of the key to be signed.
 * @param suid the userid of the signing key.
 * @param passphrase the passphrase of suid's private key.
 * @return the error code as an <code>int</code> value.
 */
public native int signKey(String toBeUid, String suid, String passphrase);



/**
 * Calls PGP to generate new key pair.
 *
 * @param ran some random characters as a <code>String</code>.
 * @param length the desired key length (384 <= length <= 1024).
 * @param uid the new key pairs's userid.
 * @param passphrase the passphrase for the new key pair.
 * @return the error code as an <code>int</code> value.
 */
public native int generateKey(String ran, int length, String uid,
                              String passphrase);



/**
 * Calls PGP to do some key maintenance.
 * <br>Impossible to change the trust value at the moment.
 * <br> -> Does nothing important right now.
 *
 * @return the error code as an <code>int</code> value.
 */
public native int keyMaint();



/**
 * Calls PGP to extract a public key from the public keyring.
 *
 * @param uid the userid of key to exract from the public keyring.
 * @param outfile the file to be created containing the extracted key.
 * @return the error code as an <code>int</code> value.
 */
public native int extractKey(String uid, String outfile);



/**
 * Calls PGP to examine a file.
 *
 * @param infile the file to examine.
 * @return the error code as an <code>int</code> value.
 */
public native int examine(String infile);



/**
 * Loads the shared object <code>libpgpi.so</code> which
 * contains the PGP code.
 */
static {
```

```
        System.loadLibrary("pgpi");
    }



    /**
     * Returns whether an ascii armor is used or not.
     *
     * @return True if ascii armor is used.
     */
    public boolean getArmor() {
        return armor;
    }



    /**
     * Allows you to set whether you want to use ascii armor or not.
     *
     * @param on switch ascii armor on or off.
     */
    public void setArmor(boolean on) {
        armor = on;
    }



    /**
     * Returns whether input is set to text or not.
     *
     * @return True if text input is used.
     */
    public boolean getText() {
        return text;
    }



    /**
     * Allows you to set text input mode.
     *
     * @param on switch text input mode on or off.
     */
    public void setText(boolean on) {
        text = on;
    }



    /**
     * Returns whether compression is used.
     *
     * @return True if compression is used.
     */
    public boolean getZip() {
        return zip;
    }



    /**
     * Choose between compressed and uncompressed encryption.
     *
     * @param on switch compression on or off.
     */
    public void setZip(boolean on) {
```

```
        zip = on;
    }



    /**
     * Has the currently decrypted/verified message been encrypted
     * or just signed?
     *
     * @return True if message was encrypted.
     */
    public boolean wasEncrypted() {
        return wasEncrypted;
    }



    /**
     * Returns True if the current message has been signed and the
     * signature is valid.
     *
     * @return whether signature was good.
     */
    public boolean signatureValid() {
        return sigOK;
    }



    /**
     * Returns the signer of the current message as a <code>String
     * </code> if the signature was good.
     * Else it returns "no signature" (also if signature was forged
     * or corrupted!
     *
     * @return the signer's userid.
     */
    public String getSigningParty() {
        return signer;
    }



    /**
     * Returns the description of the first error that occured while
     * running PGP.
     * <br>Every native method returns an error code, if this code is not
     * zero call this method in order to get the corresponding error
     * message as a <code>String</code>.
     *
     * @param errCode the error that occured.
     * @return the error message as a <code>String</code>.
     */
    public String getErrorMsg(int errCode) {
        String pre = "PGPi Error Code " + errCode + ": ";

        switch (errCode) {
        case 0:
            return pre + "No error occured, everything worked fine!";

        /** file errors */
        case 1:
            return pre + "Unable to open pubring.pgp";
        case 2:
            return pre + "Unable to open secring.pgp";
        case 3:
```

```
            return pre + "Unable to open input file";
        case 4:
            return pre + "Unable to open output file";
        case 5:
            return pre + "Unable to create new keyring";
        case 6:
            return pre + "Unable to update public keyring";
        case 7:
            return pre + "Unable to update private keyring";

        /** key errors */
        case 10:
            return pre + "Key not found";
        case 11:
            return pre + "Public key not found";
        case 12:
            return pre + "Private key not found";
        case 13:
            return pre + "Wrong IDEA key";

        /** package errors */
        case 20:
            return pre + "No PGP message";
        case 21:
            return pre + "File ended rudely";
        case 22:
            return pre + "Ascii-armored package is corrupted";
        case 23:
            return pre + "Packet error";
        case 24:
            return pre + "RSA encrypted package is bad";
        case 25:
            return pre + "Signature package is bad";
        case 26:
            return pre + "Unknown package type";

        /** password errors */
        case 30:
            return pre + "Bad passphrase";

        /** default */
        default:
            return pre + "Unknown error";
        }
    }
}
```

Literaturverzeichnis

[1] M. Günter and T. Braun. Internet service delivery control with mobile code. In H. R. van As, editor, *Telecommunication Network Intelligence*, pages 3–19. IFIP, Kluwer Academic Publishers, September 2000.

[2] S. Liang. *The Java Native Interface, Programmer's Guide and Specification.* Addison-Wesley, 2. edition, August 1999.

[3] B. Stiller, T. Braun, M. Günter, and B. Plattner. Charging and accounting technology for the internet. In *4th European Conference on Multimedia Applications, Services, and Techniques ECMAST'99*, LNCS 1629, pages 281–296. Springer-Verlag, May 1998.

[4] `http://www.iam.unibe.ch/~rvs/cati/`.

[5] `http://www.cryptography.ch/uni/pgpjava/`.

[6] `http://www.cryptography.ch/cce/`.

[7] `http://gcc.gnu.org`.

[8] `ftp://ftp.xraylith.wisc.edu/pub/khan/gnu-win32/mingw32/gcc-2.95.2/`.

[9] `http://www.media-crypt.com/pages/fidea.html`.

[10] `http://java.sun.com`.

[11] `http://java.sun.com/docs/books/tutorial/native1.1/index.html`.

[12] `http://www.ietf.org/rfc/rfc1321.txt`.

[13] `http://www.pgpi.org/doc/faq/pgpi/en/#Own`.

[14] `http://www.pgpi.com`.

[15] `http://www.infonex.com/~mark/pgp/pgptools.html`.

[16] `http://www.rsa.com`.