

# Java Smart Card (JSC)

## Digitale signaturer



Nikolaj Aggeboe & Sune Kloppenborg Jeppesen

aggeboe@it-c.dk & jaervosz@it-c.dk

IT-C København

21. december 2001

# Indhold

<b>1</b>	<b>Indledning</b>	<b>4</b>
<b>2</b>	<b>Smart cards</b>	<b>5</b>
2.1	Hvad er et smart card . . . . .	5
2.2	Historie . . . . .	5
2.3	Smart card fordele . . . . .	6
2.4	Applikationer . . . . .	6
2.5	Korte fakta . . . . .	8
<b>3</b>	<b>Java smart cards</b>	<b>9</b>
<b>4</b>	<b>Digitale Signaturer</b>	<b>11</b>
4.1	Case study . . . . .	11
4.2	Message Digests . . . . .	12
<b>5</b>	<b>Public key kryptering</b>	<b>14</b>
5.1	RSA . . . . .	14
5.2	Sikkerhed i RSA . . . . .	15
5.3	RSA i praksis . . . . .	17
<b>6</b>	<b>Hash funktioner</b>	<b>19</b>
6.1	Hash funktioners egenskaber . . . . .	20
6.2	Konstruktion af hash funktioner . . . . .	20
6.3	SHA-1 . . . . .	22
<b>7</b>	<b>Nøgledistribution</b>	<b>24</b>
<b>8</b>	<b>Sikkerhed ved brug af Smart Card</b>	<b>25</b>
8.1	Terminal/klient problemet . . . . .	25
8.2	Fysiske angreb . . . . .	26
8.3	Mulige forholdsregler . . . . .	26
<b>9</b>	<b>Implementationen</b>	<b>28</b>
9.1	Formål . . . . .	28
9.2	Problemer . . . . .	28
9.3	Oversigt over implementationen . . . . .	29
9.4	Yderligere forbedringer . . . . .	29
<b>10</b>	<b>Installationsvejledning</b>	<b>30</b>
<b>11</b>	<b>Konklusion</b>	<b>32</b>
<b>A</b>	<b>Testkørsel</b>	<b>35</b>
<b>B</b>	<b>Signature.java</b>	<b>40</b>
<b>C</b>	<b>SignatureClient.java</b>	<b>46</b>



# 1 Indledning

Målet for dette projekt er at udvikle en digital signatur applet der kan afprøves i en Java smart card simulator, samt at redegøre for fordele og ulemper ved en sådan.

I første del af rapporten vil vi redegøre for anvendelsen af smart cards herunder Java smart cards. Derefter vil vi gå mere i dybden med kryptografiske teknikker, samt hash funktioner. For at afslutte den del med en gennemgang af sikkerheden ved en implementation af digital signature vha. Java smart cards.

I anden del af vil den implementerede løsning blive gennemgået og vi vil forsøge at give bud på hvordan den implementerede løsning kunne forbedres yderligere.

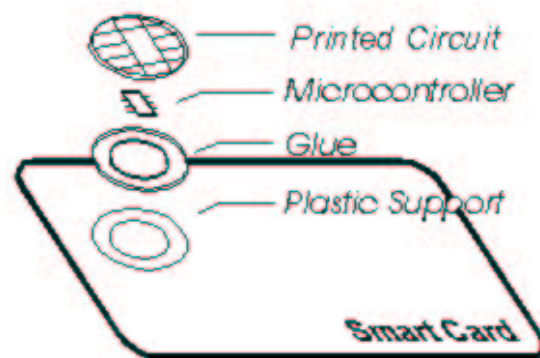
## 2 Smart cards

Eksplosionen af Internettet og trådløs digital kommunikation har udviklet måden at snakke og arbejde med andre mennesker på drastisk. Som verden er blevet mere forbundet, har forretningsmodellen ændret sig fra den traditionelle ansigt til ansigt i en butik til online transaktioner udført med et par museklik. Markedet for elektronisk handel og den udvikling der er foregået, har medført ikke bare nye muligheder for handel, men ligeledes en hel ny industri.

Det elektroniske markeds succes bygger på det samme niveau af sikkerhed, som offline firmaer har opbygget gennem mange år med ansigt til ansigt handel i butikker og bygger ligeledes på en teknologi til at lette en sådan handel. Sikkerheden og transportevnen af smart cards giver en sikker, troværdig, nyttig og effektiv måde at håndtere sikker e-handel og tilslutte en række nye applikationer.

### 2.1 Hvad er et smart card

Et smart card er på størrelse med et kredit kort. Det kan lagre og processere information gennem de elektroniske kredsløb, som er indbygget i en plastik krop.



Figur 1: Et smart cards opbygning.

Et smart card er en bærbar og brudsikker computer. Modsat kort med magnet stribe, har smart cards både processor kraft og kan lagre information. Derfor behøver de ikke adgang til en database på tidspunktet for transaktionen.

### 2.2 Historie

Ideen med at indbygge en chip i et plastik kort blev først introduceret af de to tyske opfindere, Jürgen Dethloff og Helmut Grötrupp i 1968. Uafhængigt søgte Kunitaka Arimura fra Japan, patent på smart card i 1970. Men der skete først rigtig noget med Roland Moreno's 47 smart card relaterede patenter mellem 1974 og 1979.

De første smart card forsøg foregik i Frankrig og Tyskland i 80'erne, hvor man brugte smart cards som telefonkort og sikre kredit/debit penge kort. Disse succesrige forsøg beviste potentialet af smart cards imod forfalskning og fleksibilitet.

Med nutidens avancerede chip teknologi og moderne kryptografi, er smart cards blevet

mere kraftfulde. De bruges nu til at lagre elektroniske penge (som erstatning for papir penge), til at lagre og sikre personlige oplysninger, til at modvirke ikke autoriseret adgang til kabel eller satellit udsendelser og til at forbedre trådløs telefon sikkerhed.

Sidst i 90'erne begyndte smart cards at få betydelig indflydelse i det amerikanske marked, med det voksende behov for sikker teknologi på e-handels området. Smart cards er allerede meget udbredt i især Europa og Asien pga. applikationer som GSM og bank kredit kort.

### 2.3 Smart card fordele

En umiddelbar fordel er smart cards indbyggede beregnings kraft. Sikkerhed, transportevne og "let at bruge" er andre nøgleord i forbindelse med smart cards fordele.

Processoren, hukommelse og I/O support på et smart card er indpakket i et enkelt integreret kredsløb indbygget i et plastik kort. Kortet er sikret mod angreb, da det ikke er afhængigt af potentielle trusler fra eksterne ressourcer. Når man skal hente lagret information fra kortet, kræver det den fysiske besiddelse af kortet, viden om kortets hardware og software og yderligere udstyr, som fx. en kort læser. Kortets sikkerheds funktioner, er ydermere styrket af kryptografiske funktioner. Data lagret på kortet, kan krypteres for at sikre dens hemmeligholdelse i den fysiske hukommelse, og udveksling af data mellem kortet og omverdenen kan krypteres og signeres. Adgangen til et smart card kræver som regel ligeledes at man indtaster en PIN kode, som modvirker at kortet bruges af en ikke autoriseret. Generelt er det langt sværere at bryde ind i et smart card end i en traditionel computer.



Figur 2: En kort læser.

En anden fordel ved smart cards, er deres oplagte transportevne. Et kort kan bæres i en pung på samme vis som andre kredit kort. På grund af disse karakteristika, er data lagret på kortet tilgængelige når som helst der er brug for dem og hvor som helst kortets ejer tager det.

Et smart card er let at bruge. For at begynde en transaktion, indsætter man kortet i en kort læser, helt i stil med et traditionelt kredit kort, og fjerner det efter transaktionen er udført.

### 2.4 Applikationer

Smart cards bruges ofte til at lagre data på en sikker måde og til at autentificere og sikre transaktioner. Dette afsnit handler om eksempler på applikationer, hvor smart cards benyttes.

I telekommunikations industrien tilbyder forudbetalte telefon kort, en penge fri, lav vedligeholdelses og kopieringsfri mekanisme til offentlige telefonbokse. Telekommunikations industrien er idag, det største marked for smart cards. GSM er et markant eksempel herpå. En GSM mobil telefon har et SIM kort, som er et smart card med en mindre plastik krop end de traditionelle kort. SIM kortet indentificerer ejeren og forsyner telefonen med et sæt krypteringsnøgler til digital transmissioner. Det er utrolig svært, at opfange mobil telefonnumre og ulovligt programmere dem ind i andre telefoner. Nøglen genereret af SIM kortet til kryptering er temporær og ændres for hver gang den bruges. Så selvom en GSM transmission kunne opfanges og dekrypteres, ville den være værdiløs til den næste transmission. Da brugerens identitet er programmeret i SIM kortet, kan brugeren ikke bare bruge en mobil telefon, men alle GSM kompatible telefoner der benytter SIM kort. Som trådløs kommunikation udbredes mere og mere, går udvikling langt udover stemme transaktioner. Telekommunikations operatører konkurrerer således om mange andre services, som mobil banking, internet adgang, osv. Alle disse teknologier bygger på smart cards og deres evne til at verificere en brugers identitet og sikre data transaktioner ved kryptering.

I bank og betalings verdenen, bruges smart cards ofte som sikre betalings kort. De fungerer meget som regulære kort med magnetstriben, men pga. deres indbyggede beregningskraft, kan de håndtere offline transaktioner og verifikationer. I modsætning til magnetstriben kort, kan smart cards ikke let kopieres og dermed misbruges. Smart cards er med til at afhjælpe kredit kort svindel, der hvert år koster banker rundt i verden milliarder. Et smart card kan også benyttes som en elektronisk pung, hvor man både kan foretage kredit og debit transaktioner, som erstatning for papir penge. Det er især nyttigt i forbindelse med online handel.

Smart cards anvendes ofte i masse transport systemer. Det være sig alt fra parkerings systemer, som betaling for parkering til skilift systemer, hvor mange tusind hver dag viser liftkort frem i et traditionelt system. Især de kontaktløse kort er meget praktiske i et sådant system, hvor man således ikke behøver finde sit kort frem hver gang man skal med en lift, men blot kan bære det i lommen og verificeres når man er inden for en given afstand af modtageren.

På internettet er bruger autentificering og adgangs kontrol et vigtigt motiv for at vælge smart cards. Der er et stigende forbrug af smart cards i alle former for systemer der benytter asymmetriske krypteringsmetoder. Her lagres den private nøgle og det digitale certifikat på kortet - to komponenter som verificerer brugerens identitet for den elektroniske verden. Applikationer som bruger smart cards som autentificering rækker over hjemmeside adgangs kontrol, digital signering af epost og sikre online transaktioner.

I lukkede edb systemer, som fx. universiteter, kan smart cards benyttes som autentificering ved og erstatning af traditionel logon, hvor brugernavn og adgangskode indtastes. Kortet kan ligeledes lagre informationer om status ved logout, så computeren har den samme tilstand næste gang samme bruger logger på. Smart cards benyttes også som adgangskort til døre (nøglekort), betaling i snack automater, osv.

## 2.5 Korte fakta

### Smart Card typer

- Kontakt kort og kontaktløse kort.
- Hukommelses kort og processor kort.

### Hardware

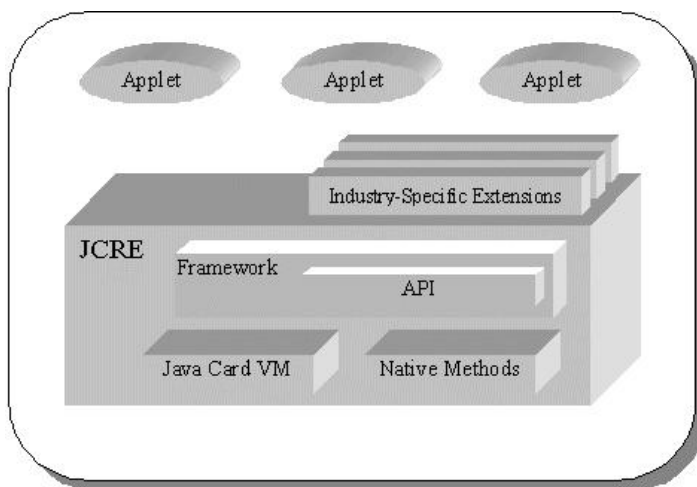
- CPU (8-bit 5 MHz mikroprocessor)
- Kryptografisk coprocessor (DES og RSA).
- ROM (permanent data), EEPROM (15-20K til program kode) og RAM (1-4K til temporær data).

### Kommunikation

- Et smart card kommunikerer med en vært via en kortlæser eller en integreret terminal.
- Udveksler APDUs via en request/response protokol.

### 3 Java smart cards

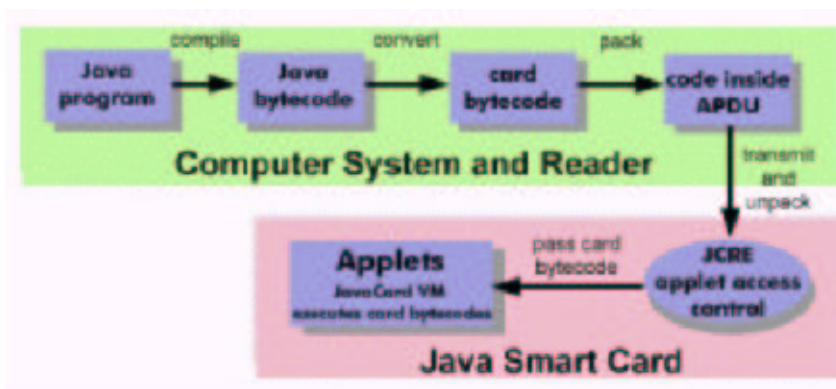
Et java smart card er et smart card der er istand til at afvikle java byte kode. *Java Card Runtime Environment* (JCRE) afvikler byte koden, styrer hukommelsen, afviklen af appletten (java applikationen på kortet). Denne består af JavaCard API (Application Programming Interface) og JCVM (JavaCard Virtual Machine).



Figur 3: Java card arkitektur.

JCRE tilbyder en standardiseret API for udvikling af JavaCard applikationer. API'en indeholder standard klasser og pakker. Disse gør det muligt at afvikle JavaCard applikationer på alle kort. En applikation er således ikke bundet til det kort den er udviklet til. Den virtuelle maskine er bygger ovenpå det integrerede kredsløb der findes på kortet i forvejen og det indbyggede operativ system. Det er kortproducentens opgave at udvikle en virtuel maskine der virker netop på deres egne kort.

Installationen og afviklingen af appletter foregår som vist på figur 3.



Figur 4: Java card arkitektur.

Kommunikationen mellem vært og kort foregår ved at transmitere data pakker. Disse kaldes *Application Protocol Data Unit* (APDU). JavaCard framework tilbyder en klasse APDU der

sørger for modtagelse og afsendelse af data pakker. Kommunikation med APDU foregår altid ved et APDU par i en command/response aktion. Java kortet afventer en command APDU og svarer igen med en response APDU. Disse pakker består af et byte array. Strukturen kan ses af figur 4 og nedenstående tabeller.

Header(5 bytes)				Body		
CLA	INS	P1	P2	LC	Data Field	LE

Tabel 1: Struktur af command APDU.

CLA er class byten, det er den der identificere hvilken applet på kortet som kommandoen er til. INS er instruktions byten, denne bestemmer hvilken instruktion appletten skal udføre. Betydningen af denne byte defineres af applet udbyderen. P1 og P2 kan indeholde ekstra parametre der skal anvendes for at udføre den angivne instruktion. LC angiver antallet af bytes i Data Field. LE angiver hvor mange byte værten forventer at få retur.

Body	Status word	
Data Field	SW1	SW2

Tabel 2: Struktur af response APDU.

SW er status word der viser resultatet af operationen. Igennem disse kan fejl kommunikerer tilbage til værten. Data Field indeholder det data der returneres efter den udførte operation. På grund af den meget begrænsede lager kapacitet og computer kraft på et java smart card understøttes den fulde Java API ikke. Istedet understøttes kun en delmængde, samt forskellige ekstra funktioner. Desuden mangler garbage collection og understøttelse for tråde.



data og det kan verificeres. Ligeledes at de transmitterede data ikke er ændret siden de blev signeret. En mere effektiv måde at opnå disse mål på, er at bruge en såkaldt message digest.

## 4.2 Message Digests

En message digest er på mange måder som en checksum. Message digest algoritmer tager en besked,  $m$ , og beregner et "fingeraftryk" med en fast længde af denne besked, kendt som en message digest,  $H(m)$ . Message digest'en beskytter data på en sådan måde, at hvis  $m$  ændres til  $m'$ , så vil den originale beregnede digest,  $H(m)$ , som sendes med disse data, ikke matche den beregnede digest,  $H(m')$ . En message digest giver os altså data integritet, men giver os ikke en digital signatur i sig selv. Målet er, at istedet for at Bob skal digitalt signere (dekryptere) hele beskeden ved at beregne  $d_B(m)$ , skal han istedet kunne nøjes med bare at signere message digest'en ved at beregne  $d_B(H(m))$ . Dvs. at har man  $m$  og  $d_B(H(m))$  (bemærk at  $m$  ikke er krypteret), er det nøjagtig ligeså godt som at have den signerede komplette besked,  $d_B(m)$ . Det betyder at,  $m$  og  $d_B(H(m))$  sammen skal kunne verificeres, ikke kunne reproduceres og forfalskes. At en digest ikke skal kunne forfalskes, betyder at message digest algoritmen skal have nogle specielle egenskaber, som vi skal se lidt nærmere på i det følgende.

Den ovenstående definition på en message digest minder meget om den for en checksum, som fx. Internet checksum. Message digest og Internet checksum er begge eksempler på hash funktioner. En hash funktion tager et input,  $m$ , og beregner en streng med fast længde, bedre kendt som en hash. Internet checksum, message digest, CRCs, osv. opfylder alle denne definition. Så hvis signering af en message digest skal være "lige så godt", som signering af hele beskeden, så skal en message digest algoritme opfylde følgende betingelser:

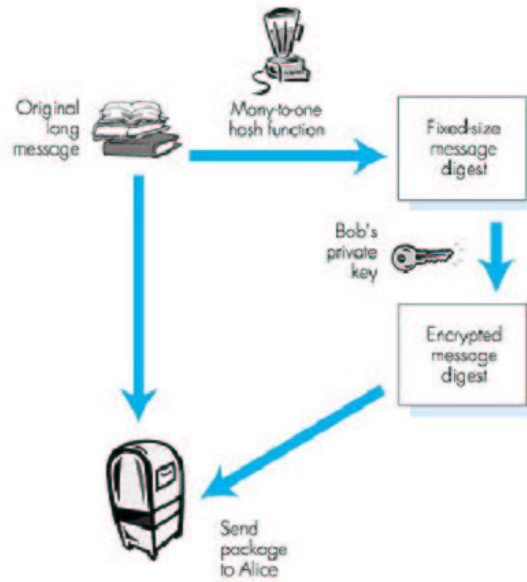
- Givet en message digest værdi,  $x$ , er det beregningsmæssigt ugørligt at finde en besked,  $y$ , så  $H(y) = x$ .
- Det er beregningsmæssigt ugørligt at finde to beskeder,  $x$  og  $y$ , så  $H(x) = H(y)$ .

Disse to egenskaber betyder, at det er beregningsmæssigt ugørligt for en person, at erstatte en besked med en anden, der er beskyttet af en message digest. Dvs. hvis  $(m, H(m))$  er beskeden og message digest parret skabt af afsenderen, så kan en intruder ikke forfalske beskeden med en anden besked,  $y$ , som har den samme message digest værdi, som den oprindelige digest. Så når Bob signerer  $m$  ved at beregne  $d_B(H(m))$ , ved vi at ingen anden besked kan erstatte  $m$ . Yderligere ved vi, at Bobs digitale signatur af  $H(m)$  unikt identificerer Bob, som den verificerbare og ikke reproducerbare underskriver af  $H(m)$ , og dermed også  $m$ .

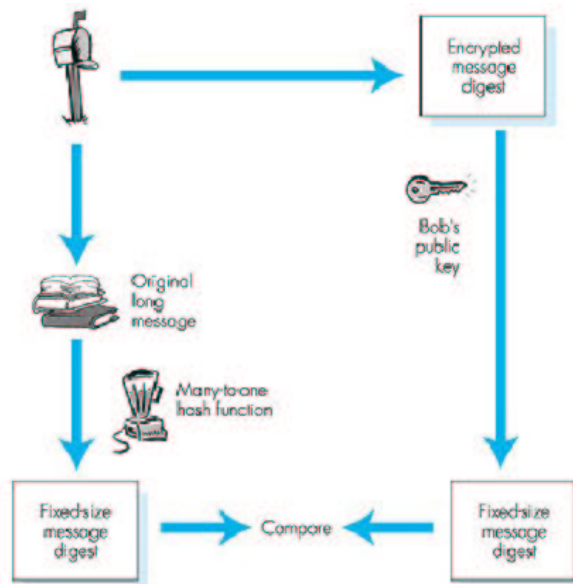
Figur 6 giver et overblik af operationerne i at skabe en digital signatur, når Bob vil sende en besked til Alice. Bob processerer sin originale besked gennem en mange-til-en hash funktion for at skabe en message digest. Herefter dekrypter Bob message digest'en med sin egen dekrypternøgle. Så sender Bob sin originale besked (i klartekst) sammen med den digitalt signerede message digest til Alice.

Figur 7 giver et overblik af operationerne i at verificere en signeret besked. Alice krypterer den signerede message digest med Bobs offentlige nøgle og får derved den beregnede message digest. Alice processerer ligeledes hash funktionen på den originale besked og får

derved endnu en message digest. Hvis de to message digests er ens, kan Alice være sikker på beskedens integritet og forfatter.



Figur 6: Signering af en digital signatur.



Figur 7: Verifikation af en digital signatur.

I det følgende skal vi se nærmere på nogle af de kryptografiske teknikker, som ofte benyttes i forbindelse med digitale signaturer.

## 5 Public key kryptering

I et public key krypterings system har hver person en offentlig nøgle  $e$  og en privat nøgle  $d$ . Som allerede nævnt, når B vil sende en krypteret besked  $m$  til A, får B fat i A's offentlige nøgle  $e_A$  og bruger krypteringssystemet til at beregne cifertexten  $c = e_A(m)$ . For at dekryptere, bruger A krypteringssystemet og A's private nøgle  $d_A$  til at beregne  $m = d_A(c)$ . Den offentlige nøgle skal ikke holdes hemmelig, men kan derimod gives til alle. Kun nøglens pålidelighed er afgørende for at garantere, at A er den eneste med den tilsvarende private nøgle. En fordel ved et sådant system er, at det ofte er meget nemmere at tilbyde pålidelige nøgler end at distribuere hemmelige nøgler, som det er nødvendigt i et symmetrisk krypteringssystem.

Public key kryptering er typisk langsommere end symmetriske krypteringssystemer, som fx. DES. Derfor anvendes public key systemer ofte til transport af hemmelige nøgler til symmetriske krypteringssystemer og til kryptering af små data mængder, som fx. kredit kort numre eller PIN koder. Ligeledes ofte i forbindelse med autentificering.

### 5.1 RSA

RSA er et sådant public key krypterings system. RSA er opkaldt efter opfinderne R. Rivest, A. Shamir og L. Adleman, og er et af de mest brugte public key systemer i verden. Det kan benyttes til at opnå både hemmeligholdelse og digitale signaturer. RSA's sikkerhed bygger på primtal faktoriserings problemet.

#### Nøgle generering i RSA

For at generere nøgler i RSA benyttes følgende algoritme:

1. Generer to store tilfældige primtal,  $p$  og  $q$ .
2. Beregn  $n = pq$  og  $\phi = (p - 1)(q - 1)$ .
3. Vælg et tilfældigt heltal  $e$ ,  $1 < e < \phi$ , så  $\gcd(e, \phi) = 1$ .  
 $\gcd$  er største fælles divisor (greatest common divisor).
4. Benyt Euklids algoritme til at beregne det unikke heltal  $d$ ,  $1 < d < \phi$ , så  $ed = 1 \pmod{\phi}$ .
5. Den private nøgle er  $d$  og den offentlige nøgle er  $(n, e)$ .

#### Definition

Heltallene  $n$  og  $d$  kaldes i RSA for encryption eksponent hhv. decryption eksponent, mens  $n$  kaldes for modulus.

#### Kryptering i RSA

B krypterer en besked til A med følgende algoritme:

1. Benyt A's offentlige nøgle  $(n, e)$ .
2. Præsenter beskeden som et heltal,  $m$  i intervallet  $[0, n - 1]$ .
3. Beregn

$$c = m^e \pmod{n} \quad (1)$$

4. Send ciffertexten  $c$  til A.

## Dekryptering i RSA

A dekrypterer ciffertexten  $c$  og får beskeden  $m$  med følgende algoritme:

1. Benyt den private nøgle  $d$ , som kun A har.

2. Beregn

$$m = c^d \bmod n \quad (2)$$

### Bevis

Lad os starte med at se på de skridt der sker i krypteringen af en besked med RSA. Beskeden repræsenteret som et heltal  $m$  opløftes eksponentielt til  $e$  modulo  $n$ . Dekryptering foregår på samme måde med eksponentiel opløft i  $d$  modulo  $n$ . Vi har altså:

$$(m^e)^d \bmod n = m^{ed} \bmod n \quad (3)$$

Nu har vi, at hvis  $p$  modulus er primtal og  $n = pq$ , så gælder

$$(m^e)^d \bmod n = m^{(ed \bmod (p-1)(q-1))} \bmod n \quad (4)$$

Og da vi valgte  $e$  og  $d$ , så  $ed - 1$  var eksakt delelig (uden rest) af  $(p - 1)(q - 1)$ , dvs.  $ed$  er delbar med  $(p - 1)(q - 1)$  med resten 1, fås nu

$$(m^e)^d \bmod n = m^1 \bmod n \quad (5)$$

Og dermed

$$(m^e)^d \bmod n = m \quad (6)$$

Dvs. ved først at eksponentielt opløfte til  $e$  (kryptere) og derefter eksponentielt opløfte til  $d$  får vi den originale besked  $m$ . Yderligere gælder det, at hvis man vender rækkefølgen på kryptering og dekryptering, får man stadig den originale besked  $m$ , dvs.

$$(m^e)^d \bmod n = m = (m^d)^e \bmod n \quad (7)$$

Som allerede nævnt bygger sikkerheden i RSA på, at der ikke er nogen kendte algoritmer til hurtigt at faktorisere et tal, i dette tilfælde værdien  $n$  til primtallene  $p$  og  $q$ . På den anden side, er det heller ikke bevist, at der ikke findes en sådan algoritme til hurtigt at faktorisere tal, og derfor er sikkerheden i RSA ikke garanteret.

## 5.2 Sikkerhed i RSA

Gennem tiden er der forsøgt en masse angreb mod RSA og i det følgende skal vi se nærmere på nogle af disse såvel som passende metoder til at sikre sig mod sådanne angreb.

Fakta

- Problemet i at beregne RSA dekrypterings eksponenten  $d$  fra den offentlige nøgle  $(e, n)$  og problemet i at faktorisere  $n$ , er beregningsmæssigt ækvivalente.
- Når man genererer RSA nøgler er det tvingende nødvendigt, at vælge  $p$  og  $q$  på en sådan måde, at det er beregningsmæssigt ugørligt at faktorisere  $n = pq$ .

### Krypterings eksponent $e$

For at forbedre krypterings effektiviteten, er det fornuftigt at vælge en lille krypterings eksponent  $e$ , som fx.  $e = 3$  eller  $e = 5$ . En gruppe kan derfor have den samme eksponent  $e$ , men det er vigtigt at de alle har en forskellig modulus  $n$ . Hvis A fx. vil sende den samme besked  $m$  til 3 personer, som alle har eksponenten  $e = 3$  og moduli  $n_1$  hhv.  $n_2$  og  $n_3$ , så vil A sende  $c_i = m^3 \pmod{n_i}$  for  $i = 1, 2, 3$ . Da disse moduli højst sandsynligt er parvist relative primtal, dvs. ikke har andre fælles faktorer, kan en person der opsnapper beskederne  $c_1, c_2$  og  $c_3$  benytte Gauss' algoritme til at finde en løsning  $x$ ,  $0 < x < n_1 n_2 n_3$  til de 3 kongruenser

$$\begin{aligned}x &= c_1 \pmod{n_1} \\x &= c_2 \pmod{n_2} \\x &= c_3 \pmod{n_3}\end{aligned}$$

Da  $m^3 < n_1 n_2 n_3$ , må det være tilfældet at  $x = m^3$ . Nu er det let for personen at tage den 3. rod af  $x$  og læse beskeden  $m$ .

Derfor skal en lille krypterings eksponent  $e$  aldrig bruges, når man vil sende den samme besked  $m$  til flere personer eller for den sags skyld, når man vil sende den samme besked med få, men kendte, variationer.

For at undgå et sådant angreb på RSA, kan man tilføje en pseudo tilfældig bit streng med passende længde til sin klar tekst besked, før kryptering. Strengen skal uafhængigt genereres for hver kryptering. Denne metode kaldes også *salting* af beskeden.

Små eksponenter  $e$  er også et problem med en lille besked  $m$ , fordi hvis  $m < n^{1/e}$ , så kan  $m$  fås fra ciffertexten  $c = m^e \pmod{n}$  ved simpelt at beregne den  $e$ 'de rod af  $c$ . Salting af beskeden kan også modvirke dette angreb.

### Forward search angreb

Hvis beskeden er lille eller forudsigelig, kan en person dekryptere ciffertexten  $c$  ved at kryptere alle mulige klartext beskeder  $m$  indtil  $c$  findes. Salting af beskeden kan ligeledes modvirke sådanne angreb.

### Dekrypterings eksponent $d$

Som det var tilfældet med krypterings eksponenten  $e$ , kunne det umiddelbart synes fornuftigt at vælge en lille eksponent  $d$  ligeså, for at forbedre effektiviteten af krypteringen. Men hvis  $\gcd(p-1, q-1)$  er lille (som det typisk er tilfældet) og  $d$  har op til kvart så mange bits som modulus  $n$ , så findes der effektive algoritmer til at beregne  $d$  ud fra den offentlige nøgle  $(n, e)$ . Disse algoritmer kan ikke udvides til tilfælde, hvor  $d$  er ca. samme størrelse som  $n$ . For at undgå sådanne angreb, skal  $d$  altså vælges ca. ligeså stor som  $n$ .

### Fælles modulus

Det følgende viser hvorfor det er tvingende nødvendigt for alle, at vælge deres egen modulus  $n$ .

Det foreslåes ofte, at en central tillidsfuld autoritet, som fx. et større firmas chef, skal vælge en enkelt RSA modulus  $n$  og så distribuere forskellige krypterings/dekrypterings par  $(e_i, d_i)$  til alle personer i firmaets netværk. Det er dog en meget dårlig ide, da det kan vises, at hvis en person har kendskab til et vilkårligt par  $(e_i, d_i)$ , kan personen også faktorisere modulus  $n$  og dermed afgøre dekrypterings eksponenterne  $d_i$  for alle andre personer i netværket. Mere vigtigt er det måske, at hvis en besked krypteres og sendes til to personer i netværket og begge beskeder opsnapes af en udenforstående person, findes der en teknik, som kan genskabe beskeden  $m$  ved kun at bruge offentlig tilgængelig information.

## Cycling angreb

Lad  $c = m^e \pmod n$  være en ciffertext. Lad  $k$  være et positivt heltal, så  $c^{e^k} = c \pmod n$ . Da kryptering er en permutation af besked intervallet  $\{0, 1, 2, \dots, n-1\}$  må der eksistere sådan et heltal  $k$ . Af samme grund må det være tilfældet, at  $c^{e^{k-1}} = m \pmod n$ . Dette fører til et såkaldt *cycling attack* på RSA. En angriber beregner  $c^e \pmod n$ ,  $c^{e^2} \pmod n$ ,  $c^{e^3} \pmod n, \dots$  indtil  $c$  findes første gang. Hvis  $c^{e^k} \pmod n = c$ , så må det tidligere nummer i cyclen, nemlig  $c^{e^{k-1}} \pmod n$ , være lig klartexten  $m$ .

Hvis det lykkes en angriber at finde  $k$ , så kan  $m = c^{e^{k-1}} \pmod n$  beregnes effektivt. Et cycling attack kan derfor anses for en algoritme til at faktorisere  $m$ . Men da faktorisering af store primtal anses for et NP-problem, udgår disse cycling angreb ikke en reel risiko for sikkerheden i RSA.

## Besked hemmeligholdelse

En klartext besked  $m$ ,  $0 < m < n-1$  i RSA, siges at være "ikke-hemmeligholdt" (unconcealed), hvis den dekrypterer til sig selv, dvs.  $m^e = m \pmod n$ . Der er altid nogle beskeder der ikke er hemmeligholdt, fx.  $m = 0$ ,  $m = 1$  og  $m = n - 1$ . Hvis  $p$  og  $q$  er tilfældige primtal og  $e$  vælges tilfældigt eller  $e$  er lille (fx.  $e = 3$ ), så er antallet af beskeder der ikke er hemmeligholdt af RSA, ubetydeligt små. Ikke hemmeligholdt beskeder er derfor ikke en reel risiko for sikkerheden i RSA.

## 5.3 RSA i praksis

Der findes utallige måder at forbedre effektiviteten af RSA i både hardware og software. Selv med disse forbedringer, er RSA i praksis betydeligt langsommere end de mest udbredte symmetriske krypteringssystemer, som fx. DES. RSA bruges derfor oftest, som allerede nævnt, til transport af symmetriske krypterings nøgler og til kryptering af små data mængder. I dette afsnit skal vi se nærmere på hvordan eksponenter, modulus og andre RSA komponenter bruges i praksis.

### Størrelse på modulus

På baggrund af de seneste algoritmer til faktorisering af heltal, giver en 512-bit modulus kun marginal sikkerhed i RSA. Fra 1996, for at modstå nogle kraftfulde faktorerings algoritmer, anbefales det derfor at anvende 768-bit moduli og 1024-bit eller højere moduli for høj sikkerhed i RSA.

### Udvælgelse af primtal

$p$  og  $q$  skal vælges, så det er beregningsmæssigt ugørligt at faktorisere  $n = pq$ . En restriktion, for at undgå den eliptiske kurve faktorerings algoritme, er at  $p$  og  $q$  skal vælges, så de har den ca. samme bit længde. Hvis fx. modulus  $n$  er 1024-bit lang, skal  $p$  og  $q$  være ca. 512-bit lange. En anden restriktion, er at forskellen på  $p$  og  $q$  ikke må være alt for lille. Det skyldes, at hvis  $p - q$  er lille, så er  $p \approx q$  og dermed  $p \approx \sqrt{n}$ , da  $n = pq \approx p^2$ . Nu kan  $n$  effektivt faktoreres ved division med alle ulige heltal tæt på  $\sqrt{n}$ . Hvis  $p$  og  $q$  vælges tilfældigt, så vil sandsynligheden for at  $p - q$  er lille være ubetydelig.

Det anbefales yderligere af mange forfattere, at  $p$  og  $q$  vælges som stærke primtal. Et primtal  $p$  siges at være stærk, hvis de 3 følgende betingelser opfyldes

- $p - 1$  har en stor primtals faktor, kaldet  $r$ .
- $p + 1$  har en stor primtals faktor.

- $r - 1$  har en stor primtals faktor.

Grunden til dette, er at stærke primtal skulle være bedre beskyttet mod forskellige faktorerings algoritmer. Det har vist sig, at stærke primtal yder minimal beskyttelse udover den allerede ydet af tilfældige primtal. Med den viden der er om faktorerings algoritmer, kan der derfor ikke gives nogen afgørende grund til at kræve udvælgelsen af stærke primtal i genereringen. På den anden side, yder stærke primtal ikke mindre sikkerhed end tilfældige primtal, og kræver kun minimal ekstra køretid at beregne dem.

### **Krypterings eksponenter**

Som det allerede er nævnt, kan man med fordel vælge krypterings eksponenten  $e$  lille, da det vil øge hastigheden af kryptering i RSA.

Krypterings eksponenten  $e = 3$  benyttes ofte i praksis, og det er i sådanne tilfælde nødvendigt at hverken  $p$  eller  $q$  er delelig med 3. Det resulterer i en meget hurtig kryptering, da kryptering kun kræver 1 modulær multiplikation og 1 modulær kvadrering. En anden ofte anvendt krypterings eksponent er  $e = 2^{16} + 1 = 65537$ . Dette tal har kun to 1 taller i dets binære repræsentation og kryptering kræver derfor kun 16 modulære multiplikationer og 1 modulær kvadrering. Krypterings eksponenten  $e = 2^{16} + 1 = 65537$  har den fordel over  $e = 3$ , at den kan modstå nogle af de nævnte angreb, da det er højst usandsynligt, at den samme besked sendes til  $2^{16} + 1$  modtagere.

## 6 Hash funktioner

Kryptografiske hash funktioner spiller en vigtig rolle i moderne kryptografi. Der findes utallige forskellige typer hash funktioner, som også benyttes uden for den kryptografiske verden. Vi skal dog koncentrere os om de kryptografiske hash funktioner, der knytter sig til data integritet og autentificering.

Hash funktioner tager et input og producerer et output, som ofte kaldes hash kode, hash værdi eller bare hash. Mere præcist, tager en hash funktion  $h$  strenge af arbitrær længde og producerer strenge af fast længde (fx. 128-bits) heraf. Ideen med kryptografiske hash funktioner er, at en hash værdi repræsenterer en kompakt version af de originale data (ofte kaldet digital fingeraftryk eller message digest) og denne hash værdi kan bruges som en unik repræsentation af de originale data.

Hash funktioner benyttes ofte i forbindelse med data integritet og digitale signaturer, hvor der først beregnes en hash værdi af beskeden og denne hash værdi (som repræsentant for beskeden) signeres så istedet for selve beskeden. En bestemt type hash funktioner, kaldet *message authentication codes* (MACs), er specielt udviklet til at sikre autentificering med symmetriske krypterings algoritmer. En anden type hash funktioner er *message detection codes* (MDCs). Hash funktioner er opdelt i kategorier og det leder os til definitionen af en hash funktion.

### Hash funktion

En hash funktion  $h$ , er en funktion der har (minimum) følgende to egenskaber:

1. Komprimering -  $h$  komprimerer et input  $x$ , af arbitrær (endelig) bitlængde, til et output  $h(x)$  af konstant bitlængde  $n$ .
2. Let at beregne - givet  $h$  og et input  $x$ , er det let at beregne  $h(x)$ .

Som allerede nævnt, findes der to store grupper af hash funktioner MACs og MDCs, som igen kan opdeles i undergrupper:

#### 1. Modification detection codes - MDC

Generelt er formålet med denne gruppe, at skabe et repræsentativt billede eller hash værdi af en besked. Målet er, i forening med andre metoder og teknikker, at opnå sikkerhed for data integritet. MDC er en undergruppe af *unkeyed* hash funktioner og kan opdeles i to primære klasser:

- (a) one way hash funktioner (OWHFs): Sådanne hash funktioner gør det svært at finde et input udfra en forud defineret hash værdi.
- (b) collision resistance hash funktioner (CRHF): Sådanne hash funktioner gør det svært at finde to inputs, der giver den samme hash værdi.

#### 2. Message authentication codes - MAC

MACs primære formål er, uden brug af andre mekanismer, at give sikkerhed for både afsender af beskeden og data integritet. MACs har to typer input, en besked og en hemmelig nøgle. MAC er en undergruppe af *keyed* hash funktioner.

Denne simplificerede klassifikation af hash funktioner, skal afspejle den fundamentale forskel på MDCs og MACs, hvor en hemmelig nøgle indgår som en del af inputet. Andre applikationer hvor keyed hash funktioner indgår tæller bl.a. challenge-response identifikations

protokoller. Ligeledes findes der andre typer af unkeyed hash funktioner, som dog ikke skal nævnes yderligere.

For en god ordens skyld, skal det lige nævnes at givet en besked som input (og en hash algoritme), kan enhver beregne et output, en hash værdi.

## 6.1 Hash funktioners egenskaber

For dykke yderligere ned i hash funktioner, er det nødvendigt at definere en række egenskaber og definitioner. I den forbindelse er der 3 vigtige egenskaber, udover de to nævnt i ovenstående afsnit. En hash funktion  $h$  med input  $x, x'$  og output  $y, y'$  kan have følgende egenskaber:

1. *preimage resistance*  
For stort set alle output, er det beregningsmæssigt ugørligt at finde et input der giver denne hash værdi som output. Dvs. givet et output  $y$ , beregnet ud fra  $h(x) = y$ , er det "umuligt" at finde et input  $x'$ , så  $h(x') = y$ .
2. *2nd preimage resistance*  
Det er beregningsmæssigt ugørligt, at finde et andet input som har det samme output, som ethvert specificeret input. Dvs. givet  $x, x \neq x'$ , så er det "umuligt" at finde  $h(x) = h(x')$ .
3. *collision resistance*  
Det er beregningsmæssigt ugørligt at finde to forskellige inputs  $x, x'$ , som giver det samme output, dvs.  $h(x) = h(x')$ . Bemærk at begge inputs kan vælges frit.

Udtrykket beregningsmæssigt ugørligt (eller svært) defineres ikke yderligere, og skal ses i relation til referencen. Man kan opfatte udtrykket som ikke beregneligt inden for overskuelig fremtid, som fx. tusindvis af år. Det samme gør sig gældende for let at beregne. Udtrykket kan opfattes som beregneligt inde for sekunder eller minutter. Figur 8 viser klassifikationen af hash funktioner ud fra de give definitioner.

Da disse egenskaber er på plads, kan vi nu definere to tidligere omtalte hash funktion-sklasser.

### One way hash funktion

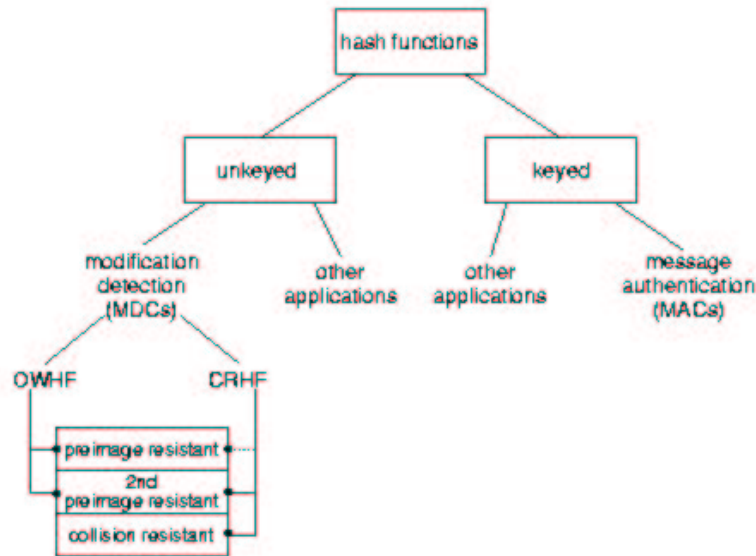
En one way hash funktion (OWHF) er en hash funktion  $h$ , med de følgende egenskaber (udover de nævnte egenskaber i definitionen af en hash funktion): preimage resistance og 2nd preimage resistance.

### Collision resistant hash funktion

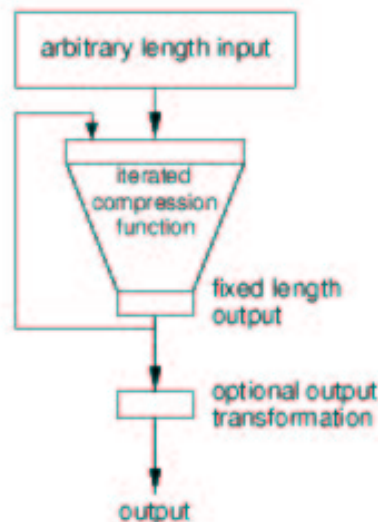
En kollisions resistant hash funktion (CRHF) er en hash funktion  $h$ , med de følgende egenskaber (udover de nævnte egenskaber i definitionen af en hash funktion): 2nd preimage resistance og collision resistance. En CRHF har næsten også altid preimage resistance egenskaben.

## 6.2 Konstruktion af hash funktioner

De fleste unkeyed hash funktioner  $h$ , er designet som iterative processor, som tager en arbitrær input bit længde, dvs. deler input op i blokke af fast længde, og beregner et konstant bit længde output. Figur 9 viser denne process.



Figur 8: Klassifikation af hash funktioner.



Figur 9: Konstruktion af en hash funktion.

Et input  $x$ , deles op i faste blokke  $x_i$ , med bit længden  $r$ . Denne første del af processen indeholder typisk tilførelsen af ekstra bits (padding), som er nødvendigt for at opretholde den samlede hash bit længde, som er et multiplum af blok længden  $r$ . Hver blok  $x_i$  processeres så af en intern fast længde hash funktion  $f$ , som er  $h$ 's kompressions funktion. Hvert delresultat  $x_i$  kædes sammen med det foregående  $x_{i-1}$ . Når alle blokke er processeret, mappes den  $n$  bit kædede variabel over i et  $m$  bit resultat.

Nogle specifikke hash funktioner udskiller sig dog fra denne procedure.

## 6.3 SHA-1

I gruppen af unkeyed hash funktioner (og MDCs) findes der generelt 3 store kategorier af iterative hash funktioner. Hash funktioner baseret på blok cifre, specialbyggede hash funktioner og hash funktioner baseret på modulær matematik. Hash funktionerne baseret på blok cifre bygger ofte videre på et eksisterende system, hvor et blok ciffer allerede er effektivt implementeret. Sådanne funktioner tæller fx. MDC-2 og MDC-4. Hash funktioner baseret på modulær matematik bygger, som navnet siger, på modulære operationer. Ideen er at genbruge et vigtigt element i de fleste asymmetriske krypterings systemer og på den måde vinde hastighed. Sådanne funktioner tæller bl.a. MASH. De specialbyggede hash funktioner er opbygget fra bunden og ofte designet til hashing med hurtighed i mente, hvor man således ikke er bundet af at genbruge eksisterende systemer. De mest udbredte sådanne funktioner er baseret på MD4.

MD4 (message digest 4) er nummer 4 i en række af hash funktioner designet af R. Rivest. MD4 blev designet til, i forfatterens egne ord, at køre ekseptionelt hurtigt implementeret i software på 32 bit maskiner. MD4 er en 128 bit hash funktion. De oprindelige mål for MD4 var (udover hastighed), at finde en besked med den samme hashværdi skulle tage ca.  $2^{64}$  operationer, og at finde en besked udfra en hash værdi skulle tage ca.  $2^{128}$  operationer. Det er nu kendt, at MD4 ikke opfylder disse oprindelige krav/mål. Sikkerhedshensyn motiverede Rivest til at designe MD5 kort tid efter, som en mere konservativ variation af MD4. Andre udspringere af MD4 tæller RIPEMD og de forstærkede varianter RIPEMD-128 og RIPEMD-160 samt Secure Hash Algorithm (SHA-1). Tabel 3 og 4 viser forskellige egenskaber for udvalgte hash funktioner.

Navn	Bit længde	Runder $\times$ skridt pr. runde	Relativ hastighed
MD4	128	$3 \times 16$	1.00
MD5	128	$4 \times 16$	0.68
RIPEMD-128	128	$4 \times 16$ (parallelt)	0.39
SHA-1	160	$4 \times 20$	0.28

Tabel 3: Udvalgte hash funktioners egenskaber.

Navn	Streng	Hash værdi (hex byte)
MD4	"a"	31d6cfe0d16ae931b73c59d7e0c089c0 bde52cb31de33e46245e05rbdbd6fb24
MD5	"a"	d41d8cd98f00b204e9800998ecf8427e 0cc175b9c0f1b6a831c399e269772661
SHA-1	"a"	da39a3ee5e6b4b0d3255bfef95601890afd80709 86f7e437faa5a7fce15d1ddcb9eaeaea377667b8

Tabel 4: Test af hash funktioner.

Secure Hash Algorithm blev designet af the U.S. National Security Agency (NSA) i forbindelse med Digital Signature Algorithm (DSA), og blev foreslået som standard for visse amerikanske regerings applikationer af the U.S. National Institute of Standards and Technology (NIST). De største forskelle mellem MD4 og SHA-1 er de følgende:

1. Hash værdien er 160 bits lang og 5 (mod MD4's 4) 32-bit kædede variabler benyttes.
2. Kompressions funktionen har 4 runder i stedet for 3, og benytter de 3 delfunktioner  $f$ ,  $g$  og  $h$  således:  $f$  i runde 1,  $g$  i runde 2 og  $h$  i runde 3 og 4. Hver runde har 20 skridt istedet for 16.
3. I kompressions funktionen bliver hver 16-ord (word) blok udvidet til en 80-ord blok af en process, hvor hver af de sidste 64 (af 80) ord er en XOR af 4 ord fra tidligere positioner, i den udvidede blok. Disse 80 ord er så input, et ord pr. skridt, til de 80 skridt.
4. Kerne skridtet er modificeret som følger: den eneste rotation benyttet, er en konstant 5 bit rotation; Den femte variabel adderes til hvert delresultat; Ord fra den udvidede blok er benyttet sekventielt; og  $C$  opdateres som  $B$  roteres 30 bit til venstre, istedet for bare at rotere  $B$ .
5. SHA-1 benytter 4 additive konstanter forskellige fra nul, hvor MD4 kun benytter 3 konstanter, hvoraf kun 2 er forskellige fra nul.

Algoritmen udmærker sig (som andre hash funktioner) ved at sprede effekten af et enkelt ændret tegn i klarteksten (diffusion) til resten af ciffer teksten (hash værdien). For eksempel vil en ændring i ord 14, i den indledende udvidelse af 16 ord til 80, direkte ændre på ord 17, 22, 28 og 30. Ændringen på ord 17 vil så ændre på ord 20, 25, 31 og 32. Ændringen på ord 22 vil ændre ord 25, 30, 36 og 41, og så videre. Dvs. ændringen i ord 14 vil få indflydelse på ord 17, 20, 22, 25, 28, 30, 31, 33, 34, 36, 37, 39, 41, 42, 44, 45, 47, 48, 49, osv, og det bare i den indledende runde. Denne effekt kaldes for lavine effekten.

## 7 Nøgledistribution

Digitale signaturer kan sikre integritet af data og autentificering af afsenderen. For at dette er anvendeligt er det nødvendigt med en måde at distribuere de offentlige nøgler på. For at hindre andre i at lade som om de er en anden er det nødvendigt at få den offentlige nøgle fra et pålideligt sted. En privat person kunne således lægge sin offentlige nøgle på sin hjemmeside hvorfra folk så kunne hente den. Denne løsning vil dog ikke skalere da det hurtigt vil være uoverskueligt at få fat i nøgler. For at komme ud over dette problem er der flere løsningsmodeller.

Den mest populære måde er at tildele visse institutioner retten til at certificere andre også kaldet *Certification Authority* eller bare CA. Et eksempel på en CA er VeriSign eller det danske KMD-CA, der er begyndt at udstede personlige certificater. Dette foregår ved at man efter at have vist passende identifikation får et certifikat der består af den offentlige nøgle plus evt. anden information, der så er krypteret med CA'ens private nøgle. Dette garanterer at nøgle ejerens identifikation er blevet kontrolleret af CA'en.

En anden løsning som PGP benytter er at lade andre brugere signere en offentlig nøgle og derved garantere at identiteten af ejeren er korrekt. Denne metode gør en CA overflødig, men gør også at det er svært at kontrollere en nøgle. Det er ikke umiddelbart klart hvem man kan stole på og hvem man ikke kan. Hvis der skal oprettes en kæde af kendte personer fra en selv til ejeren af nøglen, bliver anvendelses mulighederne kraftigt begrænset.

Ved begge løsninger kan offentlige nøgler samles på nøgle distributions servere hvorfra de kan hentes.

## 8 Sikkerhed ved brug af Smart Card

Overordnet er der kun én årsag til at Java smart cards overhovedet er relevante at anvende til sikkerheds relaterede operationer. Årsagen er det simple faktum at klienter generelt er usikre. En meget stor del af klienter er idag forbundet i en eller anden form for netværk, hovedsagligt via Internettet. Dette har gjort det stadig lettere at anskaffe nye programmer. Disse programmer er ligesom tidligere programmer naturligvis ikke fejlfri, men netværksteknologien gør det muligt for en hacker at gøre alskens ting ved den enkelte netværksklient. Det kan være ligefra inficering med simple vira, til trojanske heste eller introduktion af anden usikker kode. Dette kan resultere i næsten alt fra simple ubehagligheder over databaser til vidersending af data fra klienten. Disse muligheder for hackere gør at det meste af selve klienten må anses for usikker, når der skal opbevares strengt personlige oplysninger. Derfor kan der med fordel anvendes smart cards som en form for *black boxes*. I tilfælde hvor sikkerheden på værten bliver kompromiteret, vil den private nøgle stadig være i sikkerhed på kortet, da den aldrig lagres på værtens harddisk eller i hukommelsen. Det hjælper da heller ikke yderligere for sikkerheden at dagens mest udbredte operativsystem ikke er kendt for at prioritere sikkerheden særlig højt og øjensynligt helst vil implementere den begrænsede sikkerhed igennem *security by obscurity*. Det nyeste tiltag i denne retning er forsøg på ikke at offentliggøre fejl. Efter den senere tids udvikling i USA har de fået yderligere skyts til at følge en sådan kurs.

En løsning baseret på smart cards er naturligvis ikke 100% sikker. Som ved enhver anden netværks kommunikation er det et problem at sikre sig at den der benytter netværks ressourcer, virkelig er den den udgiver sig for at være og at det dermed er den der er blevet tildelt rettighederne til at benytte disse ressourcer. Altså et klassisk autentificerings problem. Som tidligere nævnt kunne det foregå vha. PIN-koder og/eller kryptering. Dette problem er bedre kendt som terminal problemet.

### 8.1 Terminal/klient problemet

Antagelsen om at klienten er usikker giver dog anledning til den noget paradoksale situation at et smart card må være ubrugeligt. For hvis man ikke kan stole på klienten, hvordan kan man så stole på det klienten påstår at kortet siger? For at mindske dette problem, har vi valgt ikke at implementere nogle metoder der giver værten adgang til den private nøgle. Denne skulle derfor være sikret imod et sådan angreb.

At sikre benyttelsen af signerings mulighederne er et langt mere vanskeligt problem. Her kunne man forestille sig at et snedigt lille program fik infiltreret signatur applikationen på klienten. Hackerens program kunne således aflure PIN koden. Derefter kunne det f.eks. modtage den udregnede hashværdi af dokumenter eller anden data der efterfølgende kunne signeres på kortet. Da det kun er en ganske kort hash værdi der skal krypteres ville den ekstra netværkstrafik næppe være mistænksom og hvis blot kortet blev siddende i kortlæseren kunne det hele tiden benyttes. At beskytte sig mod en sådan svaghed ville være overordentlig vanskelig.

## 8.2 Fysiske angreb

Samme muligheder kunne opnås ved fysisk at stjæle kortet og efterfølgende bryde PIN koden. Men med et smart card, med blot nogen grad af sikkerhed ville et sådan forsøg mislykkes. Kortet burde simpelthen blokere sig selv ved for mange mislykkede forsøg på at indtaste den korrekte PIN kode. Derfor ville denne mulighed alene ikke være så eftertragtet, det er her at fysiske angreb kommer ind i billedet.

De to primære fysiske angrebsformer er begge differential analyser. Den simpleste er differential fejl analyse, denne virker ved at der indføres beregningsmæssige fejl på kortet. Dette kan gøres relativt simpelt ved at regulere chippens temperatur, strøm tilførsel eller klok rate. Endelig kan det gøres ved at udsætte chippen for bestråling eller bruge noget så simpelt som en gummihammer. Ved hjælp af differential analyse af de introducerede fejl kan man derefter udlede en evt. privat nøgle eller anden sensitiv information.

En noget mere avanceret form for differential analyse er differential strøm analyse. Denne form for analyse virker ved at observere strømforbruget ved forskellige beregninger. Denne analyse kan ligesom differential fejl analyse blotlægge private nøgler og lignende. Denne angrebs metode kan dog imødegås ved at indføre noget digitalt støj. Dette kunne f.eks. gøres vha. af tilfældige beregninger der blandes med de reelle beregninger. Eller at rækkefølgen af de nødvendige beregninger randomiseres.

Generelt er det ikke muligt at lave et kort der er 100% sikkert overfor fysiske angreb hvis angriberen har tilstrækkelige tekniske og økonomiske faciliteter til rådighed. Dette kan dog gøres så svært ved en korrekt konstruktion at det bliver overodentligt svært og ikke længere kan betale sig. Der er dog stadig ingen garanti for at ingen vil finde en ny teknik der vil gøre det lettere at opdage kortets indhold.

## 8.3 Mulige forholdsregler

De fysiske angreb er i vores tilfælde ikke noget væsentligt problem sålænge kortindehaveren stadig er i besiddelse af kortet. Var der derimod tale om et kort, hvorpå der var penge, var problemet et helt andet. I så tilfælde kan man ikke anse kortindehaveren for pålidelig. Hvis han finder ud af at ændre på kortet kunne han f.eks. indsætte penge på kortet og derved lave en pengemaskine. Skulle kortet bruges som et adgangskort kunne det også medføre problemer såfremt at selve rettighederne var lagret på kortet så det kunne benyttes offline. I disse tilfælde ville det være nødvendigt at indføre ekstra sikkerhed fra kortudstederens side. Dette kunne f.eks. gøres ved at ændre krypteringsnøgler og/eller krypteringsalgoritmer regelmæssigt.

Hvis en anden fik adgang til vores signatur kort og brød PIN koden kunne der således signeres data i en andens navn. For at sikre imod dette kunne en *timestamping service* benyttes således at signaturen skulle indeholde et tidsstempel. Dette skulle naturligvis kombineres med at CA'er havde mulighed for at kende signaturer der er lavet efter et vist tidspunkt for ugyldige hvis kortet var meldt tabt.

En implementation af en *timestamping service* er dog udenfor dette projekts område, men kort sagt ville det medføre visse problemer idet at det ville kræve online adgang. En pålidelig *timestamping* der kører lokalt er ihvertfald umiddelbart svær at forestille sig.

En anden mulig løsning ville være at implementere et lille display og et tastatur på ko-

rtlæseren, sådan at disse ikke skulle indtastes på den usikre vært. Dette ville naturligvis øge sikkerheden væsentligt, men det ville stadig ikke være perfekt. Der vil således stadig ikke være nogen garanti for at det man i så fald signere også er netop det man ønsker at signere. Dette kunne dog imødegås ved at vise hashværdien i displayet også, men det ville kræve at brugeren så sammenligner den med den der vises på skærmen. Dette kunne dog relativt let lade sig gøre ved f.eks. at vise talværdierne af de bytes der skal signeres. En sådan løsning ville gøre systemet endnu mere tungt at benytte.

Hvis man forestillede sig at man modtog et dokument der var blevet ændret undervejs og havde fået en ny signatur, med en andens private nøgle behøver man en måde at sikre sig at den offentlige nøgle der anvendes også virkelig er den man forventer. Under antagelse af at værts applikationen er til at stole på kan dette opnås vha. certifikater fra en CA og kryptering af kommunikationen, men hvis værts applikationen ikke er til at stole på er der ingen garanti.

Tillid kan således ikke alene bygges på tillid til terminalen. For at opnå sikkerhed og tillid er det nødvendigt også at have tillid til værts applikationen. Ellers er der ingen garanti for at de korrekte data bliver signeret. Tilliden til værts applikationen behøver ikke at være så stor som tilliden til terminalen og dette er årsagen til at den private nøgle kun lagres på kortet. Hvis denne forudsætning er til stede er det ikke nødvendigt at stole på det underliggende operativ system, da asymmetrisk kryptering kan anvendes for at sikre at det korrekte data når frem til signering og tilbage igen.

Tillid til terminalen dækker over en antagelse om at nøglegenereringen fungerer tilfredsstillende på kortet. Ved alle kørsler har det vist sig at kortet genererer en nøgle med krypterings eksponenten 65537. Som tidligere nævnt er dette ikke umiddelbart noget problem for sikkerheden.

## 9 Implementationen

### 9.1 Formål

Målet for projektet var at implementere en signatur applet der kunne afprøves i en Java Smart Card simulator. Udformingen af applikationen blev i første række afgjort af sikkerhedshensyn. For at øge sikkerheden valgte vi at den private nøgle kun skulle lagres på kortet og at det ikke skulle være muligt at få nøglen ud af kortet. Er nøglen først overført til klienten forværres sikkerheden markant. For at øge sikkerheden yderligere genereres nøglerne på kortet, når appletten installeres derpå. Disse kunne naturligvis også genereres af en anden maskine i et sikret miljø for derefter at blive overført til kortet. Dette ville medføre en betydelig tidsbesparelse ved installation af det enkelte kort.

Kortets begrænsede beregningskraft gjorde, at vi valgte at udregningen af hashværdi skulle foregå på computeren, da det tidsmæssigt sandsynligvis ville være helt uholdbart hvis store datamængder skulle overføres til kortet, for slet ikke at nævne problemerne med at udregne hashværdier i den meget begrænset hukommelse. Det var således vores mål at være i stand til at signere store datamængder.

Det skulle naturligvis være muligt for andre at verificere en given signatur. Derfor kan kortet udlevere den offentlige nøgle, så denne kan distribueres. Denne eksporteres til en fil, der så kan distriberes på passende vis, f.eks. via en CA.

Endelig besluttede vi at krypteringen med den offentlig nøgle skulle foregå på computeren og ikke på kortet. Dette ud fra betragtningen at med den begrænsede beregningskraft på kortet så var der ingen grund til at foretage flere beregninger dér end højst nødvendigt. Dette ville også være naturligt når en signatur skulle verificeres af en anden end den signaturen tilhører.

### 9.2 Problemer

På grund af de amerikanske eksportregler er det desværre ikke muligt for Sun Microsystems at lave en implementation af Java<sup>TM</sup> Cryptography Extension 1.2 der kan hentes uden for USA. Det er dermed heller ikke muligt at have én kryptografi implementation for Java Smart Cards. Derfor er der forskelle i implementationerne og i en række af små tilfælde manglede vi en nøjagtig API for netop de kryptografiske funktioner, der var implementeret af GemPlus. Det var så simple ting som navnene på de implementerede algoritmer, at input skal være padded og problemer omkring Javas konvertering til signed datatyper. Bedre dokumentation ville have sparet os for megen besvær. Den lettere mangelfulde dokumentation og ikke 100% overholdelse af specifikationerne kunne også have gjort det mindre besværligt at implementere klient siden. Vi valgte at benytte os af Cryptix 3 implementationen af JCE, denne kan hentes gratis fra <http://www.cryptix.org>. Det viste sig at det ikke var helt problem frit at få det til at virke ordentlig sammen med vores applikation. Men problemerne omkring kryptografi var dog ikke de største problemer. Selve installationen af smart card simulatoren voldte uforholdsvis mange problemer og blev overordentlig tidskrævende. Efter megen tid spildt virkede simulatoren pludselig og det uden at nogen af os egentlig fandt ud af hvad der havde været galt. Der kan dog stadig være problemer med at køre applikationen på et kort, men det ser ud til at være et problem omkring opsætningen af selve kortlæseren og ikke i applikationen som sådan.

### 9.3 Oversigt over implementationen

Applikationen er delt op i tre klasser. Kommunikationen mellem kort og computer er placeret i de to klasser `SignatureClient` og `SignatureApplet`, mens den generelle benyttelse af programmet er placeret i klassen `Signature`. Denne inddeling betyder en klar arbejdsdeling mellem klasserne. For at gøre delingen endnu mere tydelig kunne selve interaktionen med brugeren være skilt ud i en særskilt klasse. Til klasserne `SignatureClient` og `SignatureApplet` har vi valgt at bruge en kode skabelon fra `GemPlus`. Dette har vi valgt for at kunne fokusere på at udvikle applikationen.

Når applikationen startes installeres appletten først på kortet vha. `install` metoden i `SignatureApplet`. Dette ville selvfølgelig være overflødig i en rigtig applikation, da kortet i så fald kun skal installeres en enkelt gang. Under installationen genereres nøgleparet. Efter at nøglerne er genereret og installationen overstået er kortet nu klar til brug.

Herefter har applikationen tre primære funktioner:

- Eksportering af offentlig nøgle til filen `key`.
- Signering af data til filen `signature`.
- Verificering af signatur fra filen `signature`, mod data fra fil og nøgle fra filen `key`.

Et eksempel på program kørsel kan ses af testkørslerne i appendix.

### 9.4 Yderligere forbedringer

I den implementerede applikation mangler der stadig nogen funktionalitet der især omhandler sikkerheden. For at sikre at det kun er den retmæssige ejer af kortet, der benytter det til at signere med, mangler der en eller anden form for autentificering. Det mest naturlige ville selvfølgelig være også at benytte en PIN kode her. Dette er i forvejen næsten implementeret, men det er stadig muligt at signere uden at give den korrekte PIN kode. For at øge sikkerheden yderligere kunne der i stedet benyttes en eller anden form for kodeord. Dette ville dog ikke være helt simpelt at implementere. Hvis man anser kortet for at være det eneste helt sikre sted, så ville det ikke umiddelbart være muligt at tjekke at kodeordet ikke er et alt for let ord, der sagtens kunne brydes af hackere udfra simple statistiske antagelser. Men anses værts applikationen også for at være sikker, er det ikke noget væsentligt problem. Dog er en talsekvens lettere at huske end en lang streng af helt tilfældige karakterer.

I en sådan løsning ville det være et krav at kommunikationen mellem kortet og værtsapplikationen var krypteret for at hindre at en tredje part kan få adgang til PIN koden. Vi kunne have prioriteret dette højere og implementeret det i applikationen.

Endelig kan det indvendes at fejlstyring, hjælpesystemet og konfigurationsmulighederne skulle have været udbygget væsentligt mere. Konfigurations mulighederne er forsøgt gjort lettere tilgængeligt og er således ikke *hard coded*, men findes i filen `config.cfg`.

På grund af projektets fokus og af tidsmæssige årsager har det desværre ikke været muligt at implementere dette.

## 10 Installationsvejledning

Vi har forsøgt at gøre installationen af applikationen så simpel som mulig. Men efter de problemer vi selv oplevede med at installere GemXpresso RAD III, så kan vi naturligvis ikke garantere at dette er fuldstændig fyldestgørende. Her følger en kort installations vejledning.

- Installér JBuilder 4 eller JDK 1.3.(kun JDK 1.3. er ikke testet)
- Installér GemPlus GemXpresso RAD III
  - Installér Kernel.
  - Installér Card Profiles - GemXpresso 211 PK.
  - Installér Service Pack - Full Cryptography Pack.
- Installér Cryptix 3.
  - Kan hentes fra:  
<http://www.cryptix.org/dist/cryptix32-20001002-r3.2.0.zip>
  - Cryptix32.jar skal ligge i biblioteket c:\cryptix\
- Hent Signature projektet.
  - Projektet kan hentes fra:  
<http://www.it-c.dk/~aggeboe/signature.zip>
  - Udpak zip filen til roden af C drevet. Hvis dette ændres skal config.cfg opdateres. (Ikke testet).

Dette skulle gerne resultere i at kompileret bytecode, kildekode samt andre projekt filer bliver udpakket til biblioteket: c:\it-c\jsc\signature\. For at køre test filen til projektet følges nedenstående vejledning:

- Fra signature biblioteket køres filen GSERunClient.bat. Bemærk: Der skal være en kortlæser tilsluttet computeren.
- Alternativt kan applikationen køres fra JBuilder 4.
  - Vælg File - Open Project - Vælg projektet c:\it-c\jsc\signature\signature.jpr
  - Vælg Run - Run Project
  - Under Run - Configurations - Run - Application parameters skulle følgende gerne fremgå: -f test -s signature -k key -p c:\jsc\signature\ -selftest

Dette test program laver følgende operationer:

- Eksportere kortets offentlige nøgle til filen key.
- Signere filen test og udskriver resultatet til filen signature.
- Der ændres en karakter i signatur filen.
- Filen signature forsøges verificeret mod filen test, med nøgle fra filen key.

- Filen test signeres påny.
- Den uændrede signatur forsøges igen verificeret.

En udskrift af en sådan test kørsel er vedlagt i appendix ???. Bemærk at det ikke er muligt at genskabe en nøjagtige kopi af denne testkørsel, da der genereres en ny nøgle hver gang appletten installeres og det krypterede data vil derfor være forskellig fra gang til gang.

Yderligere hjælp til benyttelse af applikationen kan fås ved på kommandolinien at tilføje argumentet -help.

## 11 Konklusion

I projektforsøget lykkedes det at implementere en simpel signatur applikation. Der er istand til at signere data og verificere en given signatur.

Den primære årsag til at benytte et smart card til at implementere applikationen var at øge sikkerheden ved en sådan løsning. Men er sikkerheden øget? Hvis alle de nødvendige punkter bliver implementeret og antagelserne holder må det konkluderes at sikkerheden er øget. Hvor meget sikkerheden er øget er svært at afgøre. Som tidligere beskrevet er det muligt at benytte sig af andres signatur uden at have direkte adgang til kortet. For at forhindre dette skulle applikationen på værten være fuldstændig sikker. Dette er naturligvis ikke tilfældet, men sikkerheden kunne være tilstrækkelig til at værts applikationen er pålidelig.

Sikkerheden i systemet bygger således på en række antagelser om hvad der er sikkert. Herunder især det paradoksale i på samme tid at have tillid til klienten og ikke at have det. Selvfølgelig skulle den private nøgle skulle være sikker på kortet, men det er kun en ringe trøst når det stadig er muligt at signere uden at have fysisk adgang til kortet. En bruger vil gøre klogt i ikke at have kortet siddende i kortlæseren hele tiden, men dette er stadig ingen garanti. Risikoen ligger i selve benyttelsen af kortet, hvilket er ret uheldigt. Men som med andre programmer er denne risiko på et niveau der er acceptabelt i praksis. Udbredelsen af handels og bank funktioner på Internettet indikerer også dette.

Allerede idag spiller digitale signaturer en vigtig, men ofte skjult rolle, ved transaktioner via Internettet. Et stadigt stigende antal websites benytter sig af sikker overførsel vha. *Secure Socket Layer* (SSL) protokollen. Denne benytter sig af certificater og herunder også digitale signaturer. Vha. disse certificater autentificere serveren sig over for klienten. Det er muligt for klienten også at autentificere sig, men dette er kun sjældent benyttet. Det er kun et fåtal af klienterne på Internettet der er i besiddelse af et personligt certifikat og dette kunne bl.a. skyldes nogle af de problemer omkring opbevaringen af nøglerne som er beskrevet tidligere. Mere sandsynligt er dog at det er problemet ved at certificere så mange private brugere. Herhjemme har KMD-CA startet et samarbejde med Post Danmark om udstedelse af certificater til private brugere. Indtil videre ser det dog ud til at disse stortset kun kan benyttes til at signere epost beskeder. En vigtig del af SSL er dog også at kommunikationen krypteres for at opnå fortrolighed. Den implementerede løsning giver ikke umiddelbart nogen form for fortrolighed, men garanterer udelukkende autentificering og integritet.

På trods af visse svagheder vil digitale signaturer utvivlsomt komme til at spille en stadig vigtigere rolle i fremtiden. Dette vil ske inden for mange områder, idag er de allerede vidt udbredt inden for bankverdenen. Initiativer som KMD-CA kunne også bruges som en start forudsætning for at der kunne gives adgang til det Offentliges informationer og til f.eks. at udfylde Selvangivelsen på Internettet med forøget sikkerhed.

Sikkerheden i sådanne systemer må nødvendigvis bygge på tilstrækkelig sikkerhed i alle led, både hvad angår kommunikationen, men også i udviklingen af de nødvendige programmer. Det ville være mest naturligt for en angriber at iværksætte et angreb mod det punkt hvor sikkerheden er svagest. Alligevel skal man ikke umiddelbart forestille sig at sikkerheden bliver implementeret fra applikationen og hele vejen ned i operativ systemet. Det er således en stor udfordring at opnå en tilstrækkelig sikkerhed på et system der i grun-

den er usikkert. Dette kræver en vel overvejet og implementeret kryptografisk protokol.

Tilstrækkelig sikkerhed kan kun opnås ved at analysere værdien af de beskyttede data op imod risikoen for at de bliver kompromiteret. For derefter at betale udviklere for at implementere denne sikkerhed og vedligeholde den. Prisen for at implementere tilstrækkelig sikkerhed er ofte overodentlig høj og når producenten vurderer at den ikke selv har det store at miste ved dårlig sikkerhed, så er der ikke umiddelbart nogen grund til at øge sikkerheden. Har en producent opnået en så dominerende position på markedet er det muligt at give risikoen videre til brugerne, der ikke har mange bedre alternativer. Analysen må således også indeholde parametre om værdien af data for brugerne.

# Litteratur

- [1] Chan, Siu-cheung Charles. "An Overview of Smart Card Security". <http://home.hkstar.com/~alanchan/papers/smartCardSecurity/>
- [2] Chen, Zhiqun. "Java Card <sup>TM</sup> Technology for Smart Cards. Architecture and Programmer's Guide". (Addison-Wesley, 2000).
- [3] McGraw, Gary. "Securing Java Getting down to business with mobile code". <http://www.securingjava.com/> (John Wiley & Sons, Inc.).
- [4] A. Menezes, P. van Oorschot, and S. Vanstone. "Handbook of Applied Cryptography". <http://www.cacr.math.uwaterloo.ca/hac> (CRC Press, 1996).
- [5] J. F. Kurose, K. W. Ross "Computer Networking - A Top-Down Approach Featuring the Internet"(Addison-Wesley, 2001).
- [6] C. P. Pfleeger "Security in Computing". (Prentice Hall, 2000).

# A Testkørsel

```
Gemplus Card Terminal Version: 4.02.001 Date: August 17, 2000
Did not find Gempluscardterminal.properties file
Uses OCF Polling list techniques
Don't uses TPDU

Found OCF Card Terminal:
- Name = Simulator
- Type = SOCKETJJC21SIMULATOR
- Adress = 127.0.0.1:5000

Found OCF Card Terminal:
- Name = gempc410_com1
- Type = GemPC410
- Adress = COM1

Security Providers installation...
- provider "cryptix.jce.provider.CryptixCrypto" was installed
- provider "com.sun.crypto.provider.SunJCE" was not installed

Checking security providers...
- found provider 1: SUN
- found provider 2: CryptixCrypto_GemXpresso

DES/3DES feature found in provider: "CryptixCrypto_GemXpresso"
Working with FULL DES/3DES feature...

Using static limited keys for default key set...

Simulator is listening on port 5000...
GemXpresso RAD III
Version: 3.0 Build 037
Package: "com.gemplus.javacard.gse"
- description: GemXpresso Simulation Environment Base
- version: 3.18 May 15, 2001
Package: "javacard.framework"
- description: GemXpresso Simulation Environment Javacard API 2.1.1
- version: 1.02 February 22, 2001

GXP211_PK_IS

Socket connected...

< Card Reset >ATR returned by the GSE...

< Card Reset ><-- Card : 3B 0D 80 31 80 65 B0 05 01 02 F0 83 00 90 00

< Selecting "CardManager" >
< Select > received...
< Initialize Update > received...
- initializing secure session...done, waiting "External Authenticate"
< External Authenticate > received...
- verifying external authenticate...session is secured
Authentication OK

Initialize OP global PIN

< Pin Change/Unblock > received...
- decrypting PIN
- reset and unblock PIN
- updating PIN value
PIN initialisation OK

Starting load process...
Loading the GSE...

< Install > received...
- load is initialized
< Load > received...
```

```

- buffering 200 bytesblock 1 loaded
< Load > received...
- buffering 200 bytesblock 2 loaded

< Load > received...
- buffering 200 bytesblock 3 loaded

< Load > received...
- buffering 200 bytesblock 4 loaded

< Load > received...
- buffering 200 bytesblock 5 loaded

< Load > received...
- buffering 200 bytesblock 6 loaded

< Load > received...
- buffering 200 bytesblock 7 loaded

< Load > received...
- buffering 200 bytesblock 8 loaded

< Load > received...
- buffering 200 bytesblock 9 loaded

< Load > received...
- buffering 200 bytesblock 10 loaded

< Load > received...
- buffering 200 bytesblock 11 loaded

< Load > received...
- buffering 200 bytesblock 12 loaded

< Load > received...
- buffering 200 bytesblock 13 loaded

< Load > received...
- buffering 200 bytesblock 14 loaded

< Load > received...
- buffering 191 bytes
- load binary...successfully loadedblock 15 loaded
package loaded in 0 s 390 ms
Loading SignatureClient succeeded
Install applet...

< Install > received...
- searching AID...applet "signature.SignatureApplet" found
- installing applet...registration Ok
Installing application SignatureClient succeeded
Select application...

< Selecting "SignatureApplet#0" >
< Command to "SignatureApplet#0" process >Select application SignatureClient OK

Performing self test

Exporting public key

< Command to "SignatureApplet#0" process >--> Term : 90 11 01 00 43
<-- Card : 9B CB BE AA 59 ED CD 82 BE 5B 70 90 6B FF A2 F3 38 12 4D B2 A3 03 67 11 F5
D9 57 5D BF A6 6D 9B 86 DC BA B1 EE 35 EF 86 04 A1 46 55 A2 7D E6 D5 E3 B0 98 4E DD
B6 26 D8 20 2C 32 33 A7 3E B7 C5 90 00

Modulus obtained from card:
Ë¼?Yií¼[p?ky?ó8^RM²$^Cg^QöÜW]¿|mÜ?±i5i^D¡FU?}æÕã?NÝ¶&Ø ,23§>·Å

< Command to "SignatureApplet#0" process >--> Term : 90 11 00 00 40

```



3

Reading from file: c:\it-c\jsc\signature\key

Result (string): ^A@^A

Result (hex):

1 0 1

Reading from file: c:\it-c\jsc\signature\key

Result (string): Ě¼?Yif¼[p?kÿ?ó8^RM²\$^Cg^QöÜW]¿|mŮ?±î5i^D;FU?}æŌã?NÝ¶&ø ,23§>-Ä

Result (hex):

9B CB BE AA 59 ED CD 82 BE 5B 70 90 6B FF A2 F3 38 12 4D B2 A3 3 67 11 F5 D9 57 5D BF  
A6 6D 9B 86 DC BA B1 EE 35 EF 86 4 A1 46 55 A2 7D E6 D5 E3 B0 98 4E DD B6 26 D8 20  
2C 32 33 A7 3E B7 C5

Hash value computed from source file: áq^A^[^BÖ(Çd3?^R?\táP^Há?

E1 71 1 1B 2 D6 28 C7 64 33 8E 12 B0 5C 74 E5 50 8 E2 A4

Hash value encrypted from signature file: ©ÿ?k\_?bynlÈXÄ·viÈuU

A9 FF 97 B0 6B 5F 8D 62 79 6E 31 C8 D7 C4 B7 76 EC CB 75 55

Verification status: FAILED!

Signing: test...

Signing file: c:\it-c\jsc\signature\test

Read data file: c:\it-c\jsc\signature\test ...

Compute hash value for file: c:\it-c\jsc\signature\test ...

Hash value (string): áq^A^[^BÖ(Çd3?^R?\táP^Há?

Hash value (hex): E1 71 1 1B 2 D6 28 C7 64 33 8E 12 B0 5C 74 E5 50 8 E2 A4

< Command to "SignatureApplet#0" process >--> Term : 90 12 00 00 40 01 E1 71 01 1B 02  
D6 28 C7 64 33 8E 12 B0 5C 74 E5 50 08 E2 A4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00  
<-- Card : 42 FB 82 10 14 A5 C6 52 B5 5D 34 7A A5 AA 2C A1 5A A8 55 20 3A 43 A4 5D CD  
9F 3B 58 3C A4 C7 CA 6B CE 3B 1B 4C DD 1B 93 A1 27 80 9C A1 CF 9F B0 96 19 3D 74 93  
85 1B 67 B6 2C CD 0C 76 58 ED 94 90 00

Generating & writing file: signature ...

Verifying signature from: signature against: test

Read data file: c:\it-c\jsc\signature\test ...

Compute hash value for file: c:\it-c\jsc\signature\test ...

Hash value (string): áq^A^[^BÖ(Çd3?^R?\táP^Há?

Hash value (hex): E1 71 1 1B 2 D6 28 C7 64 33 8E 12 B0 5C 74 E5 50 8 E2 A4

Reading from file: c:\it-c\jsc\signature\signature

Result (string): BŮ^P^T?ÆR?]4z??,;z"U :C?]Í;X<?ÇÊkÍ;^[LÝ^[;';;İ?^Y=t^[g¶,Í^LvXí

Result (hex):

42 FB 82 10 14 A5 C6 52 B5 5D 34 7A A5 AA 2C A1 5A A8 55 20 3A 43 A4 5D CD 9F 3B 58  
3C A4 C7 CA 6B CE 3B 1B 4C DD 1B 93 A1 27 80 9C A1 CF 9F B0 96 19 3D 74 93 85 1B 67  
B6 2C CD C 76 58 ED 94

Reading from file: c:\it-c\jsc\signature\key

Result (string): ^C

Result (hex):

3

Reading from file: c:\it-c\jsc\signature\key

Result (string): ^A^@^A

Result (hex):

1 0 1

Reading from file: c:\it-c\jsc\signature\key

Result (string): Ę¼?Yif¼[p?kÿ?ó8^RM²\$^Cg^QöÜW]¿|mŮ?±î5i^D;FU?}æŌã?NÝ¶&ø ,23§>-Ã

Result (hex):

9B CB BE AA 59 ED CD 82 BE 5B 70 90 6B FF A2 F3 38 12 4D B2 A3 3 67 11 F5 D9 57 5D BF  
A6 6D 9B 86 DC BA B1 EE 35 EF 86 4 A1 46 55 A2 7D E6 D5 E3 B0 98 4E DD B6 26 D8 20  
2C 32 33 A7 3E B7 C5

Hash value computed from source file: áq^A^[^BÖ(Çd3?^R?\táP^Há?

E1 71 1 1B 2 D6 28 C7 64 33 8E 12 B0 5C 74 E5 50 8 E2 A4

Hash value encrypted from signature file: áq^A^[^BÖ(Çd3?^R?\táP^Há?

E1 71 1 1B 2 D6 28 C7 64 33 8E 12 B0 5C 74 E5 50 8 E2 A4

Verification status: OK!

Connection closed !

----- end of Signature application -----

## B Signature.java

```
1  /**
2  *-----
3  * Project name      : Digital Signature with Java Smart Card
4  *                   - Java Card Open Platform applet -
5  *
6  * Platform         : Java virtual machine
7  * Language         : 1.3.0-C
8  * Devl tool        : Borland (c) JBuilder 4.0
9  *
10 * Original author  : Nikolaj Aggeboe & Sune Kloppenborg Jeppesen
11 * Date            : Thu Dec 20 10:40:16 CET 2001
12 *-----
13 */
14
15 package signature.client;
16
17 import java.io.*;
18
19 // standard OCF imports
20 import opencard.core.service.*;
21 import opencard.core.terminal.*;
22 import opencard.opt.applet.AppletID;
23
24 // security and crypto
25 import opencard.opt.security.*;
26 import java.security.*;
27 import java.math.BigInteger;
28
29 class Signature {
30     private static String path;
31     private static String signatureFile;
32     private static String keyFile;
33     private static String sourceFile;
34     private static boolean selfTest = false;
35     private static SignatureClient client;
36
37     public Signature() {
38     }
39
40     private byte[] hash(byte[] data, String file) {
41         byte[] hash = new byte[20];
42
43         if (selfTest) System.out.println("\nCompute hash value for file: " + file + " ...");
44         try {
45             //SHA-1 Hash
46             MessageDigest sha = MessageDigest.getInstance("SHA-1");
47
48             sha.update(data);
49
50             hash = sha.digest();
51         } catch (Exception e) {
52             System.err.println("Caught exception " + e.toString());
53         }
54         if (selfTest) System.out.println("\nHash value (string): " + new String(hash));
55         if (selfTest) System.out.print("\nHash value (hex): ");
56         if (selfTest) print(hash);
57         return hash;
58     }
59
60     private void sign(String filename) {
61         if (selfTest) System.out.println("\nSigning file: " + filename);
62         //Read file
63         byte[] data = readFile(filename);
64         //Compute HASH
65         byte[] signature = hash(data, filename);
66         //Pad the signature to 64 bytes
67         byte[] signaturepadded = new byte[64];
68         System.arraycopy(signature, 0, signaturepadded, 1, 20);
69         //Set the last bit of the first byte to avoid negative number and dropping of the whole byte
70         //this is not necessary anymore but padding scheme is not changed
```

```

71     signaturepadded[0] = 0x01;
72     try {
73         //Encrypt HASH value on Smartcard
74         signature = client.decrypt(signaturepadded);
75     } catch (CardTerminalException e) {
76         System.err.println("Could not read card " + e.toString());
77     }
78     //Write signature file
79     writeSignatureFile(signature);
80 }
81
82 private boolean verify(String src, String dst) {
83     boolean out;
84     byte[] md_data = new byte[20];
85     byte[] md_sign = new byte[20];
86
87     //Compute Hash for data file
88
89     md_data = hash(readFile(src), src);
90
91     //Read the line in Signature.txt with the encrypted hash
92     try {
93         //Read crypted HASH value from signature file
94         byte[] cryphash = readSignature(dst);
95
96         //Encrypt HASH value
97         md_sign = encrypt(cryphash);
98
99     } catch (Exception e) {
100         System.err.println("Caught exception " + e.toString());
101     }
102
103     String s1 = new String(md_data);
104     String s2 = new String(md_sign);
105     if (selfTest) System.out.println("\nHash value computed from source file: " + s1);
106     if (selfTest) print(md_data);
107     if (selfTest) System.out.println("\nHash value encrypted from signature file: " + s2);
108     if (selfTest) print(md_sign);
109
110     //Compare the 2 strings
111     if(s1.equals(s2)) {
112         if (selfTest) System.out.println("\nVerification status: OK!");
113         out = true;
114     } else {
115         if (selfTest) System.out.println("\nVerification status: FAILED!");
116         out = false;
117     }
118     return out;
119 }
120
121 private byte[] encrypt(byte[] input) {
122     BigInteger publicExponent = null;
123     BigInteger publicModulus = null;
124     BigInteger publicModulusMask = null;
125     BigInteger message = new BigInteger(input);
126     BigInteger result;
127     //B for byte
128     byte[] publicModulusB = new byte [65];
129     byte[] publicModulusUnsignedB = new byte[64];
130     byte[] messageB = new byte[65];
131     byte[] outB = new byte[20];
132
133     //Get public RSA key
134     publicExponent = new BigInteger(this.readExponent(path + "key"));
135     publicModulusUnsignedB = this.readModulus(path + "key");
136     //Shift the position of the bits 8 positions to make Java treat it as an unsigned number
137     //Could also have used a bit mask instead
138     System.arraycopy(publicModulusUnsignedB, 0, publicModulusB, 1, 64);
139     System.arraycopy(input, 0, messageB, 1, 64);
140     publicModulus = new BigInteger(publicModulusB);
141     message = new BigInteger(messageB);
142     //encrypt input value

```

```

143     result = cryptix.provider.rsa.RSAAAlgorithm.rsa(message, publicModulus, publicExponent);
144     //remove the padding
145     System.arraycopy(result.toByteArray(), 1, outB, 0, 20);
146     return outB;
147 }
148
149 private byte[] readFile(String file) {
150     byte[] in = null;
151     try {
152         FileInputStream infile = new FileInputStream(file);
153         in = new byte[infile.available()];
154
155         if (selfTest) System.out.println("\nRead data file: " + file + " ...");
156
157         infile.read(in);
158         infile.close();
159
160     } catch (Exception e) {
161         System.err.println("Caught exception " + e.toString());
162     }
163     return in;
164 }
165
166 private byte[] readSignature(String file) {
167     //skip to the correct position and read 64 bytes
168     return readFromFile(file, 317, 64);
169 }
170 private byte[] readModulus(String file) {
171     //skip to the correct position and read 64 bytes
172     return readFromFile(file, 318, 64);
173 }
174
175 private byte[] readExponent(String file) {
176     //skip to the correct position and read 1 byte that is the lenght of the exp
177     int length = (int) readFromFile(file, 317, 1)[0];
178     //skip to the correct position and read length bytes
179     return readFromFile(file, 318 + 64, length);
180 }
181
182 private byte[] readFromFile(String file, int offSet, int length) {
183     byte[] out = null;
184     if (selfTest) System.out.println("\nReading from file: " + file);
185     try {
186         FileInputStream infile = new FileInputStream(file);
187         out = new byte[length];
188         int count = infile.available();
189
190         infile.skip(offSet);
191         infile.read(out, 0, length);
192
193         if (selfTest) System.out.println("\nResult (string): " + new String(out));
194         if (selfTest) System.out.println("\nResult (hex): ");
195         if (selfTest) print(out);
196         infile.close();
197
198     } catch (Exception e) {
199         System.err.println("Caught exception " + e.toString());
200     }
201     return out;
202 }
203
204 private void writeSignatureFile(byte[] array) {
205
206     if (selfTest) System.out.println("\nGenerating & writing file: signature ...");
207     try {
208         FileOutputStream outfile = new FileOutputStream(path + "signature");
209         ByteArrayOutputStream output = new ByteArrayOutputStream();
210
211         PrintStream outp = new PrintStream(outfile);
212         outp.println("#####");
213         outp.println("# #");
214         outp.println("# DO NOT MODERATE THIS FILE #");

```

```

215         outp.println("#                               #");
216         outp.println("#####");
217         outp.println("#                               #");
218         outp.println("#             SHA-1             #");
219         outp.println("#                               #");
220         outp.println("#####");
221         outp.println("");
222         //write signature itself
223         output.write(array, 0, array.length);
224         output.writeTo(outfile);
225
226         outp.println("");
227         outp.println("");
228         outp.println("#####");
229         outp.println("END HASH OF FILE");
230
231         outfile.close();
232         output.close();
233         outp.close();
234
235     } catch (Exception e) {
236         System.err.println("Caught exception " + e.toString());
237     }
238 }
239
240 private void writeKeyFile(byte[] array) {
241
242     try {
243         FileOutputStream outfile = new FileOutputStream(path + "key");
244         ByteArrayOutputStream output = new ByteArrayOutputStream();
245
246         PrintStream outp = new PrintStream(outfile);
247         outp.println("#####");
248         outp.println("#                               #");
249         outp.println("# DO NOT MODERATE THIS FILE #");
250         outp.println("#                               #");
251         outp.println("#####");
252         outp.println("#                               #");
253         outp.println("#             RSA Public Key             #");
254         outp.println("#                               #");
255         outp.println("#####");
256         outp.println("");
257
258         output.write(array, 0, array.length);
259         output.writeTo(outfile);
260
261         outp.println("");
262         outp.println("");
263         outp.println("#####");
264         outp.println("END PUBLIC KEY");
265
266         outfile.close();
267         output.close();
268         outp.close();
269
270     } catch (Exception e) {
271         System.err.println("Caught exception " + e.toString());
272     }
273 }
274
275 public void exportPublicKey() {
276     byte[] mod = null;
277     byte[] exp = null;
278     byte[] key = new byte[128];
279     if (selfTest) System.out.println("\nExporting public key");
280     try {
281         mod = client.getPublicModulus();
282         if (selfTest) System.out.println("\nModulus obtained from card: \n");
283         if (selfTest) System.out.println(new String(mod));
284         exp = client.getPublicExponent();
285         if (selfTest) System.out.println("\nExponent obtained from card: \n");
286         if (selfTest) System.out.println(new String(exp));

```

```

287     } catch (CardTerminalException e) {
288         System.err.println("Error while reading from card " + e.toString());
289     }
290     //copy the keys to the return array
291     //making the first byte mark the length of the exponent
292     //modulus is always 512 bit (and exp seems to be 3 bytes always)
293     key[0] = (byte) exp.length;
294     System.arraycopy(mod, 0, key, 1, mod.length);
295     System.arraycopy(exp, 0, key, 1 + mod.length, exp.length);
296     writeKeyFile(key);
297     if (selfTest) System.out.println("\nKey written to file key.");
298 }
299
300 private void print(byte[] input) {
301     int i;
302     for(i=0; i<input.length; i++) {
303         System.out.print(toHex(input[i]));
304         System.out.print(" ");
305     }
306     System.out.println("");
307 }
308
309 public String toHex(byte in) {
310     //converting back to unsigned
311     int number = (int) in & 0xff;
312     return Integer.toHexString(number).toUpperCase();
313 }
314
315
316 private static void displayHelp() {
317     System.out.println("Signature - usage :");
318     System.out.println("Most config options are set in the file config.cfg");
319     System.out.println("Examples:");
320     System.out.println("  Signature -s file1 -f file2 -k file3  Verify file2 with signature file1 using key file3");
321     System.out.println("  Signature -f file1                    Sign file1");
322     System.out.println("  Signature                               Export public key");
323     System.out.println("Either long or short options are allowed.");
324     System.out.println("  -signaturefile  -s    Signature file to use");
325     System.out.println("  -file           -f    File to sign/verify");
326     System.out.println("  -keyfile       -k    Key file");
327     System.out.println("  -path          -p    Set working path");
328     System.out.println("  -selfttest    Perform selftest");
329 }
330
331 private void selfTest() {
332     System.out.println("\nPerforming self test");
333     exportPublicKey();
334     //readModulus(path + keyFile);
335     //readExponent(path + keyFile);
336     System.out.println("\nSigning: " + sourceFile + "...");
337     sign(path + sourceFile);
338     //read the signature, change it and write it back
339     byte[] orig = readSignature(path + "signature");
340     byte[] test = new byte[64];
341     System.arraycopy(orig, 0, test, 0, 64);
342     test[0] = (byte) 0x40;
343     System.out.println("\nChanging signature from (string): \n" + new String(orig));
344     System.out.println("\n\nto(only first byte changed):\n\n" + new String(test));
345     writeSignatureFile(test);
346     System.out.println("\nVerifying signature from: " + signatureFile + " against: " + sourceFile);
347     verify(path + sourceFile, path + signatureFile);
348     System.out.println("\nSigning: " + sourceFile + "...");
349     sign(path + sourceFile);
350     System.out.println("\nVerifying signature from: " + signatureFile + " against: " + sourceFile);
351     verify(path + sourceFile, path + signatureFile);
352 }
353
354 public static void main(String[] args) {
355     Signature s;
356     //parse arguments
357     if (selfTest) System.out.println("\n----- start of Signature application -----");
358 }

```

```

359     for(int i = 0; i < args.length; i++) {
360         if (args[i].equals("-s") || args[i].equals("-signaturefile")) {
361             i++;
362             signatureFile = args[i];
363         }
364         else if (args[i].equals("-k") || args[i].equals("-keyfile")) {
365             i++;
366             keyFile = args[i];
367         }
368         else if (args[i].equals("-f") || args[i].equals("-file")) {
369             i++;
370             sourceFile = args[i];
371         }
372         else if (args[i].equals("-p") || args[i].equals("-path")) {
373             i++;
374             path = args[i];
375         }
376         else if (args[i].equals("-selftest")) {
377             selfTest = true;
378         }
379         else {
380             displayHelp();
381             System.exit(0);
382         }
383     }
384     //initialize Signature and SignatureClient
385     s = new Signature();
386     client = new SignatureClient(path);
387     client.initOCF();
388     client.runAdminCommands();
389
390     //check if sf is set if then verify
391     //if -f is not set export key
392     //else sign and verify
393     if (selfTest) {
394         s.selfTest();
395     }
396     else if (sourceFile == null && signatureFile == null) {
397         s.exportPublicKey();
398         System.out.println("Exporting public key to file: key");
399     }
400     else if (signatureFile == null) {
401         System.out.println("Sign file: " + sourceFile);
402         s.sign(path + sourceFile);
403     }
404     else {
405         System.out.println("Verify signature file: " + signatureFile + " against sourcefile: " + sourceFile);
406         s.verify(path + sourceFile,path + signatureFile);
407     }
408
409     // terminate client application normally
410     client.stopOCF();
411     if (selfTest) System.out.println("\n----- end of Signature application -----");
412     java.lang.System.exit(0) ;
413 }
414 }

```

## C SignatureClient.java

```
1  /**
2  *-----
3  * Project name      : Digital Signature with Java Smart Card
4  *                   - Java Card Open Platform applet -
5  *
6  * Platform         : Java virtual machine
7  * Language         : 1.3.0-C
8  * Devl tool        : Borland (c) JBuilder 4.0
9  *
10 * Original author  : Nikolaj Aggeboe & Sune Kloppenborg Jeppesen
11 * Date            : Thu Dec 20 10:40:16 CET 2001
12 *-----
13 */
14
15 /*
16 * Package name
17 */
18 package signature.client;
19
20 /*
21 * Imported packages
22 */
23 import java.io.*;
24
25 // standard OCF imports
26 import opencard.core.service.*;
27 import opencard.core.terminal.*;
28 import opencard.opt.applet.AppletID;
29
30 // gemplus VISA imports
31 import com.gemplus.opencard.service.op.*;
32 import com.gemplus.opencard.service.op.vop.*;
33 import com.gemplus.opencard.service.op.vop.vop211.*;
34 import com.gemplus.opencard.service.op.vop.vop200.*;
35 import com.gemplus.tools.gemxpresso.*;
36
37 // GSE imports
38 import com.gemplus.javacard.gse.Simulator;
39 import com.gemplus.javacard.gse.util.APDUPrinter;
40 import com.gemplus.tools.gemxpresso.util.GxpSystem;
41
42 // security and crypto
43 import opencard.opt.security.*;
44
45 /**
46 * The client application that manages and uses the 'SignatureClient' applet in the Card or GSE.
47 * <p>
48 * This implementation uses the OCF 1.1.1 framework and the target specific
49 * CardTerminal to communicate with the 'SignatureClient' applet .
50 * <p>
51 * This client loads a package, installs an applet, selects it and sends it some
52 * APDU commands.
53 */
54 class SignatureClient {
55     /**
56     * global declarations
57     */
58     // OP/VOP OCF CardService and its high level API library
59     private CardServiceOPCore serv;
60     private boolean isVOP211Compliant = true;
61     private GemXpressoService libService;
62     // OCF CardTerminal for Card or GSE communication
63     private CardTerminal terminal;
64     // OCF objects for APDU transport
65     private CommandAPDU cmd;
66     private ResponseAPDU resp;
67     // print utility object
68     private APDUPrinter printer = new APDUPrinter();
69     private String homeDir;
70     //targets path (<gemxpressorad>\resources\targets)
```

```

71 private String keyFileDir;
72 //code dir (<gemxpressorad>\examples\SignatureClient)
73 private String codeHomeDir;
74 //card profile dir (<gemxpressorad>\resouces\cardprofile)
75 private String profileDir;
76 //the package path
77 private String appletPackagePath = "client" + File.separator + "SignatureClient";
78 //jar full filename (<gemxpressorad>\SignatureClient\<cardtype>\<appletPackagePath>\javacard\SignatureClient.jar)
79 private String jarFileName;
80 //sap full filename (<gemxpressorad>\examples\SignatureClient\<cardtype>\<appletPackagePath>\SignatureClient.sar)
81 private String sapFileName;
82 private String cardT;
83
84 private String configPath;
85 /**
86  * GSE target name
87  */
88 // the name defined in the CardTerminal section of the "opencard.properties" file
89 private String gseTarget = "Simulator";
90
91 /**
92  * Card target names
93  */
94 // the name defined in the CardTerminal section of the "opencard.properties" file
95 private String cardTarget;
96 /**
97  * Key file
98  */
99 // the key file (GXP211_PK is default)
100 private String cardKeyFile = "GXP211_PK.properties";
101 // the target flag, if true the GSE target is used, card otherwise
102 private static boolean simulation = true;
103 // the load flag, if true package is loaded in card or GSE
104 private boolean loadPackage = true;
105 // the type of GSE or Card (GXP211_PK is default)
106 private String FrameworkType;
107
108 /**
109  * System authentication informations
110  */
111 // the Security Domain AID used for security
112 private byte[] securityDomainAID = null;
113 // the system key File for management
114 private String keyFile ;
115 // the system key set version
116 private int keySetVersion = 13;
117 // the system key set index (0 is default)
118 private int keyIndex = 0;
119 // the security level definition flags
120 private boolean isEnciphered = false;
121 private boolean isMacing = false;
122
123 /**
124  * Application authentication informations
125  */
126 // the application key Files
127 private String appKeyFile ;
128 // the application key set version
129 private int appKeySetVersion = 13;
130 // the application key set index
131 private int appKeyIndex = 0;
132
133 // the package name
134 private String packageName;
135 // the fully qualified Applet name
136 private String appletName;
137 // the OCF object containing the "load file" AID (package AID)
138 private AppletID packaid;
139 // the OCF object containing the "within load file" AID (applet default AID)
140 private AppletID aid;
141
142 private final static byte INS_TEST_PRIV_KEY = (byte)0x10 ;

```

```

143 private final static byte INS_GET_PUBLIC_KEY      = (byte)0x11 ;
144 private final static byte INS_DECRYPT            = (byte)0x12 ;
145 private final static byte INS_ENCRYPT           = (byte)0x13 ;
146
147 /**
148  * SignatureClient default constructor
149  */
150 public SignatureClient(String path) {
151     configPath = path;
152     loadConfig();
153     GxpSystem.getInstance().setRadHome(homeDir);
154     System.setProperty("user.dir", System.getProperty("gemplus.gemxpresso.rad.home.conf"));
155 }
156
157 /**
158  * OCF layer initialisation
159  */
160 public void initOCF() {
161     // start of initialisation
162     try {
163         // start the OCF layer
164         SmartCard.start();
165         // fixe the target
166         String target = simulation ? gseTarget : cardTarget;
167         // print found terminals
168         printAvailableTerminals();
169         // select the target specific CardTerminal
170         terminal = CardTerminalRegistry.getRegistry().cardTerminalForName(target);
171         // check if required terminal name was found
172         if(terminal == null) {
173             throw new Exception("terminal not found : " + target);
174         }
175         // retrieve the CardTerminal type
176         String type = terminal.getType();
177         if((type.compareTo("SOCKETJC21SIMULATOR") == 0) && !simulation) {
178             throw new Exception("terminal " + target + " is a simulator instance");
179         }
180         if(!(type.compareTo("SOCKETJC21SIMULATOR")==0)&& simulation) {
181             throw new Exception("terminal " + target + " is not a simulator instance");
182         }
183         // create a new card request object
184         CardRequest cr = new CardRequest(CardRequest.ANYCARD, terminal, null);
185         if(simulation) {
186             // GE creation and start
187             Simulator gse = new Simulator();
188             // GSE starts
189             gse.start(5000,FrameworkType);
190             // print the GSE build version and date
191             gse.printVersion();
192             System.out.println(gse.getCardType());
193         }
194         // build the complete key file
195         keyFile = appKeyFile = keyFileDir + File.separator + cardKeyFile;
196         // wait for card insertion
197         SmartCard sc = SmartCard.waitForCard(cr);
198         // get the OP/VOP specific CardService
199         if(isVOP211Compliant)
200             serv = (CardServiceOPCore)sc.getCardService(CardServiceVOP211.class, true);
201         else
202             serv = (CardServiceOPCore)sc.getCardService(CardServiceVOP200.class, true);
203         //sets FullCrypto if FullCrypto card or Limited Crypto if IS card
204         serv.setFullCrypto(FrameworkType.indexOf("IS")==-1);
205         // create the high level API library object
206         libService = new GemXpressoService();
207         // set the OP/VOP CardService to the service library for communication
208         libService.setCardService(serv);
209         // reset the Card or GSE to look for the default ATR
210         System.out.println("ATR returned by the " +
211             ((simulation)? "GSE" : "Card") + "... ");
212         displayAPDUResp(new ResponseAPDU(serv.warmReset()));
213         // the service authentication object use for authentication configuration
214         VOPAuthenticationInput authenticationInput;

```

```

215         // service authentication object creation
216         authenticationInput = new VOPAuthenticationInput();
217         // look if key set version is defined
218         if(keySetVersion > 0) {
219             // key set version configuration
220             authenticationInput.setKeySetVersion(keySetVersion);
221             // do not use the key set version defined in the target file
222             authenticationInput.setDefaultKeySetVersion(false);
223         }
224         else {
225             // no key set version defined
226             // use the key set version defined in the target file
227             authenticationInput.setDefaultKeySetVersion(true);
228         }
229         // key set version index configuration
230         authenticationInput.setKeyIndex(keyIndex);
231         // security configuration
232         authenticationInput.setEnciphered(isEnciphered);
233         authenticationInput.setMacing(isMacing);
234         // define the target specific key file to use
235         authenticationInput.setKeyfile(keyFile);
236         // define if the security domain has to be select (yes)
237         authenticationInput.setSecurityDomainSelection(true);
238         // define the Security Domain AID
239         // null force the use of the AID present in the target key file
240         authenticationInput.setSecurityDomainAID(null);
241         try {
242             // process mutual authentication
243             // initialize/update and external/authenticate are done
244             Result result = serv.openSecureChanel(authenticationInput);
245             // check the result object for authentication status
246             if((result!=null) && !result.isOK()) {
247                 throw new Exception("authentication error : " + result.getResultMessage());
248             }
249         }
250         catch(Exception ex) {
251             // authentication fails
252             throw new Exception("authentication error : " + ex.getMessage() );
253         }
254         // authentication succeed
255         System.out.println("\nAuthentication OK");
256         System.out.println("");
257         //initialize PIN with value "1234" and a try remaining of 6
258         System.out.println("Initialize OP global PIN");
259         System.out.println("");
260         PINResult pinResult = serv.changePIN("1234", 6, null);
261         if((pinResult!=null) && !pinResult.isOK()) {
262             throw new Exception("changePIN error : " + pinResult.getResultMessage());
263         }
264         System.out.println("\nPIN initialisation OK");
265         System.out.println("");
266     } catch(Exception ex) {
267         // exception generated during OCF initialization
268         System.err.println("Exception caught in initOCF : " + ex.getMessage() );
269         // terminate client application
270         System.exit( -1 );
271     }
272 }
273
274 /**
275  * Utility that print the available OCF CardTerminal for the client
276  */
277 private void printAvailableTerminals() {
278     // get an enumeration from the registry
279     java.util.Enumeration terminals = CardTerminalRegistry.getRegistry().
280         getCardTerminals();
281     // the CardTerminal we are retrieving information
282     CardTerminal inFocusTerminal;
283     // analyse loop
284     while (terminals.hasMoreElements()) {
285         inFocusTerminal = (CardTerminal)terminals.nextElement();
286         System.out.println("Found OCF Card Terminal:");

```

```

287         // print the user defined name
288         System.out.println("\t- Name = " + inFocusTerminal.getName());
289         // print the legal type
290         System.out.println("\t- Type = " + inFocusTerminal.getType());
291         // print the corresponding adress
292         System.out.println("\t- Adress = " + inFocusTerminal.getAddress() + "\n");
293     }
294 }
295 /**
296  * Method containing administrative commands
297  */
298 public void runAdminCommands() {
299     try {
300         /*
301          * Administration commands preparing the package.
302          * IDs are created for Package and Applet(s).
303          * A downloadable Package formatted for the simulator is created,
304          * containing applet(s) defined by the client.
305          */
306
307         // create a "load file" (package) application ID
308         packaid = new AppletID(new byte[]
309         { (byte)0xA0,(byte)0x00,(byte)0x00,(byte)0x00,
310           (byte)0x18,(byte)0xFF,(byte)0x00,(byte)0x00,
311           (byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,
312           (byte)0x00,(byte)0x00,(byte)0x01,(byte)0x01
313         });
314         /**
315          * create a new "load file" (package)
316          */
317         // initialize the package name
318         packageName = "SignatureClient";
319         // create a CardService specific object for "LoadFile" type description
320         OPLoadFile pack = new OPLoadFile(packageName);
321         // personalize the load file AID
322         pack.setAID(packaid.getBytes());
323         // set Application Path
324         pack.setApplicationPath(codeHomeDir + File.separator + "out");
325         // personalize the Security Domain the applet will use for its security
326         pack.setSecurityDomainAID(null);
327         // personalize the load parameter (OP specific)
328         pack.setLoadParameters(null);
329         // create the "within load file" (applet) application ID
330         aid = new AppletID(new byte[]
331         { (byte)0xA0,(byte)0x00,(byte)0x00,(byte)0x00,
332           (byte)0x18,(byte)0xFF,(byte)0x00,(byte)0x00,
333           (byte)0x00,(byte)0x00,(byte)0x00,(byte)0x00,
334           (byte)0x00,(byte)0x00,(byte)0x01,(byte)0x02
335         });
336         /**
337          * create a new "within load file" (add applet in the package)
338          */
339         // initialize the package name
340         appletName = "SignatureClient";
341         // create a CardService specific object for "withinLoadFile" type description
342         OPApplication applet = new OPApplication(pack,appletName);
343         // personalize the within load file AID (default AID)
344         applet.setAID(aid.getBytes());
345         // personalize the application instance AID (registered applet AID)
346         applet.setInstanceAID(aid.getBytes());
347         // define if the applet has to be installed after the load
348         applet.setIsToInstallFlag(true);
349         // define if the application is a Security Domain. This information
350         // will be present in the application privilege flag
351         applet.setSecurityDomainFlag(false);
352         // define if the application state will be set to "Selectable" after install
353         applet.setSelectableFlag(true);
354         // add the "within load file" (applet) in the "load file" (package)
355         pack.addApplication(applet);
356         // looks if the "load file" has to be loaded
357         if(loadPackage) {
358             // load it

```

```

359         loadPackage(pack);
360     }
361     /* select the applet for use
362     * applet 'select' method is called. If selection is successful, 'Install Applet'
363     * is deselected and commands are sent to the selected applet 'process' method.
364     * For new administration commands Install Applet must be selected again.
365     * Its AID can be found in the 'GSEConst' class. Its value is GSEConst.SYSTEM_APPLET_AID.
366     * See Javacard 2.1 specifications for details.
367     */
368     System.out.println("Select application... ");
369     SelectResult selectResult = serv.select(aid);
370     if(!((selectResult!=null)&&(selectResult.isOK()))
371         throw new Exception("Select application error : " + selectResult.getResultMessage());
372     System.out.println("Select application " + appletName + " OK");
373 }
374 catch(Exception ex) {
375     System.err.println("Exception caught in runAdminCommands : " + ex.getMessage() );
376     // terminate client application
377     java.lang.System.exit( -1 );
378 }
379 }
380
381 public void loadPackage(OPLoadFile pack) throws Exception {
382     System.out.println("Starting load process...");
383     // the result of the load file operation
384     LoadFileResult loadFileResult = null;
385     // define the maximum length of data in each "load" APDU
386     int maxBlockSize = 200;
387     // the Security Domain OCF AID object
388     AppletID securityDomainID = null;
389     if(pack.getSecurityDomainAID()!=null) {
390         // create the Security Domain OCF AID object
391         securityDomainID = new AppletID(pack.getSecurityDomainAID());
392     }
393     if(simulation) {
394         // Loading the GSE
395         // load the "load file" in the GSE
396         System.out.println("Loading the GSE... ");
397         //
398         loadFileResult = libService.loadFromSapFile(
399             packaid,
400             securityDomainID,
401             pack.getLoadParameters(),
402             maxBlockSize,
403             sapFileName
404         );
405     }
406     // end of Loading the GSE
407     else {
408         // Loading card
409         // load the JAR file in the card
410         System.out.println("Loading the card...");
411         // check valid JAR file
412         if(!jarFileName.equals("")) {
413             loadFileResult = libService.loadFromJarFile(
414                 profileDir + File.separator + FrameworkType + ".gsc",
415                 packaid,
416                 securityDomainID,
417                 pack.getLoadParameters(),
418                 maxBlockSize,
419                 jarFileName
420             );
421         }
422     }
423     // end of Loading the card
424     // check the load result state
425     // case of null result
426     if(loadFileResult == null)
427         throw new Exception("Load error : null loadFileResult");
428     // case of wrong result
429     if(!loadFileResult.isOK())
430         throw new Exception("Load error : " + loadFileResult.getResultMessage());

```

```

431         // loading succeed
432         System.out.println("Loading " + packageName + " succeeded");
433         // install the applet with parameters
434         System.out.println("Install applet... ");
435         // the result of the load file operation
436         InstallApplicationResult installApplicationResult;
437         if(isVOP211Compliant) {
438             installApplicationResult = ((CardServiceVOP211)serv).installApplication(
439                 aid,          // application AID
440                 aid,          // application instance AID
441                 packaid,     // load File AID
442                 null,        // application properties byte array
443                 // install parameter (length Min 0, length Max 32)
444                 new byte[] {(byte)0x00,(byte)0x64},
445                 null,        // non volatile code space limit parameters
446                 null,        // volatile data space limit parameters
447                 null,        // non volatile data space limit parameters
448                 null,        // others parameters
449                 null,        // install token (none)
450                 true,        // make selectable flag
451                 null         // DAP (not present)
452             );
453         }
454         else {
455             installApplicationResult = ((CardServiceVOP200)serv).installApplication(
456                 aid,          // application AID
457                 aid,          // application instance AID
458                 packaid,     // load File AID
459                 null,        // application properties byte array
460                 // install parameter (length Min 0, length Max 32)
461                 new byte[] {(byte)0x00,(byte)0x64},
462                 null,        // install token (none)
463                 true,        // make selectable flag
464                 null         // DAP (not present)
465             );
466         }
467         // check install result
468         if(!((installApplicationResult!=null)&&(installApplicationResult.isOK())))
469             throw new Exception("Install error : " + installApplicationResult.getResultMessage());
470         // install succeed
471         System.out.println("\nInstalling application " + appletName + " succeeded");
472     }
473
474     public RSAPublicKey getPublicKey() {
475         RSAPublicKey publicRSAKey;
476         try {
477             byte[] e = getPublicExponent();
478             int eLength = e.length;
479             byte[] m = getPublicModulus();
480             int keyLength = m.length;
481             publicRSAKey = new RSAPublicKey(eLength, e, m, keyLength);
482         } catch (CardTerminalException e) {
483             return null;
484         }
485         return publicRSAKey;
486     }
487
488     public byte[] getPublicModulus() throws CardTerminalException{
489         byte[] fromCard = new byte[64];
490         // build the OCF command object containing the APDU to send
491         cmd = new CommandAPDU(new byte[]
492             {(byte)0x90, INS_GET_PUBLIC_KEY, (byte)0x01, (byte)0x00, (byte)0x43});
493         // send the APDU to the target
494         resp = serv.sendAPDU(cmd);
495         // display the exchange
496         displayAPDU(cmd, resp);
497         fromCard = resp.getBuffer();
498         fromCard = unwrap(fromCard);
499         return fromCard;
500     }
501
502     public byte[] getPublicExponent() throws CardTerminalException{

```

```

503     byte[] fromCard = new byte[64];
504     // build the OCF command object containing the APDU to send
505     cmd = new CommandAPDU(new byte[]
506         {(byte)0x90, INS_GET_PUBLIC_KEY, (byte)0x00, (byte)0x00, (byte) 0x40});
507     // send the APDU to the target
508     resp = serv.sendAPDU(cmd);
509     // display the exchange
510     displayAPDU(cmd, resp);
511     fromCard = resp.getBuffer();
512     fromCard = unwrap(fromCard);
513     return fromCard;
514 }
515
516 public byte[] decrypt(byte[] input) throws CardTerminalException{
517     byte[] toCard = new byte[70];
518     byte[] fromCard = new byte[64];
519     toCard[0] = (byte) 0x90;
520     toCard[1] = (byte) INS_DECRYPT;
521     toCard[4] = (byte) 0x40; //bytelength
522     System.arraycopy(input, 0, toCard, 5, 64);
523     toCard[69] = (byte) 0x40;
524     cmd = new CommandAPDU(toCard);
525     // send the APDU to the target
526     resp = serv.sendAPDU(cmd);
527     // display the exchange
528     displayAPDU(cmd, resp);
529     //get the buffer and prepare it for return
530     fromCard = unwrap(resp.getBuffer());
531     return fromCard;
532 }
533
534 public byte[] encrypt(byte[] input) throws CardTerminalException{
535     byte[] toCard = new byte[70];
536     byte[] fromCard = new byte[64];
537     if (input.length != 64) {
538         System.out.println("Wrong array size given " + input.length);
539         System.exit(-1);
540     }
541     //build the commandAPDU
542     toCard[0] = (byte) 0x90;
543     toCard[1] = INS_ENCRYPT;
544     toCard[4] = (byte) 0x40; //bytelength
545     System.arraycopy(input, 0, toCard, 5, 64);
546     toCard[69] = (byte) 0x40;
547     cmd = new CommandAPDU(toCard);
548     // send the APDU to the target
549     resp = serv.sendAPDU(cmd);
550     // display the exchange
551     displayAPDU(cmd, resp);
552     //get the buffer and prepare it for return
553     fromCard = unwrap(resp.getBuffer());
554     return fromCard;
555 }
556
557 private byte[] unwrap(byte[] input) {
558     int length = input.length-2;
559     byte[] output = new byte[length];
560     System.arraycopy(input, 0, output, 0, length);
561     return output;
562 }
563
564 /**
565  * Free OCF layer used by the client
566  */
567
568 public void stopOCF() {
569     try {
570         // free the OCF layer before terminating
571         SmartCard.shutdown();
572     } catch(Exception ex) {
573         System.err.println("Exception caught in stopOCF : " + ex.getMessage());
574         // terminate client application with an exception

```

```

575         java.lang.System.exit( -1 );
576     }
577 }
578
579 /**
580  * Display utility for APDU command
581  * @param termCmd a CommandAPDU type command
582  * @param cardResp a ResponseAPDU type response
583  */
584 private void displayAPDUCmd(CommandAPDU termCmd) {
585     printer.printAPDUCmd(termCmd.getBuffer());
586 }
587 /**
588  * Display utility for APDU response
589  * @param termCmd a CommandAPDU type command
590  * @param cardResp a ResponseAPDU type response
591  */
592 private void displayAPDUResp(ResponseAPDU cardResp) {
593     printer.printAPDUResp(cardResp.getBuffer());
594 }
595 /**
596  * Display utility for APDU command & response
597  * @param termCmd a CommandAPDU type command
598  * @param cardResp a ResponseAPDU type response
599  */
600 private void displayAPDU(CommandAPDU termCmd, ResponseAPDU cardResp) {
601     printer.printAPDU(termCmd.getBuffer(), cardResp.getBuffer());
602 }
603
604 //load and parse the config file
605 private void loadConfig() {
606     BufferedReader configFile;
607     String input;
608     String value;
609     //initialize needed variables
610     configFile = null;
611
612     try {
613         configFile = new BufferedReader(new FileReader(new File(configPath + File.separator + "config.cfg")));
614     } catch (FileNotFoundException e) {
615         System.err.println("Config file: " + configPath + File.separator + "config.cfg not found. " + e.toString());
616         System.exit(-1);
617     }
618     try {
619         while(configFile.ready()) {
620             input = configFile.readLine();
621             if (input.startsWith("#")) {}
622             else if (input.startsWith("homeDir:")) {
623                 value = input.substring(input.indexOf(":") + 2, input.length());
624                 homeDir = value;
625             }
626             else if (input.startsWith("keyFileDir:")) {
627                 value = input.substring(input.indexOf(":") + 2, input.length());
628                 keyFileDir = value;
629             }
630             else if (input.startsWith("codeHomeDir:")) {
631                 value = input.substring(input.indexOf(":") + 2, input.length());
632                 codeHomeDir = value;
633             }
634             else if (input.startsWith("client.profileDir:")) {
635                 value = input.substring(input.indexOf(":") + 2, input.length());
636                 profileDir = value;
637             }
638             else if (input.startsWith("client.sapFileName:")) {
639                 value = input.substring(input.indexOf(":") + 2, input.length());
640                 sapFileName = value;
641             }
642             else if (input.startsWith("client.jarFileName:")) {
643                 value = input.substring(input.indexOf(":") + 2, input.length());
644                 jarFileName = value;
645             }
646             else if (input.startsWith("port:")) {

```

```

647         //not used
648         value = input.substring(input.indexOf(":") + 2, input.length());
649     }
650     else if (input.startsWith("cardtype:")) {
651         value = input.substring(input.indexOf(":") + 2, input.length());
652         cardT = value;
653         cardKeyFile = cardT + ".properties";
654         FrameworkType = cardT;
655         if( cardT.equals("GXP211v1") || cardT.equals("GXP211v1_IS"))
656             isVOP211Compliant = false;
657         else
658             isVOP211Compliant = true;
659     }
660     else if (input.startsWith("cardTarget:")) {
661         value = input.substring(input.indexOf(":") + 2, input.length());
662         cardTarget = value;
663     }
664     else if (input.startsWith("simulation:")) {
665         value = input.substring(input.indexOf(":") + 2, input.length());
666         if (value.equals("true"))
667             simulation = true;
668         else
669             simulation = false;
670     }
671 }
672 } catch (IOException e) {
673     System.err.println("ERROR reading config file. "+ e.toString());
674     System.exit(-1);
675 }
676 }
677 }

```

## D SignatureApplet.java

```
1  /**
2  *-----
3  * Project name      : Digital Signature with Java Smart Card
4  *                   - Java Card Open Platform applet -
5  *
6  * Platform         : Java virtual machine
7  * Language         : 1.3.0-C
8  * Devl tool        : Borland (c) JBuilder 4.0
9  *
10 * Original author  : Nikolaj Aggeboe & Sune Kloppenborg Jeppesen
11 * Date            : Thu Dec 20 10:40:16 CET 2001
12 *-----
13 */
14
15 /*
16 * Package name
17 */
18 package signature;
19
20 /*
21 * Imported packages
22 */
23 // specific import for Javacard API access
24 import javacard.framework.*;
25 // specific import for OpenPlatform API access
26 import visa.openplatform.*;
27
28 // security and crypto
29 import javacard.security.*;
30 import javacardx.crypto.*;
31
32 public class SignatureApplet extends javacard.framework.Applet {
33     // the OP/VOP specific instruction set for mutual authentication if used
34     private final static byte CLA_INITIALIZE_UPDATE      = (byte)0x80 ;
35     private final static byte INS_INITIALIZE_UPDATE     = (byte)0x50 ;
36     private final static byte CLA_EXTERNAL_AUTHENTICATE = (byte)0x84 ;
37     private final static byte INS_EXTERNAL_AUTHENTICATE = (byte)0x82 ;
38
39     // the signature specific instructions
40     private final static byte INS_TEST_PRIV_KEY         = (byte)0x10 ;
41     private final static byte INS_GET_PUBLIC_KEY        = (byte)0x11 ;
42     private final static byte INS_DECRYPT                = (byte)0x12 ;
43     private final static byte INS_ENCRYPT               = (byte)0x13 ;
44
45     // RSA algo instance initialisation
46     private Cipher dRSA_NOPAD, eRSA_NOPAD;
47     private Key privateRSAKey, publicRSAKey;
48     private KeyPair keyPair;
49
50     short inDataLength;
51     short outDataLength;
52
53     boolean keyInitialized;
54     short CipherKeyLength;
55
56     byte[] wBuffer = new byte[64];
57
58     // security domain instance used for secured actions
59     private ProviderSecurityDomain securityObject;
60     // security channel number. The value is returned by the Security Manager "openSecureChannel" method
61     private byte secureChannel = (byte)0xFF;
62     // the channel status
63     private boolean channelOpened = false;
64     // the authentication status
65     private boolean authenticationDone = false;
66     // authentication Enciphered of Maced ?
67     private boolean enciphered = false;
68
69     /**
70     * SignatureApplet default constructor
```

```

71     * Only this class's install method should create the applet object.
72     */
73     protected SignatureApplet(byte[] buffer, short offset, byte length) {
74         // data offset is used for application specific parameter.
75         // initialization with default offset (AID offset).
76         short dataOffset = offset;
77         boolean isOP2 = false;
78
79         if(length > 9) {
80             // Install parameter detail. Compliant with OP 2.0.1.
81
82             // | size | content
83             // |-----|-----
84             // | 1 | [AID_Length]
85             // | 5-16 | [AID_Bytes]
86             // | 1 | [Privilege_Length]
87             // | 1-n | [Privilege_Bytes] (normally 1Byte)
88             // | 1 | [Application_Proprietary_Length]
89             // | 0-m | [Application_Proprietary_Bytes]
90
91             // shift to privilege offset
92             dataOffset += (short)( 1 + buffer[offset]);
93             // finally shift to Application specific offset
94             dataOffset += (short)( 1 + buffer[dataOffset]);
95
96             // <IF NECESSARY, USE COMMENTS TO CHECK LENGTH >
97             // // checks wrong data length
98             // if(buffer[dataOffset] != <PUT YOUR PARAMETERS LENGTH> )
99             // // return received proprietary data length in the reason
100            //     ISOException.throwIt((short)(ISO7816.SW_WRONG_LENGTH + offset + length - dataOffset));
101
102            // go to proprietary data
103            dataOffset++;
104
105            // update flag
106            isOP2 = true;
107
108        } else {
109            // Install parameter compliant with OP 2.0.
110
111            // <IF NECESSARY, USE COMMENTS TO CHECK LENGTH >
112            // if(length != <PUT YOUR PARAMETERS LENGTH> )
113            //     ISOException.throwIt((short)(ISO7816.SW_WRONG_LENGTH + length));
114
115        }
116        //set cipher key length to default 512 bit (actually only 511 bit)
117        CipherKeyLength = 512;
118
119        // KeyPair creation
120        keyPair = new KeyPair(KeyPair.ALG_RSA, (short)512);
121
122        // starts key generation process
123        keyPair.genKeyPair();
124        publicRSAKey = keyPair.getPublic();
125        privateRSAKey = keyPair.getPrivate();
126
127        //Initialize RSA algorithm with generated keys
128        dRSA_NOPAD = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
129        dRSA_NOPAD.init(privateRSAKey, Cipher.MODE_DECRYPT);
130        eRSA_NOPAD = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
131        eRSA_NOPAD.init(publicRSAKey, Cipher.MODE_ENCRYPT);
132        keyInitialized = true;
133
134        // register this instance
135        if (isOP2)
136            register(buffer, (short)(offset + 1), (byte)buffer[offset]);
137        else
138            register();
139        // ask the system for the Security Object associated to the Applet
140        securityObject = OPSSystem.getSecurityDomain();
141    }
142

```

```

143  /**
144  * Method installing the applet.
145  * @param bArray the array constaining installation parameters
146  * @param bOffset the starting offset in bArray
147  * @param bLength the length in bytes of the data parameter in bArray
148  */
149  public static void install(byte[] bArray, short bOffset, byte bLength) throws IOException    {
150      /* applet instance creation */
151      new SignatureApplet (bArray, bOffset, (byte)bLength );
152  }
153
154  /**
155  * Select method returns true if applet selection is supported.
156  * @return boolean status of selection.
157  */
158  public boolean select() {
159      // reset security if used.
160      // In case of reset deselect is not called
161      reset_security();
162      // <PUT YOUR SELECTION ACTION HERE>
163      // return status of selection
164      return true;
165  }
166
167  /**
168  * Deselect method called by the system in the deselection process.
169  */
170  public void deselect()    {
171      // reset security if used.
172      reset_security();
173      // <PUT YOUR DESELECTION ACTION HERE>
174      return;
175  }
176
177  /**
178  * Method processing an incoming APDU.
179  * @see APDU
180  * @param apdu the incoming APDU
181  * @exception IOException with the response bytes defined by ISO 7816-4
182  */
183  public void process(APDU apdu) throws IOException    {
184      // get the APDU buffer
185      byte[] apduBuffer = apdu.getBuffer();
186
187      //determine data field length
188      inDataLength = (short) (apduBuffer[ISO7816.OFFSET_LC] & 0xFF);
189
190      // ignore the applet select command dispatched to the process
191      if (selectingApplet())
192          return;
193
194      if (authenticationDone &&
195          ((apduBuffer[ISO7816.OFFSET_CLA] & 0x0F) == 4))
196          enciphered = true;
197      else
198          enciphered = false;
199
200      // APDU instruction parser
201      switch ( apduBuffer[ISO7816.OFFSET_INS] ) {
202          case INS_INITIALIZE_UPDATE :
203              if(apduBuffer[ISO7816.OFFSET_CLA] == CLA_INITIALIZE_UPDATE)
204                  init_update( apdu ) ;
205              else IOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
206              break ;
207
208          case INS_EXTERNAL_AUTHENTICATE :
209              if(apduBuffer[ISO7816.OFFSET_CLA] == CLA_EXTERNAL_AUTHENTICATE)
210                  external_authenticate( apdu ) ;
211              else IOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
212              break ;
213
214          case INS_TEST_PRIV_KEY :

```

```

215         apduBuffer = apdu.getBuffer();
216         apdu.setIncomingAndReceive();
217         if(privateRSAKey.isInitialized()) {
218             apduBuffer[0] = (byte) 0x01;
219         }
220         else {
221             apduBuffer[0] = (byte) 0x00;
222         }
223         if(publicRSAKey.isInitialized()) {
224             apduBuffer[1] = (byte) 0x01;
225         }
226         else {
227             apduBuffer[1] = (byte) 0x00;
228         }
229         // sends the APDU response
230         // switches to output mode
231         apdu.setOutgoing() ;
232         // 2 bytes to return
233         apdu.setOutgoingLength((short)2) ;
234         // offset and length of bytes to return in the APDU buffer
235         apdu.sendBytes((short)0, (short)2) ;
236     break;
237     case INS_GET_PUBLIC_KEY :
238         getPublicKey(apdu);
239     break;
240     case INS_ENCRYPT :
241         //decrypt(apdu);
242         apduBuffer = apdu.getBuffer();
243         inDataLength = apdu.setIncomingAndReceive();
244
245         //perform encryption
246         eRSA_NOPAD.doFinal(apduBuffer, (short)ISO7816.OFFSET_CDATA, (short)(512/8), apduBuffer, (short)0);
247
248         outDataLength = (short) 64;
249         // switches to output mode
250         apdu.setOutgoing() ;
251         // dataLength bytes to return
252         apdu.setOutgoingLength(outDataLength) ;
253         // offset and length of bytes to return in the APDU buffer
254         apdu.sendBytes((short)0, outDataLength) ;
255
256     break;
257     case INS_DECRYPT:
258         apduBuffer = apdu.getBuffer();
259         inDataLength = apdu.setIncomingAndReceive();
260
261         //perform decryption
262         dRSA_NOPAD.doFinal(apduBuffer, (short)ISO7816.OFFSET_CDATA, (short)(512/8), apduBuffer, (short)0);
263
264         // sends the APDU response
265         outDataLength = (short) 64;
266         // switches to output mode
267         apdu.setOutgoing() ;
268         // dataLength bytes to return
269         apdu.setOutgoingLength(outDataLength) ;
270         // offset and length of bytes to return in the APDU buffer
271         apdu.sendBytes((short)0, outDataLength) ;
272     break;
273     default :
274         // The INS code is not supported by the dispatcher
275         ISOException.throwIt( ISO7816.SW_INS_NOT_SUPPORTED ) ;
276     break ;
277 }
278 }
279 }
280 public void test(APDU apdu) {
281     // get the APDU buffer
282     byte[] apduBuffer = apdu.getBuffer();
283
284     apdu.setIncomingAndReceive();
285
286     // Decrypt and then encrypt the provided array

```

```

287         dRSA_NOPAD.doFinal(apduBuffer, (short)ISO7816.OFFSET_CDATA, (short)(512/8), wBuffer, (short)0);
288         eRSA_NOPAD.doFinal(wBuffer, (short)0, (short)(512/8), apduBuffer, (short)0);
289
290         // sends the APDU response
291         // switches to output mode
292         apdu.setOutgoing() ;
293         // 64 bytes to return
294         apdu.setOutgoingLength((short)64) ;
295         // offset and length of bytes to return in the APDU buffer
296         apdu.sendBytes((short)0, (short)64) ;
297     }
298     public void getPublicKey(APDU apdu) {
299         // get the APDU buffer
300         byte[] apduBuffer = apdu.getBuffer();
301         short action = apduBuffer[2];
302         apdu.setIncomingAndReceive();
303
304         if (action == 0x00) {
305             outDataLength = ((RSAPublicKey) publicRSAKey).getExponent(apduBuffer, (short) 0);
306         }
307         else {
308             outDataLength = ((RSAPublicKey) publicRSAKey).getModulus(apduBuffer, (short) 0);
309         }
310         // sends the APDU response
311         // switches to output mode
312         apdu.setOutgoing() ;
313         // dataLength bytes to return
314         apdu.setOutgoingLength(outDataLength) ;
315         // offset and length of bytes to return in the APDU buffer
316         apdu.sendBytes((short)0, outDataLength) ;
317     }
318     //security
319     /**
320      * Performs the "init_update" security operation.
321      *
322      * @param apdu The APDU to process.
323      */
324     private void init_update( APDU apdu ) {
325         // receives data
326         apdu.setIncomingAndReceive();
327         // checks for existing active secure channel
328         if(channelOpened) {
329             // close the opened security channel
330             reset_security();
331         }
332         try {
333             // open a new security channel
334             secureChannel = securityObject.openSecureChannel(apdu);
335             // set the channel flag to open
336             channelOpened = true;
337
338             // send authentication result
339             short expected = (short) 0x1C;
340             apdu.setOutgoingLength(expected);
341             apdu.sendBytes(ISO7816.OFFSET_CDATA, expected);
342         }
343         catch(CardRuntimeException cre) {
344             // no available channel or APDU is invalid
345             ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
346         }
347         // send authentication result
348         ISOException.throwIt(ISO7816.SW_NO_ERROR);
349     }
350
351     /**
352      * Performs the "external_authenticate" security operation.
353      *
354      * @param apdu The APDU to process.
355      */
356     private void external_authenticate( APDU apdu ) {
357         // receives data
358         apdu.setIncomingAndReceive();

```

```

359
360 // checks for existing active secure channel
361 if(channelOpened) {
362     try {
363         // try to authenticate the client
364         securityObject.verifyExternalAuthenticate(secureChannel, apdu);
365         // authentication succeed
366         authenticationDone = true;
367     } catch(CardRuntime cre) {
368         // authentication fails
369         // set authentication flag to fails
370         authenticationDone = false;
371         // close the opened security channel
372         reset_security();
373         // send authentication result
374         ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
375     }
376     // send authentication result
377     ISOException.throwIt(ISO7816.SW_NO_ERROR);
378 }
379 else
380     ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
381 }
382
383 /**
384  * The "reset_security" method close an opened secure channel if exist.
385  * @return void.
386  */
387 public void reset_security() {
388     // close the secure channel if opened.
389     if(secureChannel != (byte)0xFF) {
390         try {
391             // close the opened security channel
392             securityObject.closeSecureChannel(secureChannel);
393         } catch(CardRuntime cre2) {
394             // channel number is invalid. this case is ignored
395         }
396         secureChannel = (byte)0xFF;
397         channelOpened = false;
398         authenticationDone = false;
399     }
400     return;
401 }
402 }

```