Kryptografie verständlich

Ein Fachbuch für
Studierende und Anwender

von
Christof Paar und Jan Pelzl

www.crypto-textbook.com

Kapitel 7

Das RSA-Kryptosystem

(Version: 1. Dezember 2016)

Übersicht

- Das RSA-Verfahren
- Praktische Aspekte
- Erzeugung großer Primzahlen
- Angriffe und Gegenmaßnahmen
- Lessons Learned

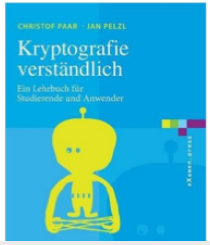
Übersicht

- **Das RSA-Verfahren**
- Praktische Aspekte
- Erzeugung großer Primzahlen
- Angriffe und Gegenmaßnahmen
- Lessons Learned

Das RSA-Verfahren
Geschichte

- Martin Hellman und Whitfield Diffie veröffentlichen 1976 ihr wegweisendes Papier zu neuartigen asymmetrischen Verfahren
- Ronald Rivest, Adi Shamir und Leonard Adleman stellen das asymmetrische RSA-Verfahren in1977 vor
- Bis heute ist RSA das am meisten verwendete asymmetrische Verfahren (auch wenn elliptische Kurve Kryptografie (ECC) populärer wird)
- Zwei wesentliche Anwendungen für RSA:
  - Transport von (symmetrischen) Schlüsseln
  - Digitale Signaturen

Das RSA-Verfahren
Geschichte



Das RSA-Verfahren
Ver- und Entschlüsselung

- RSA Operationen werden module n, d.h. im Ring Z_n durchgeführt, wobei $n = p * q$, mit den großen Primzahlen p, q
- Ver- und Entschlüsselung sind einfache Exponentiationen im Ring

Gegen: Öffentlicher Schlüssel $(n,e) = k_{pub}$ und privater Schlüssel $d = k_{pr}$

Verschlüsselung: $y = e_{k_{pub}}(x) \equiv x^e \bmod n$

Entschlüsselung: $x = d_{k_{pr}}(y) \equiv y^d \bmod n$

mit $x, y \in Z_n$.

- In der Praxis sind x, y, n und d sehr große ganze Zahlen (≥ 1024 Bit)
- Die Sicherheit des Verfahrens basiert auf der Schwierigkeit, den „privaten Exponenten“ d aus dem öffentlichen Werten (n, e) zu berechnen

Das RSA-Verfahren
Schlüsselerzeugung

- RSA hat eine Set-up Phase, in welcher die privaten und öffentlichen Schlüssel berechnet werden

Algorithmus: RSA Schlüsselerzeugung

Ausgabe: Öffentlicher Schlüssel $k_{pub} = (n, e)$ und privater Schlüssel $k_{pr} = d$

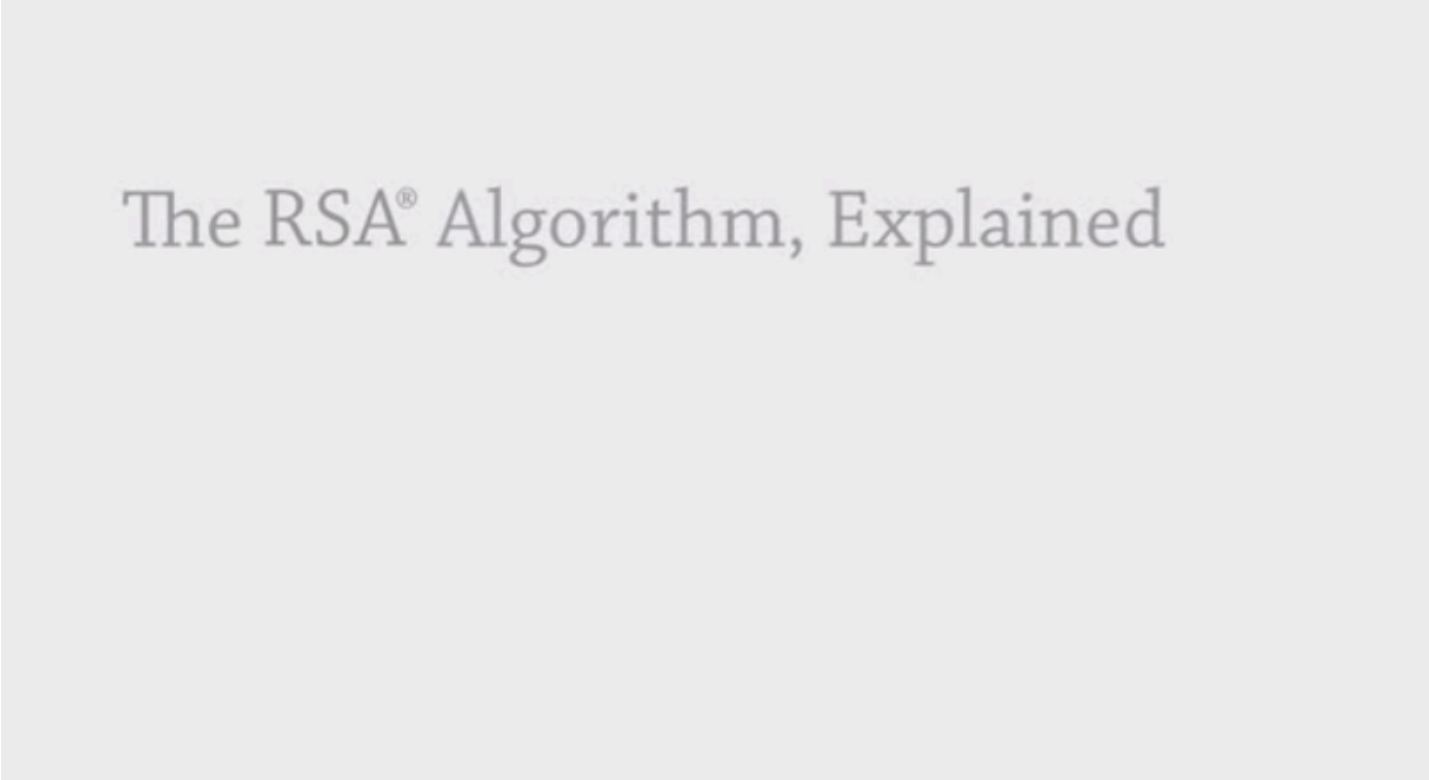
1. Wähle zwei große Primzahlen p, q
2. Berechne $n = p * q$
3. Berechne $\Phi(n) = (p-1) * (q-1)$
4. Wähle den öffentlichen Exponenten $e \; \varepsilon \; \{1, 2, \ldots, \Phi(n)-1\}$ so, dass ggT$(e, \Phi(n)) = 1$
5. Berechne den privaten Schlüssel d so, dass $d * e \equiv 1 \bmod \Phi(n)$
6. **RETURN** $k_{pub} = (n, e)$, $k_{pr} = d$

Anmerkung:

- Die Wahl von zwei großen Primzahl p, q (in Schritt 1) ist nicht trivial
- ggT$(e, \Phi(n)) = 1$ sichert, dass e eine Inverse hat und damit ein privater Schlüssel d existiert

Das RSA-Verfahren

Video: RSA explained by its inventors



Das RSA-Verfahren

Beispiel mit kleinen Zahlen

ALICE	BOB
Nachricht ***x = 4***	1. Wähle $p = 3$ and $q = 11$
	2. Berechne $n = p * q = 33$
	3. $\Phi(n) = (3-1) * (11-1) = 20$
	4. Wähle $e = 3$
	5. $d \equiv e^{-1} \equiv 7 \bmod 20$
	$K_{pub} = (33,3)$ ←
$y = x^e \equiv 4^3 \equiv 31 \bmod 33$	
y = 31 →	
	$y^d = 31^7 \equiv \mathbf{4 = x} \bmod 33$

Das RSA-Verfahren
Die RSA-Challenge

Faktorisierung:

Gegeben n, b, $y = x^b \bmod n$, **finde Faktoren** $p \bullet q = n$.

Berechne dann:

- $\Phi(n) = (p-1)(q-1)$
- $b = a^{-1} \bmod \Phi(n)$
- $x = y^a \bmod n$

Das RSA-Verfahren
Die RSA-Challenge

100.000$-Beispiel für den Modul *n* (1024 Bit):

n=135066410865995223349603216278805969938881475
605667027524485143851526510604859533833940287
150571909441798207282164471551373680419703964
191743046496589274256239341020864383202110372
958725762358509643110564073501508187510676594
629205563685529475213500852879416377328533906
109750544334999811150056977236890927563

Quelle: EMC, The RSA Challenge

Das RSA-Verfahren

Historie der RSA-Faktorisierungen

Challenge	Datum	MIPS-Jahre	Algorithmus
RSA-100	April 1991	7	Quadratisches Sieb
RSA-110	April 1992	75	Quadratisches Sieb
RSA-120	Juni 1993	830	Quadratisches Sieb
RSA-129	April 1994	5000	Quadratisches Sieb
RSA-130	April 1996	500	Allgemeines Zahlkörpersieb
RSA-140	Februar 1999	1500	Allgemeines Zahlkörpersieb
RSA-155	August 1999	8000	Allgemeines Zahlkörpersieb
RSA-576bit	Dezember 2003	n.b.	Allgemeines Zahlkörpersieb
RSA-640bit	November 2005	n.b.	Allgemeines Zahlkörpersieb
RSA-768bit	Dezember 2009	n.b.	Allgemeines Zahlkörpersieb

Übersicht

- Das RSA-Verfahren
- **Praktische Aspekte**
- Erzeugung großer Primzahlen
- Angriffe und Gegenmaßnahmen
- Lessons Learned

Praktische Aspekte
Arithmetik

- Wesentliche Operation bei RSA: modulare Exponentiation
- Durch Verwendung von sehr großen Zahlen ist RSA um Größenordnungen langsamer als symmetrische Verfahren wie z.B. DES oder AES
- Bei der Implementierung von RSA (insbesondere auf eingeschränkten Plattformen) müssen Algorithmen sorgfältig ausgewählt werden
- Der **Square-and-Multiply Algorithmus** ermöglicht schnelle Exponentiation auch mit sehr großen Zahlen

Praktische Aspekte
Square-and-Multiply

- **Prinzip:** Verarbeite Bits des Exponenten von MSB zum LSB und quadriere/ multipliziere Operanden entsprechend

Algorithm: Square-and-Multiply für $x^H \bmod n$

Eingabe: Exponent H, Basis x, Modul n

Ausgabe: $y = x^H \bmod n$

1. Bestimme binäre Darstellung $H = (h_t, h_{t-1}, ..., h_0)_2$
2. **FOR** $i = t\text{-}1$ **TO** 0
3. $y = y^2 \bmod n$
4. **IF** $h_i = 1$ **THEN**
5. $y = y * x \bmod n$
6. **RETURN** y

- Regel: Quadriere in jedem Schritt (Schritt 3) und multipliziere Zwischenergebnis mit x wenn Bit $h_i = 1$ (Schritt 5)
- Modulare Reduktion nach jedem Schritt, um Zwischenergebnis klein zu halten

Praktische Aspekte
Square-and-Multiply: Beispiel

- Ziel: Berechne x^{26} ohne modulare Exponentiation
- Binäre Darstellung des Exponenten $26 = (1,1,0,1,0)_2 = (h_4,h_3,h_2,h_1,h_0)_2$

Schritt		Exponent (binär)	Op	Kommentar
1	$x = x^1$	$(1)_2$		Initialisierung, h_4 bearbeitet
1a	$(x^1)^2 = x^2$	$(10)_2$	Q	Bearbeitung von h_3
1b	$x^2 * x = x^3$	$(11)_2$	M	$h_3 = 1$
2a	$(x^3)^2 = x^6$	$(110)_2$	Q	Bearbeitung von h_2
2b	-	$(110)_2$	-	$h_0 = 0$
3a	$(x^6)^2 = x^{12}$	$(1100)_2$	Q	Bearbeitung von h_1
3b	$x^{12} * x = x^{13}$	$(1101)_2$	M	$h_1=1$
4a	$(x^{13})^2 = x^{26}$	$(11010)_2$	Q	Bearbeitung von h_0
4b	-	$(11010)_2$	-	$h_0 = 0$

- Anmerkung: Man kann die Entwicklung des Exponenten $x^{26} = x^{11010}$ *beobachten*

Praktische Aspekte
Square-and-Multiply: Komplexität

- Komplexität des Square-and-Multiply Algorithmus ist logarithmisch, d.h. die Laufzeit ist proportional zu der Bitlänge (anstelle des absoluten Wertes des Exponenten)
  Annahme: Exponent mit einer Länge von $t+1$ Bit
  $H = (h_t, h_{t-1}, \ldots, h_0)_2$
  mit $h_t = 1$. Wir benötigen die folgende Anzahl an Operationen:
  - *# Quadrierungen* = t
  - Durchschnittliche # Multiplikationen = $0{,}5\ t$
  - Gesamt: $\#Q + \#M$ = $1{,}5\ t$
- Da Exponenten häufig zufällig gewählt werden, ist $1.5\ t$ eine sinnvolle Annahme für die durchschnittliche Anzahl an Operationen
- Anmerkung: Je Multiplikation und Quadrierung wird mit sehr großen Zahlen (z.B. 20148 Bit) durchgeführt

Praktische Aspekte
Techniken zur Beschleunigung

- Modulare Exponentiation ist rechenintensiv
- Selbst mit Hilfe des Square-and-Multiply Algorithmus kann RSA auf eingeschränkten Umgebungen wie z.B. Chipkarten langsam sein
- Häufig verwendete Tricks zur Beschleunigung:
  - Kurzer öffentlicher Exponent *e*
  - Chinesischer Restsatz (CRT)
  - Exponentiation mit Vorausberechnung

Praktische Aspekte
Schnelle Verschlüsselung mit kleinem Exponenten

- Die Wahl eines kurzen öffentlichen Exponenten *e* schwächt nicht die Sicherheit von RSA
- Ein kurzer öffentlicher Exponent beschleunigt die RSA Verschlüsselung deutlich

Öffentlicher Schlüssel	***e* in binärer Darstellung**	**#M + #S**
$2^1+1 = 3$	$(11)_2$	1 + 1 = 2
$2^4+1 = 17$	$(1\ 0001)_2$	4 + 1 = 5
$2^{16} + 1$	$(1\ 0000\ 0000\ 0000\ 0001)_2$	16 + 1 = 17

- Dies ist ein weit verbreiteter Trick (z.B. in SSL/TLS) und macht RSA zu dem schnellsten asymmetrischen Verfahren bezüglich der Verschlüsselung

Praktische Aspekte
Schnelle Entschlüsselung mit CRT

- Wahl eines kurzen privaten Schlüssels *d* wäre eine Schwachstelle!
  - *d* muss mindestens *0,3t* Bit haben, wobei *t* die Bitlänge des Moduls *n* ist
- Der Chinesische Restsatz (CRT) kann zur Beschleunigung der Exponentiation mit dem privaten Schlüssel *d* verwendet werden
- Durch Anwendung des CRT kann die Berechnung

$$x^{d \bmod \Phi(n)} \bmod n$$

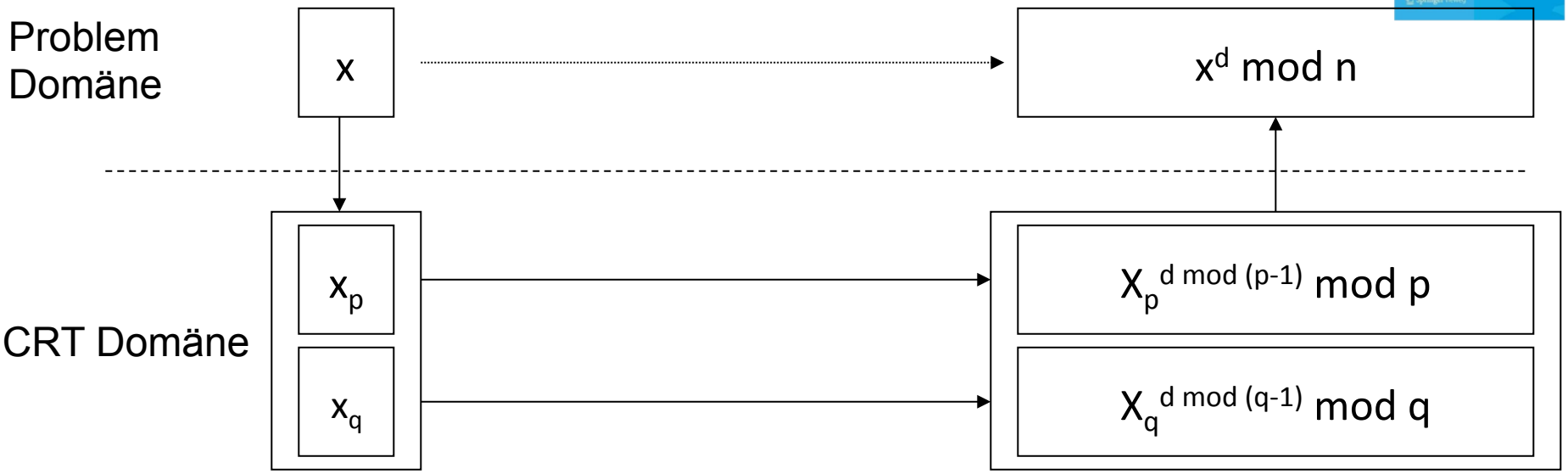
durch folgende zwei Berechnungen ersetzt werden:

$$x^{d \bmod (p-1)} \bmod p \quad \text{und} \quad x^{d \bmod (q-1)} \bmod q$$

wobei *q* und *p* vergleichsweise „klein“ in Bezug auf *n* sind

Praktische Aspekte

CRT: Prinzip



- CRT besteht aus drei Schritten

  (1) Transformation der Operanden in die CRT Domäne

  (2) Modulare Exponentiation in der CRT Domäne

  (3) Rücktransformation in die Problem Domäne

- Das Ergebnis der drei Schritte ist äquivalent zu der direkten modulare Exponentiation in der Problem Domäne

Praktische Aspekte

CRT: Schritt 1 – Transformation

- Transformation in die CRT Domäne erfordert die Kenntnis von p und q
- Nur der Besitzer des privaten Schlüssels kennt p und q, daher kann das CRT nicht zu Beschleunigung der Verschlüsselung eingesetzt werden
- Die Transformation berechnet (x_p, x_q) als Repräsentation von x in der CRT Domäne:

$$x_p \equiv x \bmod p \qquad \text{und} \qquad x_q \equiv x \bmod q$$

Praktische Aspekte

CRT: Schritt 2 – Exponentiation

- Gegeben d_p und d_q.

  Mit $d_p \equiv d \bmod (p-1)$ und $d_q \equiv d \bmod (q-1)$

  benötigt eine Exponentiation der der Problem Domäne zwei Exponentiationen in der CRT Domäne:

  $$y_p \equiv x_p^{d_p} \bmod p \quad \text{und} \quad y_q \equiv x_q^{d_q} \bmod q$$

- In der Praxis wählt man p und q in der Größe der halben Bitlänge von n, d.h., $|p| \approx |q| \approx |n|/2$

Praktische Aspekte

CRT: Schritt 3 – Inverse Transformation

- Die inverse Transformation benötigt zwei modulare Inversionen, welche aufwendig sind:

$$c_p \equiv q^{-1} \bmod p \qquad \text{und} \qquad c_q \equiv p^{-1} \bmod q$$

- Mit der inversen Transformation werden y_p, y_q zum Ergebnis $y \bmod n$ in der Problem Domäne zusammengesetzt

$$y \equiv [\, q * c_p \,] * y_p + [\, p * c_q \,] * y_q \bmod n$$

- Da sich die Primzahlen p und q nicht häufig ändern, kann der Aufwand für die Berechnung vernachlässigt werden und die Werte von

  $[\, q * c_p \,]$ und $[\, p * c_q \,]$

  können vorausberechnet werden

Praktische Aspekte
CRT: Komplexität

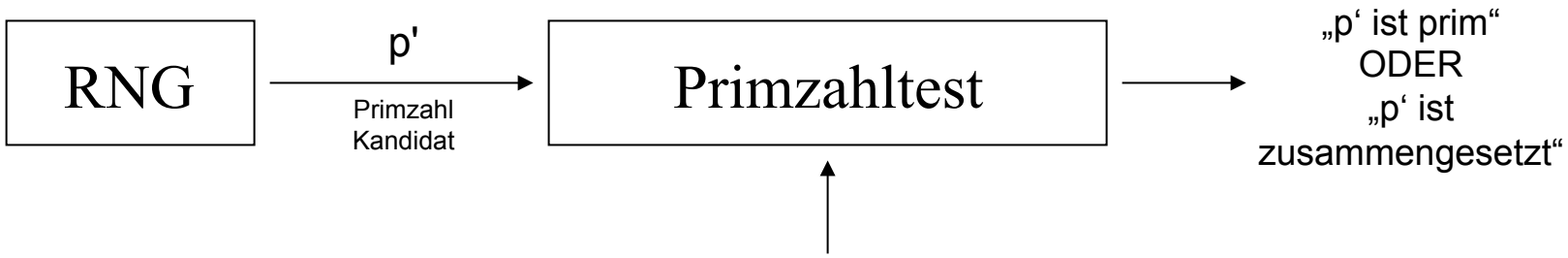
- Unter normalen Annahmen können wir die Transformation und die inverse Transformation vernachlässigen
- Unter der Annahme, *n* hat *t+1* Bit sind *p* und *q* beide etwas *t/2* Bit groß
- Die Komplexität wird durch die beiden Exponentiation in der CRT Domäne bestimmt. Die Operanden sind nur *t/2* Bit groß. Komplexität unter Verwendung des Square-and-Multiply Algorithmus:
  - # Quadrierung (eine Exp.): *#Q = 0,5 t*
  - # Multiplikationen im Durchschnitt (eine Exp.): *#M = 0,25 t*
  - Gesamte Komplexität: *2 * (#M + #Q) = 1,5 t*
- Da die Operanden nur die halbe Bitlänge haben, ist jede Operation viermal schneller!
- Daher ist die Exponentiation mit CRT **viermal schneller** als die direkte Exponentiation

Übersicht

- Das RSA-Verfahren
- Praktische Aspekte
- **Erzeugung großer Primzahlen**
- Angriffe und Gegenmaßnahmen
- Lessons Learned

Erzeugung großer Primzahlen
Prinzip

- Die Schlüsselerzeugung von RSA benötigt zwei große Primzahlen *p* und *q*, so dass $n = p * q$ ausreichend groß ist
- *p* und *q* haben typischerweise die halbe Größe von von *n*
- Zur Erzeugung von Primzahlen werden zufällige ganze Zahlen erzeugt und auf Primalität getestet:



- Der Zufallszahlengenerator (Random Number Generator - RNG) sollte nichtvorhersagbare Zahlen produzieren, damit ein Angreifer nicht die Faktorisierung von *n* erraten kann

Erzeugung großer Primzahlen
Primzahltests

- Faktorisierung von *p* und *q* um auf Primalität zu testen ist typischerweise nicht möglich
- Anstelle der Faktorisierung interessiert uns jedoch nur, ob *p* und *q* zusammengesetzte Zahlen sind
- Typische Primzahltests sind probabilistisch, d.h., ihre Aussage ist nicht 100% sondern nur mit hoher Wahrscheinlichkeit korrekt
- Ein probabilistischer Test hat zwei Ausgaben:
  - „p‘ ist zusammengesetzt “ – stimmt immer
  - „p‘ ist eine Primzahl “ – nur mit einer gewissen Wahrscheinlichkeit korrekt
- Bekannte Primzahltests:
  - Fermat Primzahltest
  - Miller-Rabin Primzahltest

Erzeugung großer Primzahlen
Primzahltest: Fermat

- Idee: Kleiner Satz von Fermat gilt für alle Primzahlen, d.h. finden wir eine Zahl *p'* mit $a^{p'-1} \not\equiv 1 \bmod p'$, ist diese keine Primzahl

Algorithmus: Fermat Primzahltest

Eingabe: Primzahlkandidat *p'*, Sicherheitsparameter *s*

Ausgabe: „*p'* ist zusammengesetzt" oder „*p'* ist wahrscheinlich eine Primzahl"

1. **FOR** *i = 1* **TO** *s*

2. wähle zufälliges $a \in \{2,3, ..., p'-2\}$

3. **IF** $a^{p'-1} \not\equiv 1 \bmod p'$ **THEN**

4. **RETURN** „*p'* ist zusammengesetzt"

5. **RETURN** „*p'* ist wahrscheinlich eine Primzahl"

- Für bestimmte Zahlen (Carmichael Zahlen) gibt der Test oft „*p'* ist wahrscheinlich eine Primzahl" aus, obwohl diese Zahlen zusammengesetzt sind
- Daher wird der Miller-Rabin Test bevorzugt

Erzeugung großer Primzahlen
Primzahltest: Miller-Rabin

- Der mächtigere Miller-Rabin Primzahltest basiert auf folgendem Satz:

Satz

Sei p' eine ungerade Zahl mit folgender Eigenschaft

$$p' - 1 = 2^u * r$$

wobei r ungerade ist. Wenn wir einen ganze Zahl a mit

$$a^r \not\equiv 1 \bmod p' \quad \text{and} \quad a^{r2j} \not\equiv p' - 1 \bmod p'$$

für alle $j = \{0,1, ..., u-1\}$ finden können, ist p' zusammengesetzt.

Ansonsten ist p' wahrscheinlich eine Primzahl.

- Wir nutzen diesen Satz für den folgenden Primzahltest

Erzeugung großer Primzahlen

Primzahltest: Miller-Rabin

Algorithmus: Miller-Rabin Primzahltest

Eingang: Primzahlkandidat p' mit $p'-1 = 2^u * r$ Sicherheitsparameter s

Ausgang: „p' ist zusammengesetzt“ oder „p' ist wahrscheinlich eine Primzahl“

1. **FOR** $i = 1$ **TO** s

2. Wähle zufälliges $a \varepsilon \{2,3, ..., p'-2\}$

3. $z \equiv a^r \bmod p'$

4. **IF** $z \neq 1$ **AND** $z \neq p'-1$ **THEN**

5. **FOR** $j = 1$ **TO** $u-1$

6. $z \equiv z^2 \bmod p'$

7. **IF** $z = 1$ **THEN**

8. **RETURN** „p' ist zusammengesetzt“

9. **IF** $z \neq p'-1$ **THEN**

10. **RETURN** „p' ist zusammengesetzt“

11. **RETURN** „p' ist wahrscheinlich eine Primzahl“

Übersicht

- Das RSA-Verfahren
- Praktische Aspekte
- Erzeugung großer Primzahlen
- **Angriffe und Gegenmaßnahmen**
- Lessons Learned

Angriffe und Gegenmaßnahmen
Übersicht

- Es gibt zwei grundlegende Arten von Angriffen auf Kryptosysteme:
  - **Analytische Angriffe**, welche versuchen, die zugrundeliegende mathematische Struktur zu brechen
  - Implementierungen, welche versuchen, die real existierende Implementierung durch Ausnutzen von **Schwachstellen in Soft- oder Hardware** zu brechen

Angriffe und Gegenmaßnahmen
Analytische Angriffe

- Typische Angriffsvektoren auf RSA
  - **Mathematische Angriffe**
    - Der beste bekannte Angriff auf RSA ist die Faktorisierung des Moduls n, um $\Phi(n)$ zu bestimme
    - Kann durch Wahl eines großen Moduls n verhindert werden
    - Der aktuelle Faktorisierungsrekord liegt bei 768 Bit. Daher wird für n eine Bitlänge 1024 bis 3072 Bit empfohlen
  - **Protokoll Angriffe**
    - Ausnutzen der Malleabilität von RSA, d.h. der Eigenschaft, dass ein Chiffrat in ein anderes Chiffrat transformiert werden kann, welches dann zu einem gültigen Klartext entschlüsselt werden kann – ohne dass man den privaten Schlüssel kennt
    - Kann durch entsprechendes Padding verhindert werden

Angriffe und Gegenmaßnahmen
Implementierungsangriffe

- Inplementierungsangriffe
  - **Seitenkanalangriffe**
    - Nutzen unbeabsichtigten Informationsabfluss durch physikalische Effekte aus (z.B. Stromverbrauch, elektromagnetische Abstrahlung, ...)
  - Angriffe mit **Fault Injection**
    - Einbringen von Fehlern während der CRT auf dem System läuft. Kann zu vollständigem Verlust des geheimen Schlüssels führen

Übersicht

- Das RSA-Verfahren
- Praktische Aspekte
- Erzeugung großer Primzahlen
- Angriffe und Gegenmaßnahmen
- **Lessons Learned**

Lessons Learned

- RSA ist derzeit das am meisten verbreitete asymmetrische Kryptosystem
- Haupteinsatzgebiet von RSA sind Schlüsseltransport und digitale Signatur
- Der öffentliche Exponent *e* kann kurz gewählt werden, der private Schlüssel *d* muss die volle Länge des Moduls *n* haben
- RSA basiert auf der Schwierigkeit, *n* zu faktorisieren
- Gegenwärtig können1024 Bit nicht faktorisiert werden, aber gegenwärtiger Fortschritt in Faktorisierung könnte dies in 10-15 Jahren ermöglichen. Für Langzeitsicherheit sollte daher RSA mit mindestens 2048 oder 3076 Bit Moduln verwendet werden
- Eine direkte Implementierung der RSA-Mechanismen würde einige Angriffe erlauben, so dass man in der Praxis RSA mit Padding verwendet