KRY04 - MNG

Model 2019

Kryptografie

Část 4 Asymetrické algoritmy

Post 19/20

Souhrnné materiály

Ver 0.1

KRY

Kryptografie

Pre 2011/12
Pre 2013/14 NTRU
Pre 2015/16
Post 2015/16
Pret 2016/17
Post 2017/18
2020 MNGprep

KRY 4

Asymetrická kryptografie

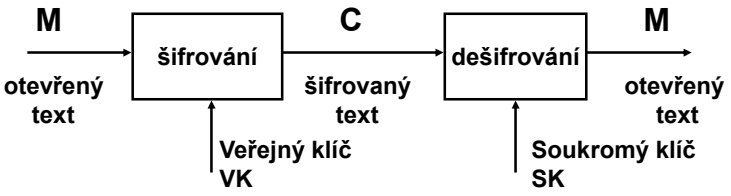
Asymetrická kryptografie

- **Objevili ji Diffie a Hellman, a nezávisle na nich Merkle (1976)**
  - **Idea: použít pro šifrování a dešifrování dva různé klíče**
- **Každý uživatel generuje dvojici klíčů**
  - **Veřejný klíč je zveřejněn**
  - **Soukromý je udržován v tajnosti a je výpočetně nezvládnutelné jej spočítat z veřejného klíče a zpráv**
  - **Klíče jsou v svázány párech**
- **Aplikace**
  - **Šifrování**
  - **Elektronický podpis**
  - **Výměna klíčů (Key Exchange) pro ustavení klíče relace**

Diffie

Hellman

Asymetrický algoritmus 1

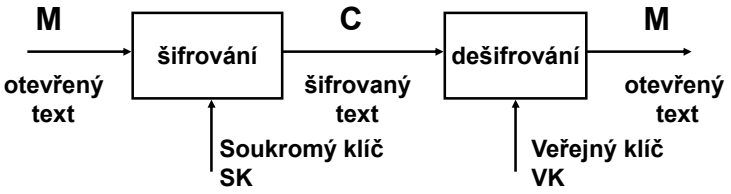


- Uživatel tají soukromý klíč, zveřejní veřejný klíč

Důvěrnost
Autentizace
Integrita
Nepopiratelnost

©Petr Hanáček — CLACRYPT Slide 3

Asymetrický algoritmus 2



- Uživatel tají soukromý klíč, zveřejní veřejný klíč
- Elektronický podpis

Důvěrnost
Autentizace
Integrita
Nepopiratelnost

©Petr Hanáček — CLACRYPT Slide 4

Asymetrické algoritmy

- **Knapsack**
  - **Knapsack – první algoritmus, Merkle-Hellman, 1976**
- **Faktorizace čísel**
  - **RSA**
- **Diskrétní logaritmus**
  - **DSS (DSA)**
  - **El Gamal**
  - **Diffie-Hellman**
- **Eliptické křivky**
  - **Např. ECDSA**

 CLACRYPT Slide 5

Akronymy (některé nové)

- DLC: **Discrete Logarithm Cryptography**
  - FFC**: Finite Field Cryptography**
    - » **Digital Signature Algorithm (DSA), Diffie-Hellman (DH) and MQV***
  - ECC**: Elliptic Curve Cryptography**
    - » **ECDSA, ECDH a ECMQV***
  - **Jsou bezpečné, pokud je obtížné nalézt diskrétní logaritmy v prostoru FF nebo EC**
- IFC: **Integer Factorization Cryptography**
  - **RSA je jediný algoritmus v této kategorii, který se používá**
    - » **Reverzibilní: dá se použít pro podpis i šifrování**
  - **Je bezpečný, pokud je obtížné faktorizovat velká čísla**

*** MQV: Menenzes, Qu a Vanstone – nový bezpečný protokol pro dohodu na klíči, který využívá DLC**

 CLACRYPT Slide 6

Problém faktorizace

- **Máme kladné celé číslo n, máme nalézt prvočísla p_1, …, p_k taková, že $n=p_1^{e1}p_2^{e2}…p_k^{ek}$**

RSA Factoring Challenge
The RSA challenges ended in 2007.[1] RSA Laboratories stated: "Now that the industry has a considerably more advanced understanding of the cryptanalytic strength of common symmetric-key and public-key algorithms, these challenges are no longer active."[2]

RSA Number	Decimal digits	Binary digits	Cash prize offered	Factored on	Factored by
RSA-100	100	330		April 1991	Arjen K. Lenstra
RSA-110	110	364		April 1992	Arjen K. Lenstra and M.S. Manasse
RSA-120	120	397		June 1993	T. Denny et al.
RSA-129	129	426	$100 USD	April 1994	Arjen K. Lenstra et al.
RSA-130	130	430		April 10, 1996	Arjen K. Lenstra et al.
RSA-140	140	463		February 2, 1999	Herman J. J. te Riele et al.
RSA-150[1]	150	496		April 16, 2004	Kazumaro Aoki et al.
RSA-155	155	512		August 22, 1999	Herman J. J. te Riele et al.
RSA-160	160	530		April 1, 2003	Jens Franke et al., University of Bonn
RSA-170	170	563			*open*
RSA-576	174	576	$10,000 USD	December 3, 2003	Jens Franke et al., University of Bonn
RSA-180	180	596			*open*
RSA-190	190	629			*open*
RSA-640	193	640	$20,000 USD	November 2, 2005	Jens Franke et al., University of Bonn
RSA-200	200	663		May 9, 2005	Jens Franke et al., University of Bonn
RSA-210	210	696			*open*
RSA-704	212	704	$30,000 USD		*open*
RSA-220	220	729			*open*
RSA-230	230	762			*open*
RSA-232	232	768			*open*
RSA-768	232	768	$50,000 USD		*open*

©Petr Hanáček

Modulární aritmetika

- **Sčítání, odčítání a násobení modulo n**
- $a + b \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
- $a - b \bmod n = ((a \bmod n) - (b \bmod n)) \bmod n$
- $a \times b \bmod n = ((a \bmod n) \times (b \bmod n)) \bmod n$

©Petr Hanáček CLACRYPT Slide 8

Kongruence

- Dvě celá čísla *a* a *b* jsou kongruentní modulo n
  - Psáno **a ≡ b**
- pokud
  - ***a* mod *n* = *b* mod *n***
- **nebo**
  - ***a* = *b* + *kn*, *k* ∈ Z**

 CLACRYPT Slide 9

Multiplikativní inverze modulo n

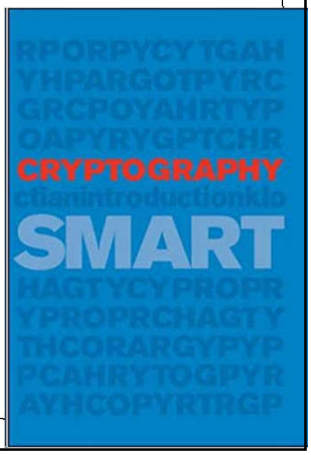
- **Multiplikativní inverze hodnoty *a* modulo n je celé číslo x takové, že**
  - $a \times x \equiv 1 \pmod{n}$
- Označuje se
  - a^{-1} mod n
- Pak platí:
  - $a \times a^{-1} \equiv 1 \pmod{n}$

 CLACRYPT Slide 10

Učebnice

4

- Nigel Smart: Cryptography - An Introduction, 3rd Edition,
  - Mcgraw-Hill College, 3rd Edition, 2013
  - ISBN-10: 0077099877
- Kapitoly
  - Kapitola 11 - Basic Public Key Encryption Algorithms
    - » Zajímavé jsou pro nás podkapitoly 1, 2 a 3



The third edition is now online. You may make copies and distribute the copies of the book as you see fit, as long as it is clearly marked as having been authored by N.P. Smart.

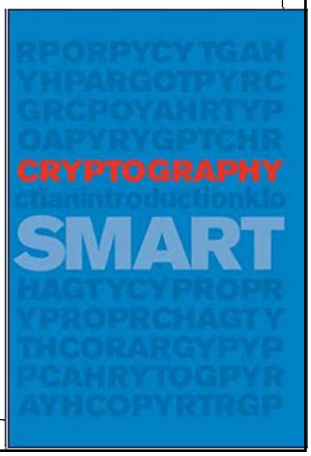
Učebnice je v dokumentovém skladu

©Petr Hanáček

Učebnice

4

- Nigel Smart: Cryptography - An Introduction, 3rd Edition,
  - Mcgraw-Hill College, 3rd Edition, 2013
  - ISBN-10: 0077099877
- Kapitoly
  - Kapitola 14 - Key Exchange and Signature Schemes
    - » Zajímavé jsou pro nás podkapitoly 1, 2 a 3



The third edition is now online. You may make copies and distribute the copies of the book as you see fit, as long as it is clearly marked as having been authored by N.P. Smart.

Učebnice je v dokumentovém skladu

©Petr Hanáček

RSA

Algoritmus RSA

- **Rivest, Shamir, Adelman 1978**
  - **Údajně objeven už v GCHQ (Ellis a Cocks) v roce 1973**
- **Asymetrický šifrovací algoritmus s veřejným klíčem**
- **Založený na problému faktorizace velkých čísel**
- **Funguje jako bloková šifra, kde blok je celé číslo mezi 0 a n**

RSA – Generování klíčů

- **Klíče**
  - *n*: veřejný modulus
  - *e*: veřejný exponent (typicky 3 nebo $2^{16}+1$)
  - *d*: soukromý exponent
  - *p*,*q*: činitele (factors) modulu n
    - » $n = p \times q$
  - Musí platit vztah
    - » $d \times e \bmod (p-1)(q-1) = 1$
- Veřejný klíč je (*n*,*e*).
- Soukromý klíč je (*n*,*d*).
- Postup
  - **Vygeneruj prvočísla *p* a *q*, *n=pq***
  - **Spočti Φ(n)=(*p*-1)(*q*-1)**
  - **Zvol hodnotu *e*<Φ(*n*) takovou, že *gcd*(Φ(*n*),e) = 1**
  - **Spočti *d* tak, že $d = e^{-1} \bmod \Phi(n)$**

 CLACRYPT Slide 15

Šifrování / Dešifrování

- **Zpráva m (celé číslo)**
- **Zašifrovaný text c (celé číslo)**
- **Šifrování veřejným klíčem**
  - $c = m^e \bmod n$
- **Dešifrování soukromým klíčem**
  - $m = c^d \bmod n$
- **Použití**
  - **Utajení**

 CLACRYPT Slide 16

Šifrování / Dešifrování

- **Zpráva m (celé číslo)**
- **Zašifrovaný text s (signature)**
- **Šifrování veřejným klíčem**
  - $s = m^d \bmod n$
- **Dešifrování veřejným klíčem**
  - $m = s^e \bmod n$
- **Použití**
  - **Elektronický podpis**

Příklad RSA

- **Generování klíčů**
  - **Vygenerujeme prvočísla p a q**
    - » **p=7, q=17, n=pq=119, Φ(119)=96**
  - **Zvolíme e=5; vypočtem d=77**

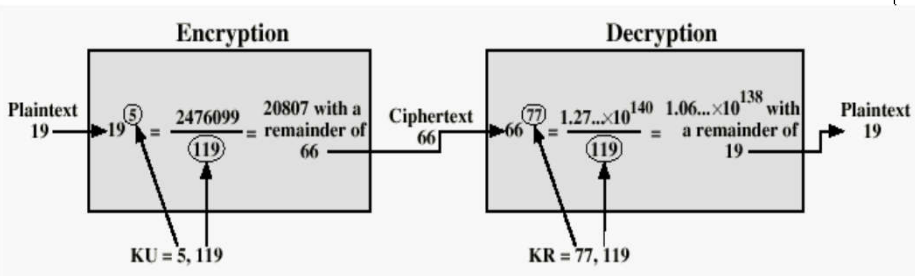


Figure 3.9 Example of RSA Algorithm

Příklad RSA

- p = 11, q = 7, n=77, d = 13, d = 37
- m = 15
- Šifrování
  - $c \equiv m^d \bmod n$
  - $c \equiv 15^{37} \pmod{77} = 71$
- Dešifrování
  - $m \equiv c^e \bmod n$
  - $m \equiv 71^{13} \pmod{77} = 15$

RSA Example

- Encryption Algorithm :
  - Let P = 7 be the plaintext (message)
  - Let p = 11 and q = 7
  - thus, n = p * q = 11 * 13 = 143
  - $\Phi(n) = (p-1) * (q-1) \quad = 10 * 12 \quad = 120$
  - Let e = 11 (relatively prime to (p-1) * (q-1))
  - Thus,
    $C = P^e \bmod n = 7^{11} \bmod 143 = 106$
- Decryption Algorithm:
  - decryption key is the inverse of the encryption key, ie;
    $e * d == 1 \bmod (p-1) * (q-1)$
  - Thus, inverse of 11 mod 120 = 11 because 11 * 11 = 121 = 1 mod 120
  - Having e = d is possible but not recommended
  - To decrypt :
    $D(106) = C^d \bmod n = 106^{11} \bmod 143 = 7$

Rychlost: operace s SK

- **Připomeňme:**
  - $c \equiv m^d \bmod n$
- **Modulární umocňování**
  - n, m ~ β bitů; d ~ δ bitů
  - $O(\beta^2\delta)$ bitových operací

Modular-Exponentiation(a,b,n)

1 $d \leftarrow 1$
2 let $\langle b_\delta, b_{\delta-1}, \ldots, b_0 \rangle$ represent b
3 for $i \leftarrow \delta$ downto **0** (δ iterations)
4 do $d \leftarrow (d \cdot d) \bmod n$ $(O(\beta^2))$
5 if $b_i = 1$
6 then $d \leftarrow (d \cdot a) \bmod n$ $(O(\beta^2))$
7 return *d*

Rychlost: operace s VK

- **Připomeňme:**
  - $m \equiv c^e \bmod n$
- **Modulární umocňování**
  - n, c ~ β bitů; e ~ ε bitů
  - $O(\beta^2\varepsilon)$ bitových operací
- **e může být malé liché prvočíslo (např. 3)**
  - Šifrování je rychlé
  - Ověření podpisu je rychlé
  - Bezpečnost je zachována

Výkon RSA

- Knihovna Crypto++ v.5.2.1, (Windows XP SP1)
- Hardware: Pentium 4, 2.1 GHz
  - » alg. ms/oper.
  - » RSA-1024 šifr. 0.18
  - » RSA-1024 dešifr. 4.77
  - » RSA-2048 šifr. 0.45
  - » RSA-2048 dešifr. 28.41

NIST:

Symetrický klíč	Asymetrický klíč
≤ 64 bitů	512 bitů
80 bitů	1024 bitů
128 bitů	3072 bitů
256 bitů (AES)	15360 bitů

©Petr Hanáček CLACRYPT Slide 23

Útoky na algoritmus RSA

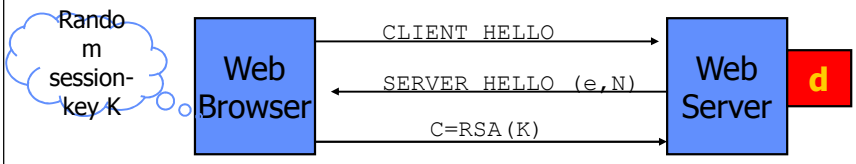
- Pokud útočník umí rozložit *n,* na činitele *p* a *q*, může dopočítat soukromý klíč
- Pokud útočník uhodne hodnotu *(p-1)(q-1),* vypočte soukromý klíč i bez faktorizace n
- Vždy doplňovat zprávy náhodnými čísly, aby m mělo přibližně stejnou velikost jako *n*
  - – Pokud e je malé, nemuselo by se uplatnit mod n
- Nepoužívat malé hodnoty pro *d*

©Petr Hanáček CLACRYPT Slide 24

Textbook RSA

- **Popisovaný systém je tzv. Textbook RSA**
  - **V základní verzi je zcela nepoužitelný**
  - **Mnoho různých útoků**
- **Slovníkový útok**
  - **Šifrujeme jenom několik málo druhů zpráv**

Jednoduchý útok na textbook RSA



- **Session-key K má 64 bitů, tzn. K ∈ {0,…,2^{64}}**
  Útočník vidí : $C = K^e \pmod N$.
- **Předpokládejme** $K = K_1 \cdot K_2$ **kde** $K_1, K_2 < 2^{34}$. **(pravď. ≈20%)**
  Pak: $C/K_1^e = K_2^e \pmod N$
- **Vyrobíme tabulku:** $C/1^e, C/2^e, C/3^e, \ldots, C/2^{34e}$ **. V čase:** 2^{34}
  Pro $K_2 = 0,\ldots, 2^{34}$ **hledáme zda** K_2^e **je v tabulce. V čase:** $2^{34} \cdot 34$
- **Složitost útoku:** $\approx 2^{40} << 2^{64}$

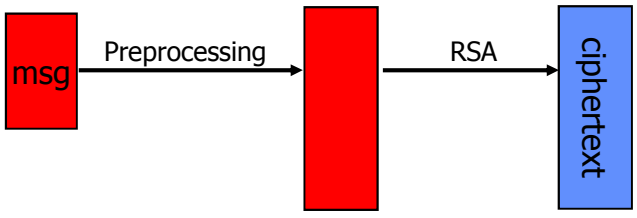
Jiný útok na textbook RSA

- **Chosen ciphertext attack**
  - **Útok: přimějeme odesílatele, aby podepsal (dešifroval) naši zprávu**
  - **Vstup: originální zašifrovaná zpráva $C=M^e \bmod n$**
  - **Vytvoříme**
    - » **$X=R^e \bmod n$, pro náhodné R**
    - » **$Y=XC \bmod n$**
    - » **$T=R^{-1} \bmod n$**
  - **Požádáme odesílatel o podepsání Y, získáme $U=Y^d \bmod n$**
  - **Spočteme**
    - » **$TU \bmod n = R^{-1}Y^d \bmod n = R^{-1} X^d C^d \bmod n = C^d \bmod n = M$**
  - **Využívá zachování vlastnosti násobení v modulární aritmetice**
- **Závěr:**
  - **Nikdy nepodepisovat náhodné zprávy**
  - **Podepisovat jenom haše**
  - **Používat různé klíče pro šifrování a podpis**

 CLACRYPT Slide 27

Použití RSA

- **Nikdy nepoužívat textbook RSA**
- **RSA v praxi:**

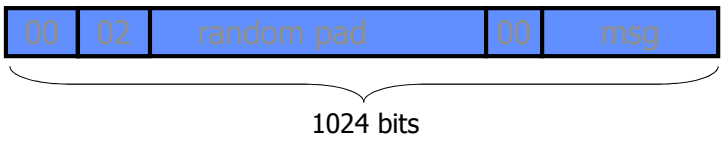


- **Otázky:**
  - **Jak provést předzpracování (zarovnání, padding)?**
  - **Je bezpečnost výsledného systému prokazatelná?**

 CLACRYPT Slide 28

PKCS1 V1.5

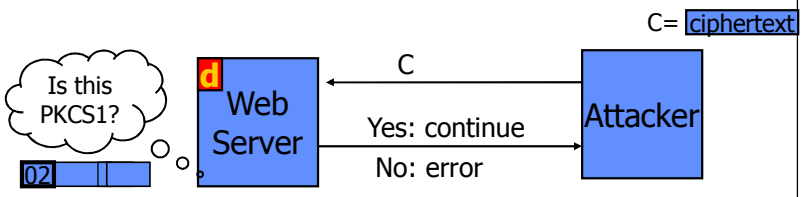
- **PKCS1 mode 2: (šifrování)**



- **Výsledná hodnota je zašifrovaná pomocí RSA**
- **Velmi rozšířené, např. ve webových serverech a prohlížečích**
- **Neexistuje formální analýza bezpečnosti**

Útok na PKCS1

- **Bleichenbacher 98. Chosen-ciphertext attack**
- **Např. PKCS1 použité v SSL:**

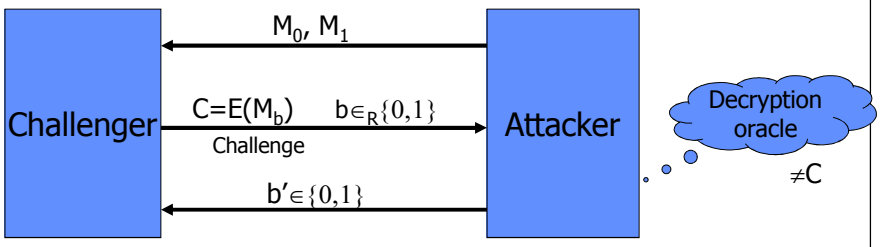


⇒ **Útočník může otestovat zda 16 bitů plaintextu je'02'**

- **Útok: pro dešifrování zašifrovaného textu C:**
  - **Náhodně vyber $r \in Z_N$. Spočti $C' = r^e \cdot C = (r \cdot \text{PKCS1}(M))^e$**
  - **Pošli C' serveru a využij odpovědi**

Chosen ciphertext security (CCS)

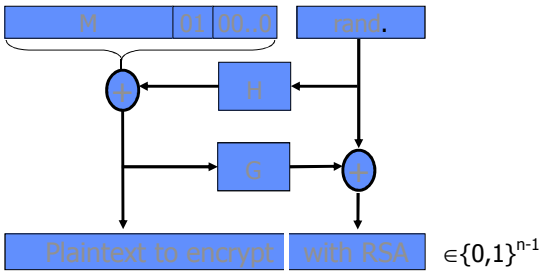
- Žádný útočník nemůže vyhrát následující hru:
  (s výrazně vyšší pravděpodobností)



Útočník vyhrává pokud
b=b'

PKCS1 V2.0 - OAEP

- **Nová funkce pro předzpracování: OAEP**
  - **Optimal Asymmetric Encryption Padding**



- **OAEP je CCS pokud H a G jsou "*náhodná orákula*"**
- **V praxi: použití hašovacích funkcí pro H a G**

Zákeřnosti implementace OAEP

```
OAEP-decrypt(C) {
    error = 0;
    ........
    if ( RSA^-1(C) > 2^(n-1) )
        { errcode =1;  goto exit; }
    ........
    if ( pad(OAEP^-1(RSA^-1(C))) != "01000" )
        { errcode = 2;  goto exit; }
}
```

©Petr Hanáček CLACRYPT Slide 33

Zákeřnosti implementace OAEP

```
OAEP-decrypt(C) {
    error = 0;
    ........
    if ( RSA^-1(C) > 2^(n-1) )
        { error =1;  goto exit; }
    ........
    if ( pad(OAEP^-1(RSA^-1(C))) != "01000" )
        { error = 1;  goto exit; }
}
```

- Problém: časový útok dává informaci o typu chyby.
  ⇒ Útočník může dešifrovat libovolný šifrový text C
- Ponaučení: Neimplementuj to sám…

©Petr Hanáček CLACRYPT Slide 34

Implementace RSA (a dalších algoritmů)

 CLACRYPT Slide 35

Počet bitů vs. počet číslic

- $10^{\#\text{číslic}} = 2^{\#\text{bitů}}$
- $\#\text{číslic} = (\log_{10} 2) \cdot \#\text{bitů} \approx 0.30 \cdot \#\text{bitů}$
  - 256 bitů = 77 číslic
  - 384 bitů = 116 číslic
  - 512 bitů = 154 číslic
  - 768 bitů = 231 číslic
  - 1024 bitů = 308 číslic
  - 2048 bitů = 616 číslic

 CLACRYPT Slide 36

Modulární umocňování

- Naivní řešení – opakovaným násobením
- $Y = X^E \bmod N = X \times X \times X \times X \times X \ldots \times X \times X \bmod N$
  E-krát
- E může mít velikost $2^{1024} \approx 10^{308}$
- Problémy
  - 1. Pro uložení X^E před operací modulo je třeba velké množství paměti
  - 2. Počet výpočtů je neúnosně velký
- Řešení
  - 1. Modulo provádět po každém násobení
  - 2. Chytřejší algoritmus

©Petr Hanáček CLACRYPT Slide 37

Binární umocňování zprava doleva

$$Y = X^E \bmod N$$

$$E = (e_{L-1}, e_{L-2}, \ldots, e_1, e_0)_2$$

S: $X \quad X^2 \bmod N \quad X^4 \bmod N \quad X^8 \bmod N \quad \ldots \quad X^{2^{L-1}} \bmod N$

E: $e_0 \quad e_1 \quad e_2 \quad e_3 \quad \ldots \quad e_{L-1}$

$$Y = X^{e_0} \cdot (X^2 \bmod N)^{e_1} \cdot (X^4 \bmod N)^{e_2} \cdot (X^8 \bmod N)^{e_3} \cdot \ldots \cdot (X^{2^{L-1}} \bmod N)^{e_{L-1}}$$

$$(X^a)^b = X^{ab} \qquad X^a \cdot X^b = X^{a+b}$$

$$Y = X^{e_0 + 2 \cdot e_1 + 4 \cdot e_2 + 8 \cdot e_3 + \ldots + 2^{L-1} \cdot e_{L-1}} \bmod N =$$

$$= X^{\sum_{i=0}^{L-1} e_i \cdot 2^i} = X^E \bmod N$$

©Petr Hanáček CLACRYPT Slide 38

Binární umocňování zprava doleva

$$Y = 3^{19} \bmod 11$$

$$E = 19 = 16 + 2 + 1 = (10011)_2$$

S:	X	$X^2 \bmod N$	$X^4 \bmod N$	$X^8 \bmod N$	$X^{16} \bmod N$
	3	$3^2 \bmod 11 = 9$	$9^2 \bmod 11 = 4$	$4^2 \bmod 11 = 5$	$5^2 \bmod 11 = 3$

E:	e_0	e_1	e_2	e_3	e_4
	1	1	0	0	1

$$Y = X \cdot X^2 \bmod N \cdot 1 \cdot 1 \cdot X^{16} \bmod N =$$

$$3 \cdot 9 \cdot 1 \cdot 1 \cdot 3 \bmod 11$$

$$= X^{19} \bmod N$$

$$(27 \bmod 11) \cdot 3 \bmod 11 = 5 \cdot 3 \bmod 11 = 4$$

©Petr Hanáček CLACRYPT Slide 39

Binární umocňování zleva doprava

$$Y = X^E \bmod N$$

$$E = (e_{L-1}, e_{L-2}, \ldots, e_1, e_0)_2$$

E: e_{L-1} e_{L-2} e_{L-3} … e_1 e_0

$$Y = ((\ldots(((1^2 \cdot X^{e_{L-1}})^2 \cdot X^{e_{L-2}})^2 \cdot X^{e_{L-3}})^2 \ldots)^2 \cdot X^{e_1})^2 \cdot X^{e_0} \bmod N$$

$$(X^a)^b = X^{ab} \qquad X^a \cdot X^b = X^{a+b}$$

$$Y = X^{(e_{L-1} \cdot 2 + e_{L-2}) \cdot 2 + e_{L-3}) \cdot 2 + \ldots + e_1) \cdot 2 + e_0} \bmod N =$$

$$= X^{2^{L-1} \cdot e_{L-1} + 2^{L-2} \cdot e_{L-2} + 2^{L-3} \cdot e_{L-3} + \ldots + 2 \cdot e_1 + e_0} \bmod N = X^{\sum_{i=0}^{L-1} e_i \cdot 2^i} =$$

$$= X^E \bmod N$$

©Petr Hanáček CLACRYPT Slide 40

Binární umocňování zleva doprava

$$Y = 3^{19} \bmod 11$$

$$E = 19 = 16 + 2 + 1 = (10011)_2$$

E:	e_4	e_3	e_2	e_1	e_0
	1	0	0	1	1

$$Y = ((...(((1^2 \cdot X)^2 \cdot 1)^2 \cdot 1)^2 \cdot X)^2 \cdot X \bmod N$$

$$= (((3^2 \bmod 11))^2 \bmod 11)^2 \bmod 11 \cdot 3)^2 \bmod 11 \cdot 3 \bmod 11$$
$$= (81 \bmod 11)^2 \bmod 11 \cdot 3)^2 \bmod 11 \cdot 3 \bmod 11 =$$
$$= (5 \cdot 3)^2 \bmod 11 \cdot 3 \bmod 11 =$$
$$= 4^2 \bmod 11 \cdot 3 \bmod 11 =$$
$$= 5 \cdot 3 \bmod 11 = \mathbf{4}$$

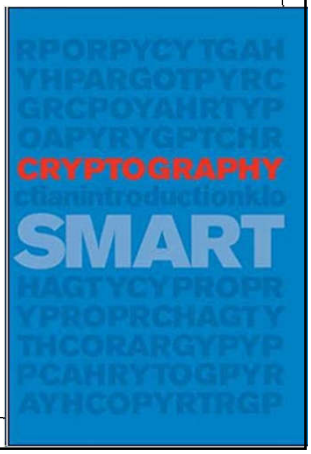
$$Y = (X^8 \cdot X)^2 \cdot X \bmod N = X^{19} \bmod N$$

©Petr Hanáček — CLACRYPT Slide 41

Učebnice

4

- **Nigel Smart: Cryptography - An Introduction, 3rd Edition,**
  - Mcgraw-Hill College, 3rd Edition, 2013
  - ISBN-10: 0077099877
- **Kapitoly**
  - **Kapitola 15 - Implementation Issues**
    - » Zajímavé jsou pro nás podkapitoly 1 a 2



The third edition is now online. You may make copies and distribute the copies of the book as you see fit, as long as it is clearly marked as having been authored by N.P. Smart.

Učebnice je v dokumentovém skladu

©Petr Hanáček

Umocňování $Y = X^E \bmod N$

- **Zprava doleva**
- **Zleva doprava**

$$E = (e_{L-1}, e_{L-2}, \ldots, e_1, e_0)_2$$

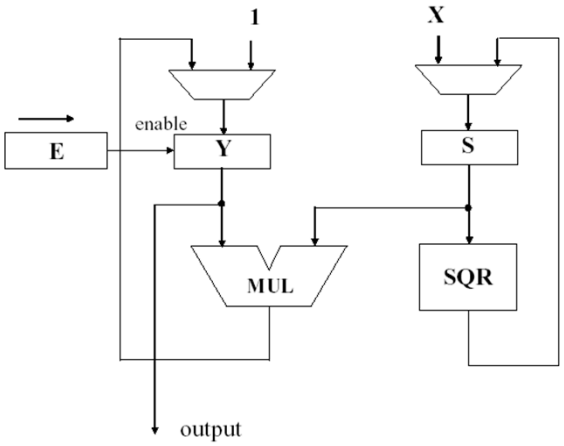
```
Y = 1;
S = X;
for  i=0 to L-1
 {
   if (e_i == 1)
     Y = Y · S mod N;
  S = S^2 mod N;
 }
```

```
Y = 1;
for  i=L-1 downto 0
 {
  Y = Y^2 mod N;
   if (e_i == 1)
     Y = Y · X mod N;
 }
```

 CLACRYPT Slide 43

Binární umocňování zprava doleva

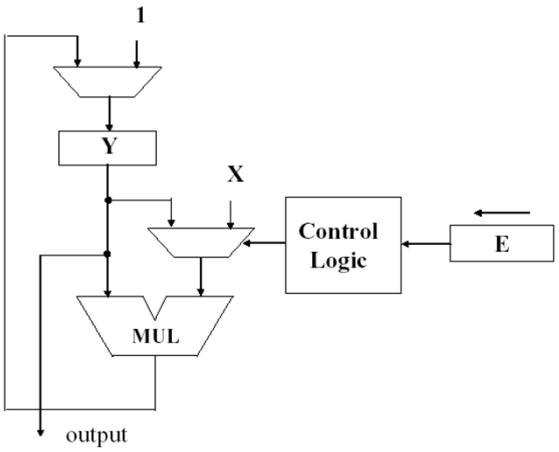
- **Realizace v HW**



 CLACRYPT Slide 44

Binární umocňování zleva doprava

- **Realizace v HW**



Modulární násobení

Multiplication

- Paper-and-pencil $\theta(k^2)$
- Karatsuba $\theta(k^{3/2})$
- Schönhage-Strassen (FFT) $\theta(k \cdot \ln(k))$

Multiplication combined with modular reduction

- Montgomery algorithm $\theta(k^2)$

Modular Reduction

- classical $\theta(k^2)$
- Barrett complexity same as multiplication used
- Selby-Mitchell $\theta(k^2)$

Knapsack

KNAPSACK PROBLEM

- **A problem to pack items into a knapsack**
  - **How to select items such that the 'sum' (amount of space they take up) exactly equals the knapsack capacity (the target)**
  - **The same to adding integers such that their sum equals the target ie :**
  - **Given a set S = { a_1, a_2,...,a_n} and a target sum, T, where each a_j >= 0, is there V = {v_1, v_2,...v_n}, where v_i = 0 or 1 such that**

$$\Sigma (a_i * v_i) = T$$

- **Example :**
  - **Let S = {4, 7, 1, 12, 10} and T = 17**
  - **Since T = 17 = 4 +1+12 then there exist a selection vector V = {1, 0, 1, 1, 0}**

Merkle-Hellman Knapsack

- **Merkle-Hellman Knapsack Algorithm**
  - First public-key cryptography algorithm (1976)
  - Broken by Shamir and Zippel in 1980, showing a reconstruction of superincreasing knapsacks from the normal knapsacks
- **Needs two keys**
  - a) Superincreasing knapsack / private key
  - b) Hard knapsack / public key
- **Superincreasing knapsack**
  - A set of integers, where each integer is greater than the sum of all preceeding integers.
  - Example : [1, 4, 11, 17, 38, 73] , [1, 2, 4, 9, 19]
  - we can use any method of generating the sequence as for as the set satisfies the above condition
- **Hard knapsack**
  - generate hard knapsack from simple knapsack using the following modular operation:
    $h_i = w * s_i \bmod n$
  - where hard knapsack = H = $[h_1, h_2, \ldots h_m]$
  - n = prime number, w = multiplier (both n and w are relatively prime), n > s_m

Example

- **Let S = [1, 2, 4, 9]**
- **Let W = 15 and n = 17**
- **Then**
  - S = [1, 2, 4, 9]
  - i) 1 * 15 mod 17 = 15
  - ii) 2 * 15 mod 17 = 13
  - iii) 4 * 15 mod 17 = 9
  - iv) 9 * 15 mod 17 = 16
- **Therefore**
  - H = [15, 13, 9,16]

Knapsack - encryption

- **Encryption**
  - message is in binary form
  - divide the message in block of m bits
  - m = number of elements in private or public key
  - multiply each corresponding elements in the plaintext with elements in the public key and add up the values.
  - the output is the cipher text
- **Example:**
  - P = 0100101110100101
  - H = [15, 13, 9, 16]
  - Thus,

```
P = 0 1 0 0   1 0 1 1   1 0 1 0   0 1 0 1
  = 0+13+0+0  15+0+9+16 15+0+9+0  0+13+0+16
  = 13          40          24          29
```

- **Therefore**
  - C = 13, 40, 24, 29

 CLACRYPT Slide 51

Knapsack - decryption

- **To decipher, multiply C_i by W^{-1} mod n**
  - Since C = H * P = W * S * P mod n
  - because $W^{-1} * C = W^{-1} * H * P = W^{-1} * W * S * P = S * P \bmod n$
- **Example:**
  - Let the private key S = [1, 2, 4, 9], with W = 15 and n = 17
  - C = 13, 40, 24, 29
  - Then, $13 * 15^{-1} \bmod 17 = 13 * 8 \bmod 17 = 2$
    $40 * 15^{-1} \bmod 17 = 40 * 8 \bmod 17 = 14$
    $24 * 15^{-1} \bmod 17 = 24 * 8 \bmod 17 = 5$
    $29 * 15^{-1} \bmod 17 = 29 * 8 \bmod 17 = 11$
- **Thus, the target sums are 2, 14, 5, 11**
- **Therefore, using the private key**
  - S = [1, 2, 4, 9] with T = 2 V1 = [0, 1, 0, 0]
  - S = [1, 2, 4, 9] with T = 14 V2 = [1, 0, 1, 1]
  - S = [1, 2, 4, 9] with T = 5 V3 = [1, 0, 1, 0]
  - S = [1, 2, 4, 9] with T = 11 V4 = [0, 1, 0, 1]
  - Therefore, the recovered P = 0100101110100101

 CLACRYPT Slide 52

DSA

Problém diskrétního logaritmu DL

- Nechť G je konečná cyklická grupa o velikosti n, generovaná generátorem g, např.
  - G = <g> = {g^i | i = 1, 2, …, n}
  - Nebo {g^i | i = 0, 1, …, n-1}
- Při známém g a i, je snadné spočítat g^i opakovaným násobením
- Problém diskrétního logaritmu
  - Známe $a \in G$
  - Máme nalézt x takové, že $g^x = a$
  - Označujeme $x = \log_g a \ (or \ \mathrm{ind}_g a)$

Příklad 1

- **G = Z^*_{19} = { 1, 2, …, 18}**
  n=18, generátor g = 2

i	1	2	3	4	5	6	7	8	9
g^i	2	4	8	16	13	7	14	9	18

10	11	12	13	14	15	16	17	18
17	15	11	3	6	12	5	10	1

- **Potom**
  - **$\log_2 14 = 7$**
  - **$\log_2 6 = 14$**

Příklad 2

- **G=$GF^*(2^3)$ s polynomem $p(x) = x^3 + x + 1$**
- **G=$Z^*_p/p(x)$ = { 1, x, x^2, 1+x, 1+x^2, x+x^2, 1+x+x^2 }**
  n=7, generátor g = x

i	1	2	3	4	5	6	7
g^i	x	x^2	$x+1$	x^2+x	x^2+x+1	x^2+1	1

- **Potom**
  - **$\log_x (x+1) = 3$**
  - **$\log_x (x^2+x+1) = 5$**
  - **$\log_x (x^2+1) = 6$**

Digital Signature Algorithm (DSA)

- **Navržen NISTem v r. 1991 jako standard (DSS)**
- **Založen na diskrétních logaritmech**
- **Má některé nevýhody**
  - **Nedá se použít pro šifrování nebo dostribuci klíčů**
  - **Rychlejší než RSA při podpisu ale pomalejší při verifikaci**
  - **Přichází v době, kdy už je značně rozšířeno RSA**
  - **Obavy, zda neobsahuje zadní vrátka od NIST**
- **Velikost klíče původně 512 bitů, později zvětšena na 1024 bitů**

 CLACRYPT Slide 57

DSA

- **Veřejné parametry**
  - **p: 1024-bitové náhodné prvočíslo**
  - **q: 160-bitové prvočíslo, faktor (p-1)**
  - **g: $g = h^{(p-1)/q} \bmod p$**
- **Hašovací funkce**
  - **H: hašovací funkce se 160bitovým výstupem**
- **Klíče**
  - *y*: 1024-bitový veřejný klíč
  - *x*: 160-bitový soukromý klíč
    - » $y = g^x \bmod p$

 CLACRYPT Slide 58

Podpis

- **Vygeneruj náhodné 160bitové číslo k<q**
- **Podpis je (r, s), kde**
  - **$r = (g^k \bmod p) \bmod q$**
  - **$s = k^{-1}(H(M)+xr) \bmod q$**
- **Verifikace**
  - **Spočti $w=s^{-1} \bmod q$**
  - **Spočti $u1 = H(M)w \bmod q$**
  - **Spočti $u2 = rw \bmod q$**
  - **Spočti $v = (g^{u1}*y^{u2} \bmod p) \bmod q$**
  - **Pokud v = r pak je podpis v pořádku**

 CLACRYPT Slide 59

Rychlost

		RSA	DSS
Podepsání	*off-line*	n/a	**52**
	on-line	159	**< 1**
Verifikace		**2 až 17**	229
Generování parametrů		n/a	34944
Generování klíčů		4452	52

Údaje jsou v počtu modulárních násobení.

 CLACRYPT Slide 60

Bezpečnost

- **Základní útok**
  - Znáš veřejný klíč *y*, nalezni soukromý klíč
    - » $x = \log_g y$.
- **Problém *diskrétního logaritmu***
- **Bezpečnost je porovnatelná s RSA**

- **Další útoky**
  - **Uhodnutí náhodného čísla k**
    - » **Je-li k známé, dá se spočítat r a x**
  - **Zadní vrátka**
    - » **Pro některé speciální hodnoty p je problém diskrétního logaritmu snadno řešitelný**
    - » **Možnost, že v parametrech NIST jsou zadní vrátka**

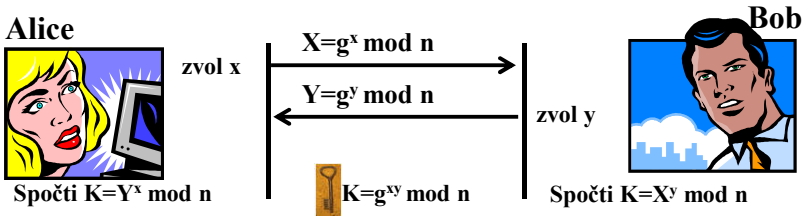
 CLACRYPT Slide 61

Diffie-Hellman

 CLACRYPT Slide 62

Diffie-Hellman

- První algoritmus s veřejným klíčem, založený na problému diskrétních logaritmů modulo n
- Protokol:
  - 1. Algoritmus D-H se použije pro ustavení klíče relace K
  - 2. Klíč relace se použije pro šifrování další komunikace
- Zvolení parametrů:
  - Zvolí se velké prvočíslo n, a hodnota g, která nedělí n



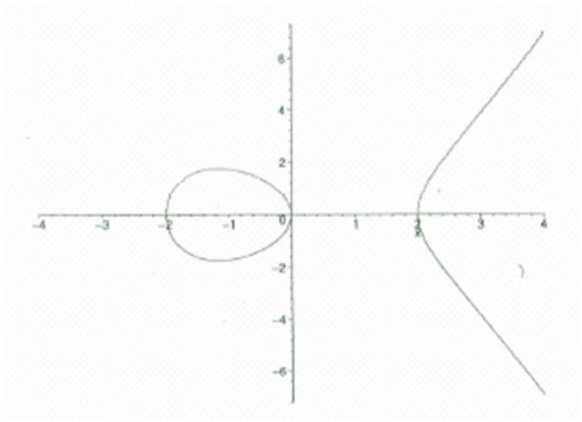
©Petr Hanáček CLACRYPT Slide 63

EC

©Petr Hanáček CLACRYPT Slide 64

Základní aparát EC

- $y^2 = x^3 - 4x$ $\quad 4a^3 + 27b^2 = -256$

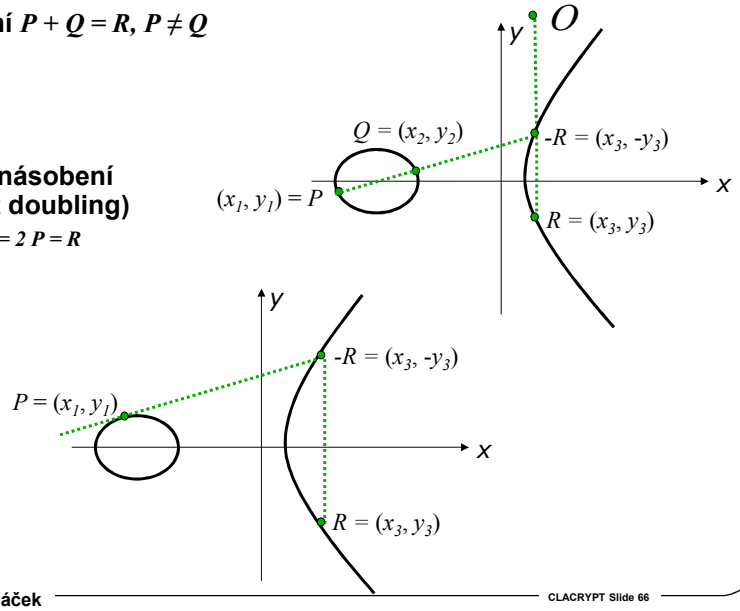


©Petr Hanáček CLACRYPT Slide 65

Základní operace nad EC

- **Sčítání** $P + Q = R$, $P \neq Q$

- **Zdvojnásobení (Point doubling)**
  $P + P = 2P = R$



©Petr Hanáček CLACRYPT Slide 66

Elliptic Curve DLP

- **Snadný výpočet**
  - **Q = kP =** $\underbrace{P + P + \cdots + P}_{k \text{ times}}$

    kde P je bod na křivce, k je celé číslo
- **Obtížný výpočet**
  - **Máme danou křivku, bod P a kP**
    - » **Je obtížné vypočítat k**
  - **Jde o tzv. Elliptic Curve Discrete Logarithm Problem (ECDLP)**

 CLACRYPT Slide 67

Vlastnosti EC vzhledem k DSA,RSA

- **Kryptosystémy s EC poskytují největší míru obtížnosti na jeden bit délky klíče ze všech známých asymetrických systémů**
- **Zdá se, že problém ECDLP je mnohem obtížnější než problémy faktorizace a DL**
- **Síla kryptosystému s EC stoupá s délkou klíče rychleji než síla RSA**

 CLACRYPT Slide 68

Bezpečnost EC systémů

Počet bitů EC	MIPs roky	Počet bitů RSA
120	10^{12}	1024
320	10^{36}	5120
600	10^{78}	21000
1200	10^{168}	120000

©Petr Hanáček CLACRYPT Slide 69

Výhody EC

- EC je zvláště výhodné tam, kde
  - Je omezený výpočetní výkon (bezdrátová zařízení, čipové karty,…)
  - Je omezená plocha čipu integrovaného obvodu (bezdrátová zařízení, čipové karty,…)
  - Je potřebná vysoká rychlost
  - Je omezená přenosová rychlost (bezdrátová zařízení, čipové karty,…)
  - Je omezená kapacita pamětí (bezdrátová zařízení, čipové karty,…)

©Petr Hanáček CLACRYPT Slide 70

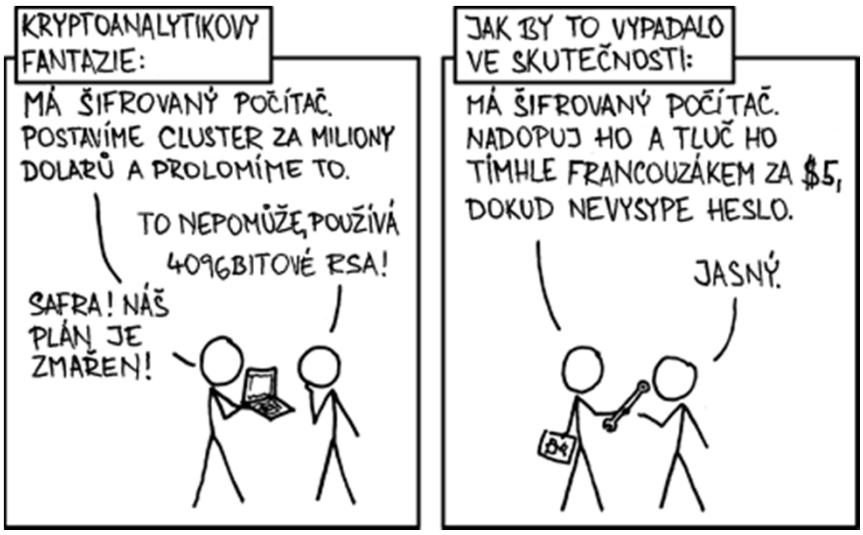
Porovnatelné délky klíčů

- **Předpokládá se, že žlutý sloupec je doporučován minimálně do roku 2010**

Velikost v bitech

Symetrický klíč	80	112	128	192	256
Hašovací funkce	160	224	256	384	512
FFC a IFC	1k	2k	3k	7.5k	15k
ECC	160	224	256	384	512

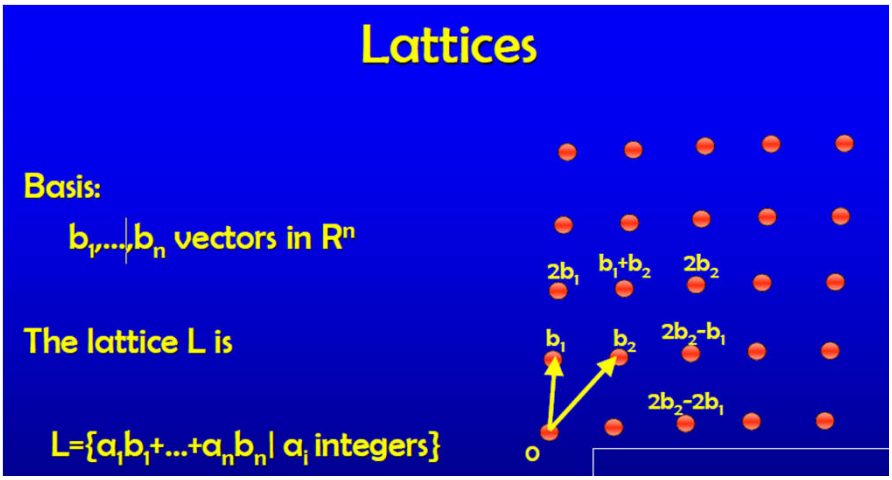
©Petr Hanáček CLACRYPT Slide 71



©Petr Hanáček CLACRYPT Slide 72

NTRU

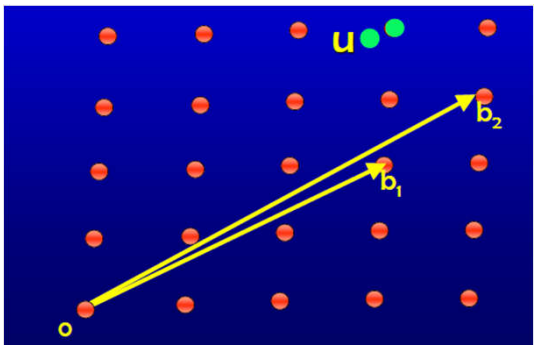
- Kryptografie založená na mřížce



©Petr Hanáček CLACRYPT Slide 73

Closest Vector Problem (CVP)

- CVP: Given a lattice and a target vector, find the closest lattice point
- • Seems very difficult; best algorithms take time 2n
- • However, checking if a point is in a lattice is easy

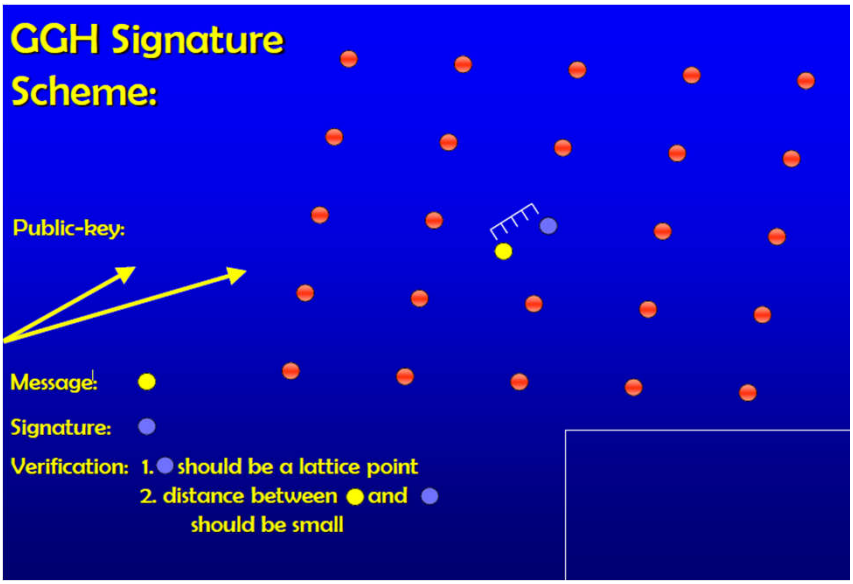


©Petr Hanáček CLACRYPT Slide 74

The GGH Signature Scheme [1997]

- Suggested in [GoldreichGoldwasserHalevi97]; no security proof
- • Idea: CVP is hard, but easy with good basis
- • The scheme:
  - – Key generation algorithm: choose a lattice with some good basis
    - » Private-key = good basis
    - » Public-key = bad basis
  - – Signing algorithm: given a message and a private key,
    - » Map message to a point in space
    - » Apply Babai's algorithm with good basis to obtain the signature
  - – Verification algorithm: given message+signature and a public key, verify that
    - » Signature is a lattice point, and
    - » Signature is close to the message

 CLACRYPT Slide 75



 CLACRYPT Slide 76

NTRU

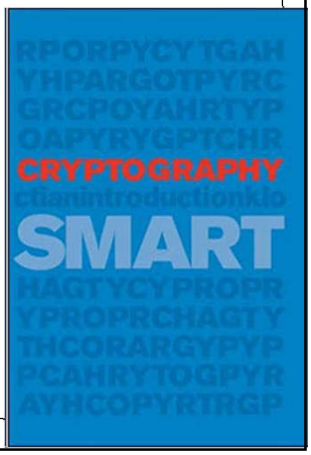
- http://crypto.biu.ac.il/winterschool2012/slides/slides-barilan8.pdf
- http://www.wisdom.weizmann.ac.il/~yaakovh/PKC2007/PrivatePapers/ntru-tutorial-slides.pdf

Hašovací funkce

Učebnice

4

- Nigel Smart: Cryptography - An Introduction, 3rd Edition,
  - Mcgraw-Hill College, 3rd Edition, 2013
  - ISBN-10: 0077099877
- Kapitoly
  - Kapitola 10 - Hash Functions and Message Authentication Codes
    - » Zajímavá je pro nás celá kapitola 10



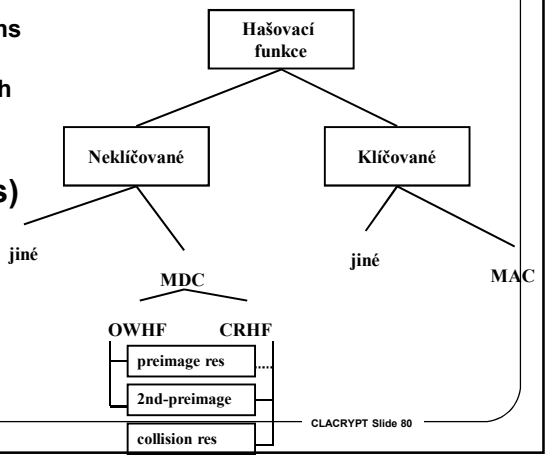
The third edition is now online. You may make copies and distribute the copies of the book as you see fit, as long as it is clearly marked as having been authored by N.P. Smart.

Učebnice je v dokumentovém skladu

©Petr Hanáček

Klasifikace

- MDC (manipulation detection codes) nebo MIC (message integrity codes), neklíčované
  - One-Way Hash Functions (OWHFs)
  - Collision Resistant Hash Functions (CRHFs)
- MAC (message authentication codes)
  - Zajišťují autentizaci a integritu
  - Klíčované



©Petr Hanáček

CLACRYPT Slide 80

Hašovací funkce

- Hašovací funkce, charakteristika zprávy, jednocestná funkce, message digest, digest, hash, hash function, one way function
- je to funkce F taková, že
  - je aplikovatelná na argument libovolné velikosti
  - její výstupní hodnota má konstantní délku (zpravidla 128, 160 nebo 256 bitů)
  - lze rychle spočíst F(x)
  - pro dané *y* je výpočetně nezvládnutelné nalézt takové *x*, aby platilo *F(x)=y* *(first preimage resistance)*
  - pro dané *x* je výpočetně nezvládnutelné nalézt takové *x'<>x*, aby platilo *F(x')=F(x)* *(second preimage resistance)*
  - je výpočetně nezvládnutelné nalézt takové *x' a x, x'<>x*, aby platilo *F(x')=F(x)* *(collision resistance)*
- implementace
  - MD2, MD4, MD5
  - SHS (Secure Hash Standard), SHA

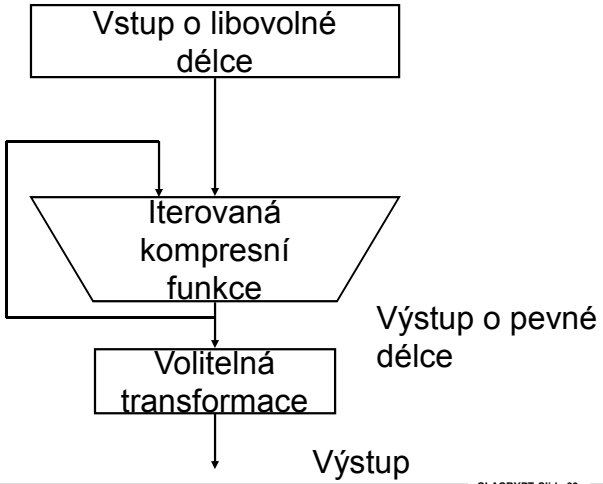
 CLACRYPT Slide 81

Vztah mezi vlastnostmi

- collision resistance⇒2nd preimage resistance
- collision resistance nezaručuje preimage resistance
- Pokud funkce h_k je MAC, pak h_k vzhledem k útoku se zvoleným textem (chosen-text attack) je:
  - 2nd preimage a collision resistant
  - preimage resistant

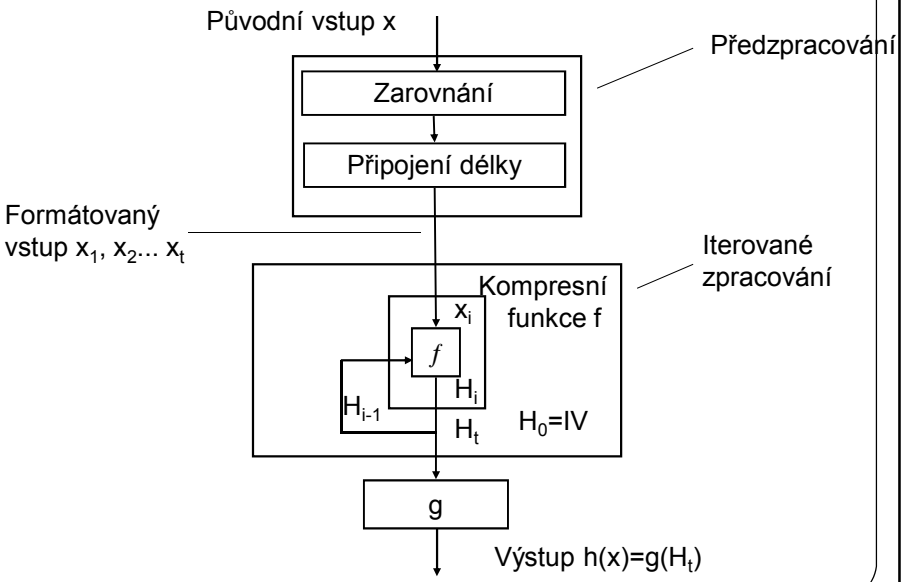
 CLACRYPT Slide 82

Obecný model iterované hašovací funkce



©Petr Hanáček CLACRYPT Slide 83

Detailní pohled



©Petr Hanáček CLACRYPT Slide 84

Merklova meta-metoda

- Jakákoli kompresní funkce f odolná proti kolizím se dá rozšířit na CRHF
- Merklova meta-metoda je efektivní způsob jak z f vytvořit CRHF
  - n bitový výstup, r bitová proměnná
  - Pokud existuje kolize pro h, pak to znamená, že vznikla kolize pro f v určitém kole i
  - Vložením délky bloku je zajištěno, že žádný vstup není prefizem jiného vstupu
    - » Merkle-Damgardovo zesílení

 CLACRYPT Slide 85

Zarovnání (Padding)

- Nejednoznačné zarovnání (Ambiguous Padding): připoj ke zprávě tolik nulových bitů, aby zpráva byla násobkem délky bloku
- Jednoznačné zarovnání (Unambiguous Padding)
  - Připoj ke zprávě 1
  - Proveď nejednoznačné zarovnání
  - Neintuitivní pro programátora

 CLACRYPT Slide 86

Bezpečnostní cíle

Typ funkce	Cíl návrhu	Ideální síla	Cíl útočníka
OWHF	preimage res; 2^{nd}-preimage res	2^n 2^n	Vytvořit preimage Vytvořit 2-preimage
CRHF	collision res	$2^{n/2}$	Vytvořit kolizi
MAC	key non-recovery; computation res	2^n Pf	Nalézt klíč Vytvořit nový MAC

* $Pf=\max(2^{-n}, 2^{-t})$ kde t je délka klíče

©Petr Hanáček CLACRYPT Slide 87

Základní útok

- **Základní útok na haš**
  - **n-bitový neklíčovaný haš má ideální bezpečnost, pokud splňuje požadavky na OWHF a CHRF**
- **Útok silou na klíč MAC (known-text attack),vyžaduje 2^t operací**
- **Uhodnutí MAC – vyžaduje 2^n operací**
- **Předvypočítání haše (memory-time tradeoff)**
- **Paralelizace 2nd-preimage**
- **Útoky na dlouhé zprávy pro 2nd-preimage. Pokud h je iterovaná funkce a nepoužívá se MD zesílení, pak 2nd-preimage může být nalezeno v čase $(2^n/s)+s$, v prostoru $n(s+\log s)$ bitů, pro $1 \le s \le \min(t, 2n/2)$**
  - **Narozeninový útok na mezivýsledky**

©Petr Hanáček CLACRYPT Slide 88

Birthday paradox

- The apparent paradox that, in a room of only 23 people, there is a 50 percent probability that at least two will have the same birthday. The "paradox" is that we have an even chance of success with just 23 of 365 possible days represented.
- Birthday paradox:
  $r1, \ldots, rn \in [0,1,\ldots,B]$ indep. random integers.
  When $n = 1.2\sqrt{B}$ then
  $Pr[\exists i \neq j : r_i = r_j] > \frac{1}{2}$
- msg-digest only 64 bits long $\Rightarrow$
  can find collision in 2^{32} tries.
- Typical digest size = 160 bits. (e.g. SHA-1)
  $\Rightarrow$collision time is 2^{80} tries.

©Petr Hanáček CLACRYPT Slide 89

Příklad Birthday Attack

- Předpokládejme hašovací funkci, která má n bitový výstup
- Útočník vytvoří dokument „přátelská dohoda“ a přibližně $2^{n/2+1}$ sémanticky ekvivalentních verzí
- Podobně útočník vytvoří dokument „nepřátelská dohoda“ a přibližně $2^{n/2+1}$ sémanticky ekvivalentních verzí
- S pravděpodobností ½ bude existovat verze „přátelské dohody“ a „nepřátelské dohody“, které budou mít stejný haš

©Petr Hanáček CLACRYPT Slide 90

Vyžadované délky

- OWHF n>=80
- CHRF n>=160 (birthday attack)
- MAC n>=64 s klíčem alespoň 64 bitů
  - Je vhodné omezit počet pokusů hádání

 CLACRYPT Slide 91

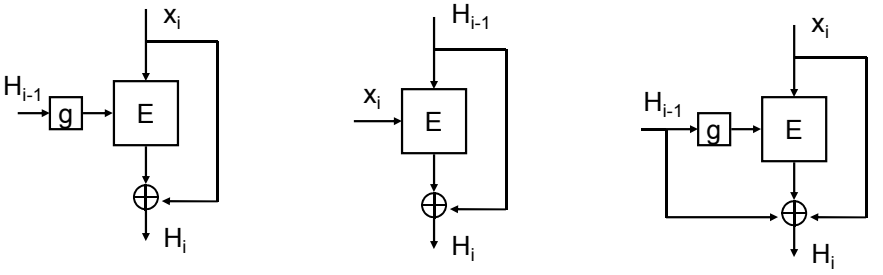
Některé hašovací algoritmy

	SHA-1	MD5 (MD4+)	RIPEMD-160
Velikost výstupu	160 bits	128 bits	160 bits
Základní velkost bloku	512 bits	512 bits	512 bits
Počet kroků	80 (4 rounds of 20)	64 (4 rounds of 16)	160 (5 paired rounds of 16)
Maximální velikost zprávy	2^{64}-1 bits	unlimited	unlimited

 CLACRYPT Slide 92

Hašovací funkce z blokové šifry

- Blokové šifry už existují (není třeba je navrhovat)
- Jednoduché (n bitů) nebo dvojité (2n bitů)
  - Jednoduché pro OWHF
  - Dvojité pro CHRF (obvykle n=64, pro odolnost proti kolizím potřebujem 128 bitů)

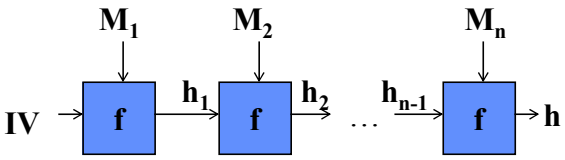


Matyas-Meyer-Oseas | Davies-Meyer | Miyaguchi-Preneel

©Petr Hanáček — CLACRYPT Slide 93

Konstrukce hašovacích algoritmů

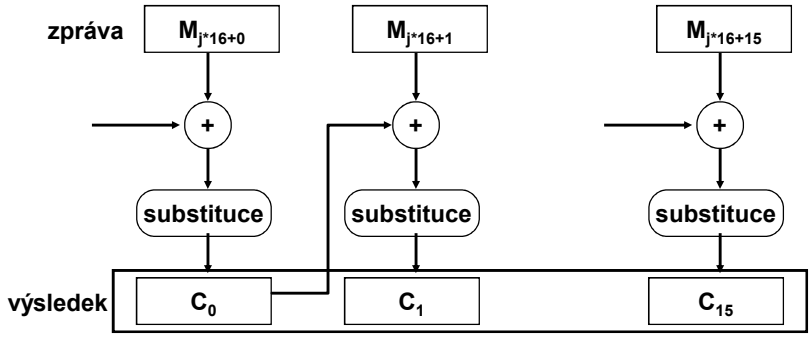
- Jsou obvykle založeny na kompresní funkci f, která pracuje nad bloky M



- Podobné blokovým šifrám v CBC režimu
- Vytvářejí hodnotu haše pro každý blok, která je závislá na hodnotě bloku a hodnotě haše předchozích bloků

©Petr Hanáček — CLACRYPT Slide 94

Algoritmus MD2



- MD2 dává 128-bitovou charakteristiku zprávy
- algoritmus je jednoduchý a poměrně velmi rychlý
- náhodná substituce je tabulka z číslic Ludolfova čísla, takže nemůže obsahovat skrytá vrátka

©Petr Hanáček CLACRYPT Slide 95

MD4

- Navrženo specielně jako hašovací funkce specielně pro softwarové implementace na 32-bitových strojích
- Startovací bod pro MD5, SHA-1 a RIPEMD
- 128 bitový výstup
  - Rozlomena jako CRHF, Dobbertin nalezl kolize pro smysluplné zprávy

©Petr Hanáček CLACRYPT Slide 96

RIPEMD-160

- **Kompresní funkce mapuje 21-slovní vstup (5-slovní stavová proměnná, 16-slovní blok zprávy, 32-bitová slova) na 5-slovní výstup**
- **Více kol než MD-4**
- **Bezpečnost porovnatelná s SHA-1**

 CLACRYPT Slide 97

MD5

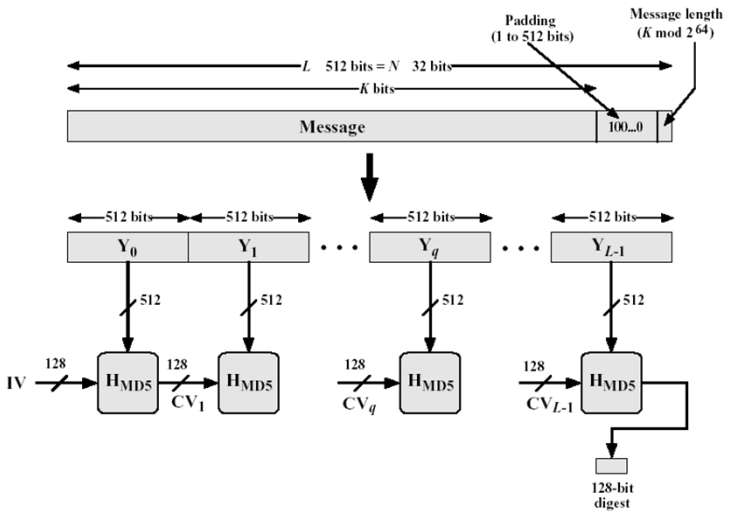
 CLACRYPT Slide 98

MD5

- Vyvinul ho Ron Rivest na MIT
- Zpracovává zprávu libovolné délky na haš o délce 128 bitů po blocích o délce 512 bitů
- Zpráva je doplněna na délku
  - » k = 448 mod 512
- Na konec zprávy je přidán 64bitový blok s délkou zprávy. Výsledná délka zprávy je násobkem 512 bitů.
- Detailní popis MD5 je v dokumentu RFC1321.
- Hans Dobbertin ukázal, že MD5 není odolné proti kolizím
- Používá se v IPSec a v jiných protokolech

 CLACRYPT Slide 99

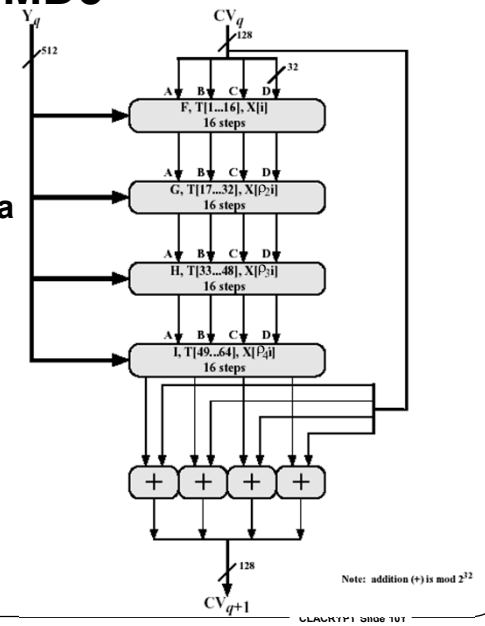
MD5



 CLACRYPT Slide 100

MD5

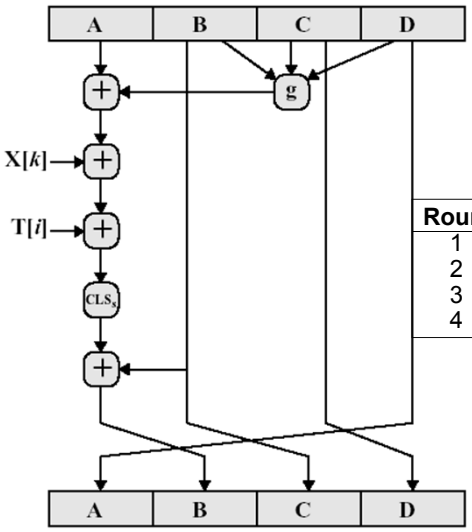
- Zpracování jednoho 512bitového bloku (kompresní funkce)
- F,G,H,I – funkce
- T – konstantní tabulka
- X – vstup Y



©Petr Hanáček

CLACRYPT Slide 101

MD5 – elementární operace



Round	Function g	g(b, c, d)
1	F(b, c, d)	$(b \wedge c) \vee (b' \wedge d)$
2	G(b, c, d)	$(b \wedge d) \vee (c \wedge d')$
3	H(b, c, d)	$b \oplus c \oplus d$
4	I(b c, d)	$c \oplus (b \vee d')$

©Petr Hanáček

CLACRYPT Slide 102

Principle of Most Surprise

- **https://news.ycombinator.com/item?id=9484757**

```
md5('240610708') == md5('QNKCDZO')
```


SHA

Secure Hash Algorithm (SHA)

- SHA byla vytvořena organizací NIST v roce 1993
- Podobná MD5
- Revidována v r. 1995 jako SHA-1
- Revidována v r. 2001 jako SHA-2
  - "SHA-256", "SHA-384", and "SHA-512"

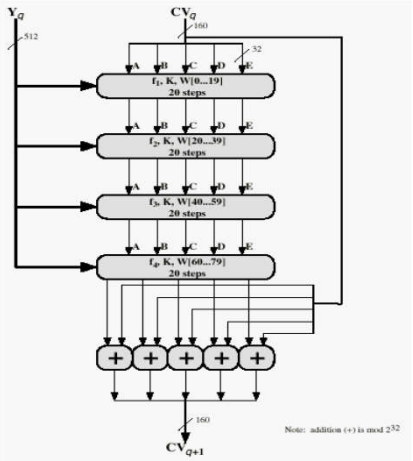
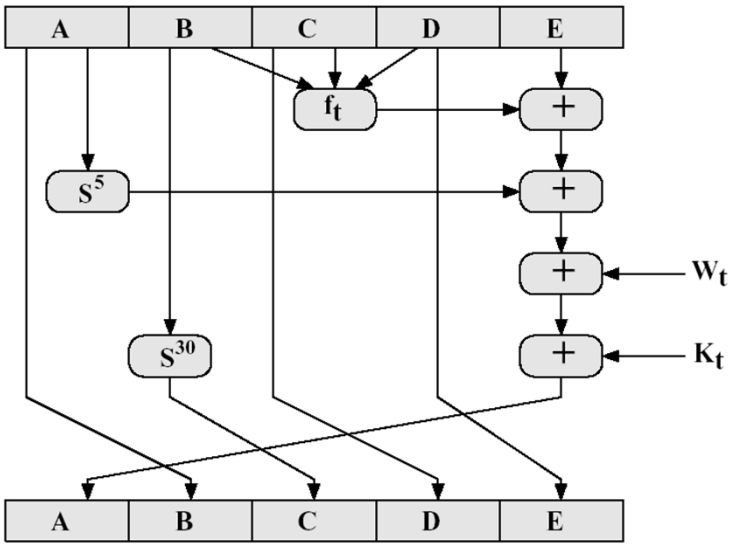


Figure 3.5 SHA-1 Processing of a Single 512-bit Block

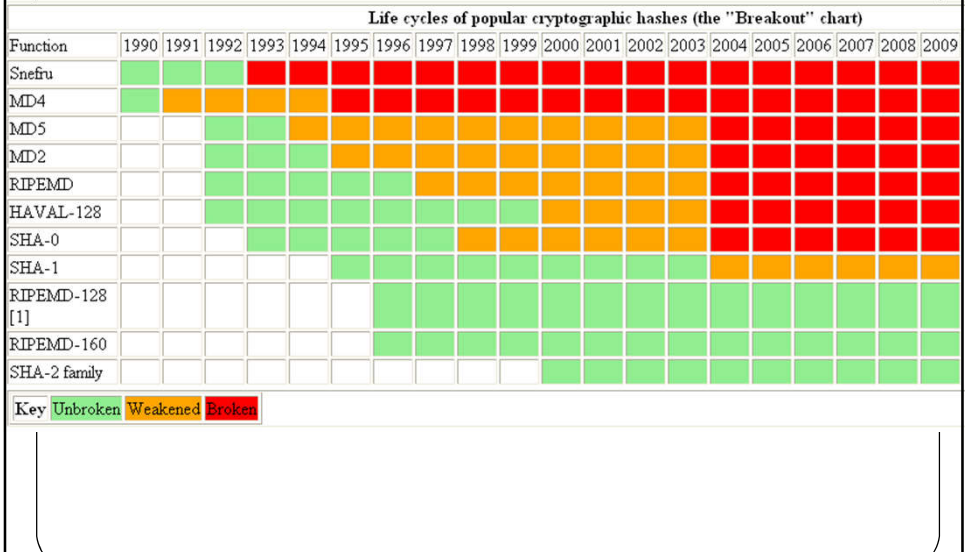
©Petr Hanáček CLACRYPT Slide 105

SHA – Elementární operace



©Petr Hanáček CLACRYPT Slide 106

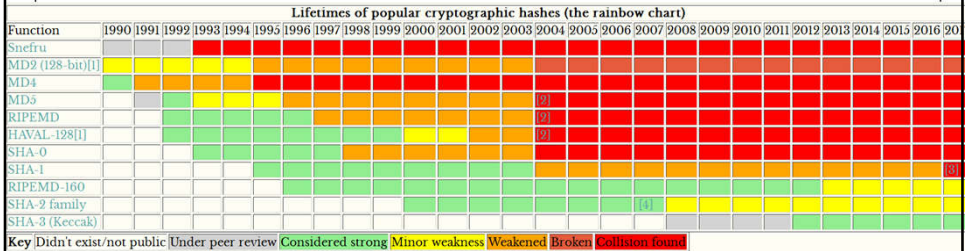
Breakout chart 2009



©Petr Hanáček CLACRYPT Slide 107

Rainbow chart 2017

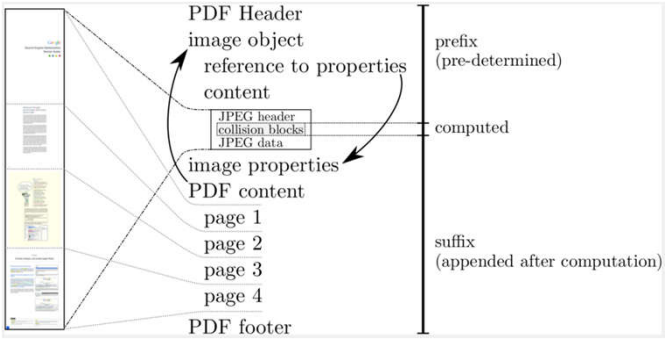
- **http://valerieaurora.org/hash.html**



©Petr Hanáček CLACRYPT Slide 108

23.2.2017 – zemřelo SHA-1

- **Announcing the first SHA1 collision**
  - **https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html**
  - **https://techcrunch.com/2017/02/23/security-researchers-announce-first-practical-sha-1-collision-attack/**
  - **https://shattered.it/static/shattered.pdf**



Soutěž o SHA-3

- **2005-2006: NIST přemýšlí o vyhlášení soutěže na SHA-3**
  - **MD5 a SHA-1 utrpěli těžké rány**
  - **SHA-2 je založen na stejných základech jako MD5 a SHA-1**
  - **Hledáme následníka SHA-2**
- **Říjen 2008: Deadline pro návrhy**
  - **Efektivnější než SHA-2**
  - **Délky výstupů: 224, 256, 384, 512 bitů**
  - **Bezpečnost: collision and (2nd) pre-image resistant**

Soutěž o SHA-3

- První kolo: Říjen 2008 až léto 2009
  - 64 návrhů, 51 přijato
  - 37 prezentováno na první konferenci kandidátů SHA-3 v Leuvenu, únor 2009
  - Mnoho z nich rozbito kryptoanalýzou
  - NIST zúžil výběr na 14 semifinalistů
- Druhé kolo: léto 2009 až podzim 2010
  - Analýzy prezentovány na druhé konferenci kandidátů SHA-3 v Santa Barbaře, srpen 2010
  - NIST zúžil výběr na 5 finalistů
- Třetí kolo: podzim 2010 až říjen 2012
  - Analýzy na třetí konferenci SHA-3 ve Washingtonu, březen 2012
- 2. říjen : NIST oznamuje, že vítězem SHA-3 se stal Keccak

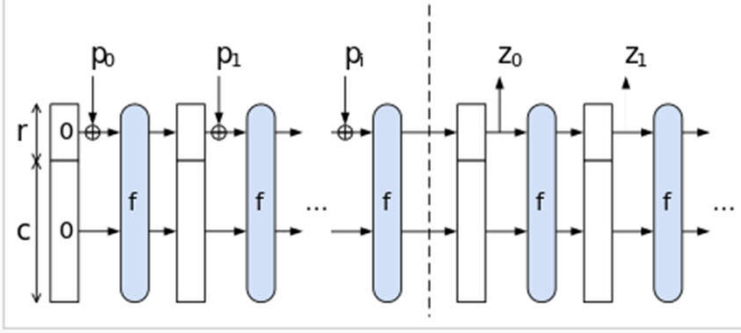
Keccak

- SHA-3 je kryptografická hašovací funkce, která byla určena v soutěži hledající nástupce starších funkcí SHA-1 a SHA-2 a organizované americkým NIST.
- Vítězná funkce byla do soutěže přihlášena pod svým původním jménem Keccak (výslovnost [kɛtʃak]), jejími autory jsou Guido Bertoni, Joan Daemen, Michaël Peeters a Gilles Van Assche
- Ostatní finalisty (i SHA-2) překonává v rychlosti v hardware.
- Při běhu na běžném procesoru Core 2 má rychlost zhruba 13 cyklů na bajt.
- Zcela odlišný princip od SHA-2, což znamená, že průlomový pokrok, který by ohrozil bezpečnost jedné z funkcí, pravděpodobně neohrozí druhou z nich.

•Wikipedia

SHA-3

- **Princip houby (sponge)**



The sponge construction for hash functions. p_i are input, z_i are hashed output. The unused "capacity" c should be twice the desired resistance to collision or preimage attacks.

https://en.wikipedia.org/wiki/SHA-3

MAC
Message Authentication Code

Vlastnosti MAC

- **MAC je rodina funkcí h_k (parametrizovaných tajným klíčem k)**
  - **Snadný výpočet (pokud je k známé)**
  - **Komprese, x má libovolnou délku, $h_k(x)$ má pevnou délku**
  - **Výpočetní bezpečnost, při znalosti páru $(x_i, h_k(x_i))$ je výpočetně nemožné spočíst novou dvojici $(x, h_k(x))$ pro nové $x \neq x_i$**

 CLACRYPT Slide 115

Message Authentication Code

- **Message Authentication Code (MAC)**
  - » **MAC=F(Message,Key)**

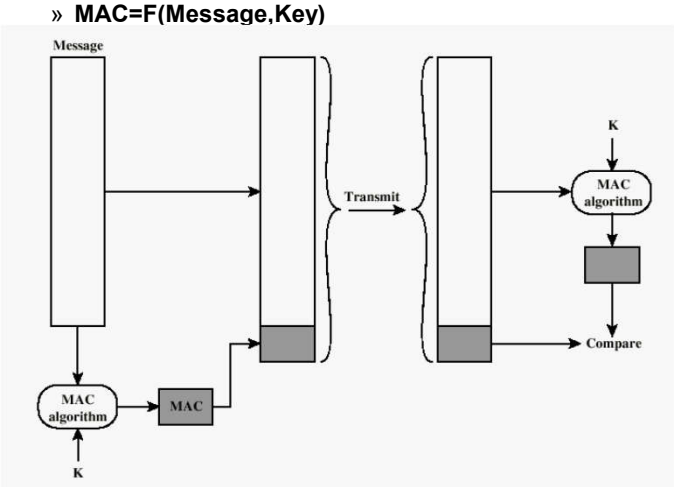
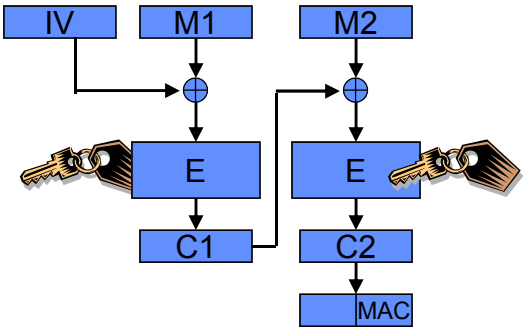


Figure 3.1 Message Authentication Using a Message Authentication Code (MAC)

 lide 116

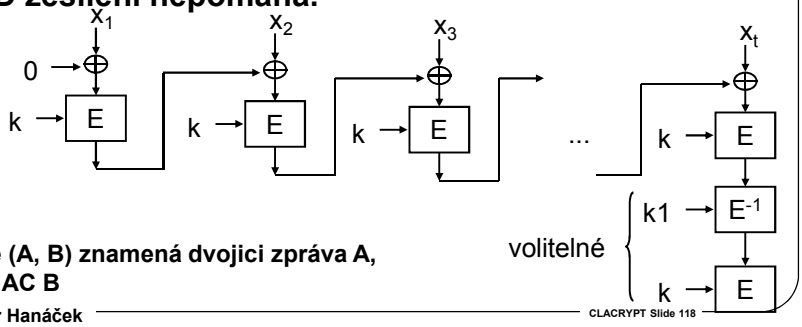
CBC MAC



- Typicky 32 bitů z posledního bloku (48, 64)
- Je to dost?

Bezpečnost CBC-MAC

- Volitelný krok má zabránit útoku chosen-text existential forgery bez ovlivnění předchozích kroků
- Existential forgery: základní CBC-MAC je bezpečný jenom pro zprávy z pevným počtem bloků. Jinak pokud máme dvojice (x1, H1) a (x2, H2) a můžeme požadovat ((x1 || z), M) pak je možné zkonstruovat novou zprávu (x2 || (H1⊕z⊕ H2), M) která je platná. MD zesílení nepomáhá.



Notace (A, B) znamená dvojici zpráva A, a její MAC B

MAC vytvořené z MDC

- Velmi rozšířená konstrukce (např. IPSec, SSL)
- Tři různé strategie
  - secret prefix
  - secret suffix
  - enveloping

 CLACRYPT Slide 119

Secret prefix

- Mějme MDC funkci *h* s kompresní funkcí *f*: H_0=IV, $H_i=f(H_{i-1}, x_i)$, $h(x)= H_t$
- Konstrukce: na začátek zprávy se přidá tajný klíč *k* a MAC je potom M=h(k||x)
- Je zde útok, kdy je možné na konec zprávy přidat *y* a spočíst h(k||x||y) ze znalosti h(k||x) bez znalosti *k* !!
- Ani MD zesílení nepomáhá (i délka x se dá zahrnout do zprávy)
- Stejně tak není bezpečná ani varianta, kdy *k* použijeme jako H_0

 CLACRYPT Slide 120

Secret suffix

- MAC hodnoty *x* se spočte jako M=h(x||k)
- Možnost narozeninového útoku, útočník, lterý může zvolit *x* může také vytvořit *x'* pro které h(x)=h(x') se složitostí $O(2^{n/2})$ bez ohledu na délku klíče k
- Útočník tedy může zkonstruovat dvojici (x',M)
- Metoda v podstatě vypočte haš a v konečné fázi ho „zašifruje“
- Není to dobrý způsob

Enveloping

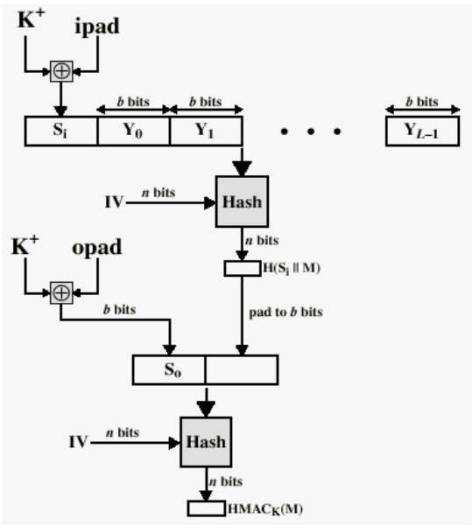
- $h_k(x)=h(k||p||x||k)$
- p je řetězec, použitý pro zarovnání klíče k na délku jednoho bloku
- Lepší než předchozí dvě metody, není to však nejlepší metoda
- Základ pro algoritmus HMAC

Hash Function MAC (HMAC)

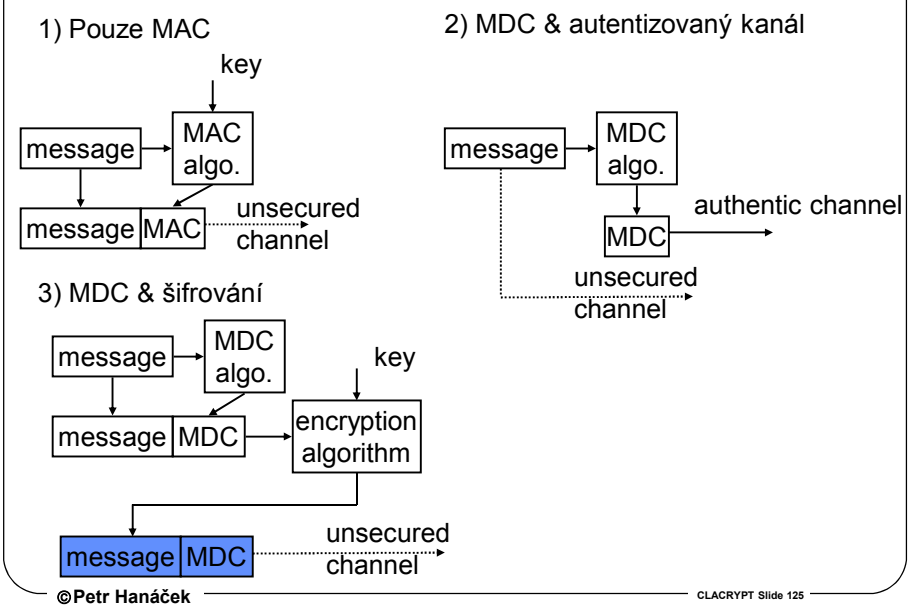
- „Klíčovaný haš“
- Myšlenka: vytvořit MAC z hašovací funkce
  - Dodání klíče
    - » „Přihašování klíče“
- Použití:
  - IPsec
  - Transport Layer Security (TLS)

HMAC

- Spočte se H1 jako haš konkatenace M a K1
- Pro zabránění útoku “dodatečný blok”, se spočte H2 jako haš konkatenace H1 a K2
- K1 a K2 používají polovinu bitů klíče K
- Vymaskování bitů:
  - K^+ = K doplněný nulami
  - Ipad = 00110110 x b/8
  - Opad = 01011100 x b/8



Metody pro integritu dat



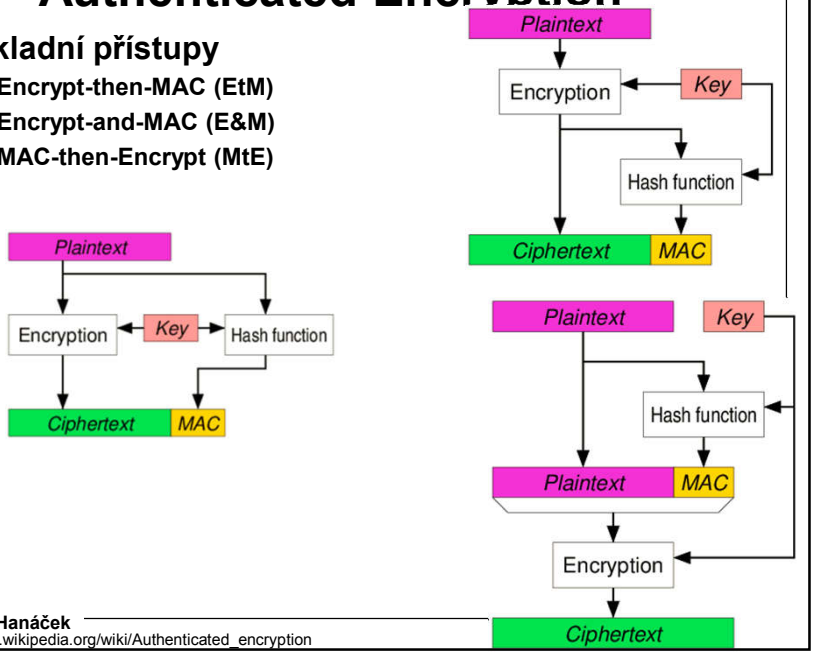
©Petr Hanáček CLACRYPT Slide 125

AE
Authenticated Encryption

©Petr Hanáček CLACRYPT Slide 126

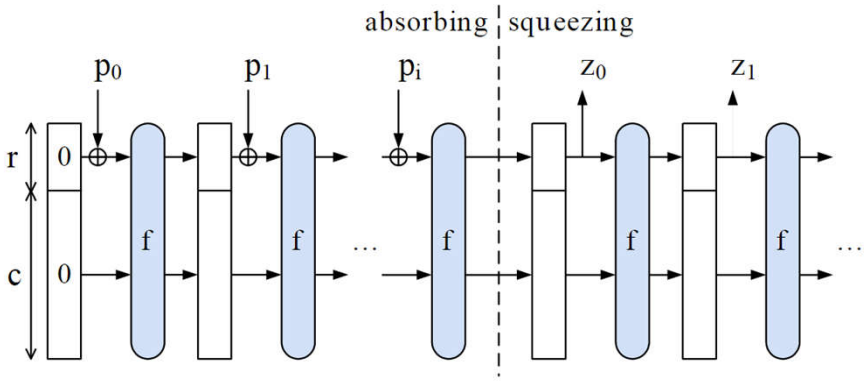
Authenticated Encryption

- **Základní přístupy**
  - **Encrypt-then-MAC (EtM)**
  - **Encrypt-and-MAC (E&M)**
  - **MAC-then-Encrypt (MtE)**



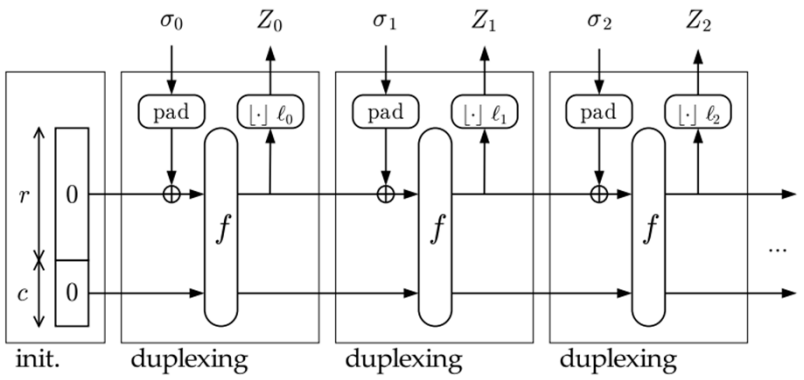
©Petr Hanáček
https://en.wikipedia.org/wiki/Authenticated_encryption

Sponge



©Petr Hanáček
CLACRYPT Slide 128
https://en.wikipedia.org/wiki/Sponge_function

Sponge duplexing



KONEC

CHAPTER 10

Hash Functions and Message Authentication Codes

Chapter Goals

- To understand the properties of cryptographic hash functions.
- To understand how existing deployed hash functions work.
- To examine the workings of message authentication codes.

1. Introduction

In many situations we do not wish to protect the confidentiality of information, we simply wish to ensure the integrity of information. That is we want to guarantee that data has not been tampered with. In this chapter we look at two mechanisms for this, the first using cryptographic hash functions is for when we want to guarantee integrity of information after the application of the function. A cryptographic hash function is usually used as a component of another scheme, since the integrity is not bound to any entity. The other mechanism we look at is the use of a message authentication code. These act like a keyed version of a hash function, they are a symmetric key technique which enables the holders of a symmetric key to agree that only they could have produced the authentication code on a given message.

Hash functions can also be considered as a special type of manipulation detection code, or MDC. For example a hash function can be used to protect the integrity of a large file, as used in some virus protection products. The hash value of the file contents is computed and then either stored in a secure place (e.g. on a floppy in a safe) or the hash value is put in a file of similar values which is then digitally signed to stop future tampering.

Both hash functions and MACs will be used extensively later in other schemes. In particular cryptographic hash functions will be used to compress large messages down to smaller ones to enable efficient digital signature algorithms. Another use of hash functions is to produce, in a deterministic manner, random data from given values. We shall see this application when we build elaborate and "provably secure" encryption schemes later on in the book.

2. Hash Functions

A cryptographic hash function h is a function which takes arbitrary length bit strings as input and produces a fixed length bit string as output, the output is often called a hashcode or hash value. Hash functions are used a lot in computer science, but the crucial difference between a standard hash function and a cryptographic hash function is that a cryptographic hash function should at least have the property of being one-way. In other words given any string y from the range of h, it should be computationally infeasible to find any value x in the domain of h such that

$$h(x) = y.$$

Another way to describe a hash function which has the one-way property is that it is preimage resistant. Given a hash function which produces outputs of n bits, we would like a function for which finding preimages requires $O(2^n)$ time.

In practice we need something more than the one-way property. A hash function is called collision resistant if it is infeasible to find two distinct values x and x' such that

$$h(x) = h(x').$$

It is harder to construct collision resistant hash functions than one-way hash functions due to the birthday paradox. To find a collision of a hash function f, we can keep computing

$$f(x_1), f(x_2), f(x_3), \ldots$$

until we get a collision. If the function has an output size of n bits then we expect to find a collision after $O(2^{n/2})$ iterations. This should be compared with the number of steps needed to find a preimage, which should be $O(2^n)$ for a well-designed hash function. Hence to achieve a security level of 80 bits for a collision resistant hash function we need roughly 160 bits of output.

But still that is not enough; a cryptographic hash function should also be second preimage resistant. This is the property that given m it should be hard to find an $m' \neq m$ with $h(m') = h(m)$. Whilst this may look like collision resistance, it is actually related more to preimage resistance. In particular a cryptographic hash function with n-bit outputs should require $O(2^n)$ operations before one can find a second preimage.

In summary a cryptographic hash function needs to satisfy the following three properties:

(1) **Preimage Resistant:** It should be hard to find a message with a given hash value.
(2) **Collision Resistant:** It should be hard to find two messages with the same hash value.
(3) **Second Preimage Resistant:** Given one message it should be hard to find another message with the same hash value.

But how are these properties related. We can relate these properties using reductions.

LEMMA 10.1. *Assuming a function is preimage resistant* for every element of the range of h *is a weaker assumption than assuming it either collision resistant or second preimage resistant.*

PROOF. Suppose h is a function and let $\mathcal{O}$ denote an oracle which on input of y finds an x such that $h(x) = y$, i.e. $\mathcal{O}$ is an oracle which breaks the preimage resistance of the function h.

Using $\mathcal{O}$ we can then find a collision in h by pulling x at random and then computing $y = h(x)$. Passing y to the oracle $\mathcal{O}$ will produce a value x' such that $y = h(x')$. Since h is assumed to have infinite domain, it is unlikely that we have $x = x'$. Hence, we have found a collision in h.

A similar argument applies to breaking the second preimage resistance of h. □

However, one can construct hash functions which are collision resistant but are not one-way for some of the range of h. As an example, let $g(x)$ denote a collision resistant hash function with outputs of bit length n. Now define a new hash function $h(x)$ with output size $n+1$ bits as follows:

$$h(x) = \begin{cases} 0\|x & \text{If } |x| = n, \\ 1\|g(x) & \text{Otherwise.} \end{cases}$$

The function $h(x)$ is clearly collision resistant, as we have assumed $g(x)$ is collision resistant. But the function $h(x)$ is not preimage resistant as one can invert it on any value in the range which starts with a zero bit. So even though we can invert the function $h(x)$ on some of its input we are unable to find collisions.

LEMMA 10.2. *Assuming a function is second preimage resistant is a weaker assumption than assuming it is collision resistant.*

PROOF. Assume we are given an oracle $\mathcal{O}$ which on input of x will find x' such that $x \neq x'$ and $h(x) = h(x')$. We can clearly use $\mathcal{O}$ to find a collision in h by choosing x at random. □

3. Designing Hash Functions

To be effectively collision free a hash value should be at least 128 bits long, for applications with low security, but preferably its output should be 160 bits long. However, the input size should be bit strings of (virtually) infinite length. In practice designing functions of infinite domain is hard, hence usually one builds a so called compression function which maps bits strings of length s into bit strings of length n, for $s > n$, and then chains this in some way so as to produce a function on an infinite domain. We have seen such a situation before when we considered modes of operation of block ciphers.

We first discuss the most famous chaining method, namely the Merkle–Damgård construction, and then we go on to discuss designs for the compression function.

3.1. Merkle–Damgård Construction. Suppose f is a compression function from s bits to n bits, with $s > n$, which is believed to be collision resistant. We wish to use f to construct a hash function h which takes arbitrary length inputs, and which produces hash codes of n bits in length. The resulting hash function should be collision resistant. The standard way of doing this is to use the Merkle–Damgård construction described in Algorithm 10.1.

Algorithm 10.1: Merkle–Damgård Construction

$l = s - n$
Pad the input message m with zeros so that it is a multiple of l bits in length
Divide the input m into t blocks of l bits long, $m_1, \ldots, m_t$
Set H to be some fixed bit string of length n.
for $i = 1$ **to** t **do**
  $H = f(H \| m_i)$
end
return (H)

In this algorithm the variable H is usually called the *internal state* of the hash function. At each iteration this internal state is updated, by taking the current state and the next message block and applying the compression function. At the end the internal state is output as the result of the hash function.

Algorithm 10.1 describes the basic Merkle–Damgård construction, however it is almost always used with so called *length strengthening*. In this variant the input message is preprocessed by first padding with zero bits to obtain a message which has length a multiple of l bits. Then a final block of l bits is added which encodes the original length of the unpadded message in bits. This means that the construction is limited to hashing messages with length less than 2^l bits.

To see why the strengthening is needed consider a "baby" compression function f which maps bit strings of length 8 into bit strings of length 4 and then apply it to the two messages

$$m_1 = 0b0, \ m_2 = 0b00.$$

Whilst the first message is one bit long and the second message is two bits long, the output of the basic Merkle–Damgård construction will be

$$h(m_1) = f(0b00000000) = h(m_2),$$

i.e. we obtain a collision. However, with the strengthened version we obtain the following hash values in our baby example

$$\begin{aligned} h(m_1) &= f(f(0b00000000) \| 0b0001), \\ h(m_2) &= f(f(0b00000000) \| 0b0010). \end{aligned}$$

These last two values will be different unless we just happen to have found a collision in f.

Another form of length strengthening is to add a single one bit onto the data to signal the end of a message, pad with zeros, and then apply the hash function. Our baby example in this case would become

$$\begin{aligned} h(m_1) &= f(0b01000000), \\ h(m_2) &= f(0b00100000). \end{aligned}$$

Yet another form is to combine this with the previous form of length strengthening, so as to obtain

$$\begin{aligned} h(m_1) &= f(f(0b01000000)\|0b0001), \\ h(m_2) &= f(f(0b00100000)\|0b0010). \end{aligned}$$

3.2. The MD4 Family. A basic design principle when designing a compression function is that its output should produce an avalanche affect, in other words a small change in the input produces a large and unpredictable change in the output. This is needed so that a signature on a cheque for 30 pounds cannot be altered into a signature on a cheque for 30 000 pounds, or vice versa. This design principle is typified in the MD4 family which we shall now describe.

Several hash functions are widely used, they are all iterative in nature. The three most widely deployed are MD5, RIPEMD-160 and SHA-1. The MD5 algorithm produces outputs of 128 bits in size, whilst RIPEMD-160 and SHA-1 both produce outputs of 160 bits in length. Recently NIST has proposed a new set of hash functions called SHA-256, SHA-384 and SHA-512 having outputs of 256, 384 and 512 bits respectively, collectively these algorithms are called SHA-2. All of these hash functions are derived from an earlier simpler algorithm called MD4.

The seven main algorithms in the MD4 family are

- **MD4**: This has 3 rounds of 16 steps and an output bitlength of 128 bits.
- **MD5**: This has 4 rounds of 16 steps and an output bitlength of 128 bits.
- **SHA-1**: This has 4 rounds of 20 steps and an output bitlength of 160 bits.
- **RIPEMD-160**: This has 5 rounds of 16 steps and an output bitlength of 160 bits.
- **SHA-256**: This has 64 rounds of single steps and an output bitlength of 256 bits.
- **SHA-384**: This is identical to SHA-512 except the output is truncated to 384 bits.
- **SHA-512**: This has 80 rounds of single steps and an output bitlength of 512 bits.

We discuss MD4 and SHA-1 in detail, the others are just more complicated versions of MD4, which we leave to the interested reader to look up in the literature.

In recent years a number of weaknesses have been found in almost all of the early hash functions in the MD4 family, for example MD4, MD5 and SHA-1. Hence, it is wise to move all application to use the SHA-2 algorithms.

3.3. MD4. In MD4 there are three bit-wise functions of three 32-bit variables

$$\begin{aligned} f(u,v,w) &= (u \wedge v) \vee ((\neg u) \wedge w), \\ g(u,v,w) &= (u \wedge v) \vee (u \wedge w) \vee (v \wedge w), \\ h(u,v,w) &= u \oplus v \oplus w. \end{aligned}$$

Throughout the algorithm we maintain a current hash state

$$(H_1, H_2, H_3, H_4)$$

of four 32-bit values initialized with a fixed initial value,

$$\begin{aligned} H_1 &= \texttt{0x67452301}, \\ H_2 &= \texttt{0xEFCDAB89}, \\ H_3 &= \texttt{0x98BADCFE}, \\ H_4 &= \texttt{0x10325476}. \end{aligned}$$

There are various fixed constants (y_i, z_i, s_i), which depend on each round. We have

$$y_j = \begin{cases} 0 & 0 \leq j \leq 15, \\ \texttt{0x5A827999} & 16 \leq j \leq 31, \\ \texttt{0x6ED9EBA1} & 32 \leq j \leq 47, \end{cases}$$

and the values of z_i and s_i are given by following arrays,

$$\begin{aligned} z_{0\ldots15} &= [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15], \\ z_{16\ldots31} &= [0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15], \\ z_{32\ldots47} &= [0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15], \\ s_{0\ldots15} &= [3,7,11,19,3,7,11,19,3,7,11,19,3,7,11,19], \\ s_{16\ldots31} &= [3,5,9,13,3,5,9,13,3,5,9,13,3,5,9,13], \\ s_{32\ldots47} &= [3,9,11,15,3,9,11,15,3,9,11,15,3,9,11,15]. \end{aligned}$$

The data stream is loaded 16 words at a time into X_j for $0 \leq j < 16$. The length strengthening method used is to first append a one bit to the message, to signal its end and then to pad with zeros to a multiple of the block length. Finally the number of bits of the message is added as a seperate final block.

We then execute the steps in Algorithm 10.2 for each 16 words entered from the data stream.

Algorithm 10.2: MD4 Overview

$(A, B, C, D) = (H_1, H_2, H_3, H_4)$
Execute Round 1
Execute Round 2
Execute Round 3
$(H_1, H_2, H_3, H_4) = (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$

After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4.$$

The details of the rounds are given by Algorithm 10.3 where $\lll$ denotes a bit-wise rotate to the left:

3.4. SHA-1. We use the same bit-wise functions f, g and h as in MD4. For SHA-1 the internal state of the algorithm is a set of five, rather than four, 32-bit values

$$(H_1, H_2, H_3, H_4, H_5).$$

These are assigned with the initial values

$$\begin{aligned} H_1 &= \texttt{0x67452301}, \\ H_2 &= \texttt{0xEFCDAB89}, \\ H_3 &= \texttt{0x98BADCFE}, \\ H_4 &= \texttt{0x10325476}, \\ H_5 &= \texttt{0xC3D2E1F0}, \end{aligned}$$

Algorithm 10.3: Description of the MD4 round functions

Round 1
for $j = 0$ **to** 15 **do**
  $t = A + f(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end
Round 2
for $j = 16$ **to** 31 **do**
  $t = A + g(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end
Round 3
for $j = 32$ **to** 47 **do**
  $t = A + h(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end

We now only define four round constants y_1, y_2, y_3, y_4 via

$$y1 = \texttt{0x5A827999},$$
$$y2 = \texttt{0x6ED9EBA1},$$
$$y3 = \texttt{0x8F1BBCDC},$$
$$y4 = \texttt{0xCA62C1D6}.$$

The data stream is loaded 16 words at a time into X_j for $0 \leq j < 16$, although note the internals of the algorithm uses an expanded version of X_j with indices from 0 to 79. The length strengthening method used is to first append a one bit to the message, to signal its end and then to pad with zeros to a multiple of the block length. Finally the number of bits of the message is added as a seperate final block.

We then execute the steps in Algorithm 10.4 for each 16 words entered from the data stream. The details of the rounds are given by Algorithm 10.5.

Algorithm 10.4: SHA-1 Overview

$(A, B, C, D, E) = (H_1, H_2, H_3, H_4, H_5)$
/* Expansion */
for $j = 16$ **to** 79 **do**
  $X_j = ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \lll 1)$
end
Execute Round 1
Execute Round 2
Execute Round 3
Execute Round 4
$(H_1, H_2, H_3, H_4, H_5) = (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E)$

Note the one bit left rotation in the expansion step, an earlier algorithm called SHA(now called SHA-0) was initially proposed by NIST which did not include this one bit rotation. This was however soon replaced by the new algorithm SHA-1. It turns out that this single one bit rotation improves the security of the resulting hash function quite a lot.

Algorithm 10.5: Description of the SHA-1 round functions

Round 1
for $j = 0$ **to** 19 **do**
  $t = (A \lll 5) + f(B, C, D) + E + X_j + y_1$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 2
for $j = 20$ **to** 39 **do**
  $t = (A \lll 5) + h(B, C, D) + E + X_j + y_2$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 3
for $j = 40$ **to** 59 **do**
  $t = (A \lll 5) + g(B, C, D) + E + X_j + y_3$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 4
for $j = 60$ **to** 79 **do**
  $t = (A \lll 5) + h(B, C, D) + E + X_j + y_4$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end

After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4, H_5.$$

3.5. Hash Functions and Block Ciphers. One can also make a hash function out of an n-bit block cipher, E_K. There are a number of ways of doing this, all of which make use of a constant public initial value IV. Some of the schemes also make use of a function g which maps n-bit inputs to keys.

We first pad the message to be hashed and divide it into blocks

$$x_0, x_1, \ldots, x_t,$$

of size either the block size or key size of the underlying block cipher, the exact choice of size depending on the exact definition of the hash function being created. The output hash value is then the final value of H_i in the following iteration

$$H_0 = IV,$$
$$H_i = f(x_i, H_{i-1}).$$

The exact definition of the function f depends on the scheme being used. We present just three, although others are possible.

- **Matyas–Meyer–Oseas hash**
$$f(x_i, H_{i-1}) = E_{g(H_{i-1})}(x_i) \oplus x_i.$$
- **Davies–Meyer hash**
$$f(x_i, H_{i-1}) = E_{x_i}(H_{i-1}) \oplus H_{i-1}.$$
- **Miyaguchi–Preneel hash**
$$f(x_i, H_{i-1}) = E_{g(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1}.$$

4. Message Authentication Codes

Given a message and its hash code, as output by a cryptographic hash function, ensures that data has not been tampered with between the execution of the hash function and its verification, by recomputing the hash. However, using a hash function in this way requires the hash code itself to be protected in some way, by for example a digital signature, as otherwise the hash code itself could be tampered with.

To avoid this problem one can use a form of keyed hash function called a message authentication code, or MAC. This is a symmetric key algorithm in that the person creating the code and the person verifying it both require the knowledge of a shared secret.

Suppose two parties, who share a secret key, wish to ensure that data transmitted between them has not been tampered with. They can then use the shared secret key and a keyed algorithm to produce a check-value, or MAC, which is sent with the data. In symbols we compute

$$\text{code} = MAC_k(m)$$

where

- MAC is the check function,
- k is the secret key,
- m is the message.

Note we do not assume that the message is secret, we are trying to protect data integrity and not confidentiality. If we wish our message to remain confidential then we should encrypt it before applying the MAC. After performing the encryption and computing the MAC, the user transmits

$$e_{k_1}(m) \| MAC_{k_2}\left(e_{k_1}(m)\right).$$

This is a form of encryption called a data encapsulation mechanism, or DEM for short. Note, that different keys are used for the encryption and the MAC part of the message and that the MAC is applied to the ciphertext and not the message.

Before we proceed on how to construct MAC functions it is worth pausing to think about what security properties we require. We would like that only people who know the shared secret are able to both produce new MACs or verify existing MACs. In particular it should be hard given a MAC on a message to produce a MAC on a new message.

4.1. Producing MACs from hash functions.

A collision-free cryptographic hash function can also be used as the basis of a MAC. The first idea one comes up with to construct such a MAC is to concatenate the key with the message and then apply the hash function. For example

$$MAC_k(M) = h(k \| M).$$

However, this is not a good idea since almost all hash functions are created using methods like the Merkle–Damgård construction. This allows us to attack such a MAC as follows: We assume that first that the non-length strengthed Merkle–Damgård construction is used with compression function f. Suppose one obtains the MAC c_1 on the t block message m_1

$$c_1 = MAC_k(m_1) = h(k \| m_1)$$

We can then, without knowledge of k compute the MAC c_2 on the $t+1$ block message $m_1 \| m_2$ for any m_2 of one block in length, via

$$\begin{aligned} c_2 &= MAC_k(m_1 \| m_2) \\ &= f(c_1 \| m_2). \end{aligned}$$

Clearly this attack can be extended to a appending an m_2 of arbitrary length. Hence, we can also apply it to the length strengthed version. If we let m_1 denote a t block message and let b denote

the block which encodes the bit length of m_1 and we let m_2 denote an arbitrary new block, then from the MAC of the message m_1 one can obtain the MAC of the message

$$m_1 \| b \| m_2.$$

Having worked out that prepending a key to a message does not give a secure MAC, one might be led to try appending the key after the message as in

$$MAC_k(M) = h(M \| k).$$

Again we now can make use of the Merkle–Damgård construction to produce an attack. We first, without knowledge of k, find via a birthday attack on the hash function h two equal length messages m_1 and m_2 which hash to the same values:

$$h(m_1) = h(m_2).$$

We now try to obtain the legitimate MAC c_1 on the message m_1. From, this we can deduce the MAC on the message m_2 via

$$\begin{aligned} MAC_k(m_2) &= h(m_2 \| k) \\ &= f\left(h(m_2) \| k\right) \\ &= f\left(h(m_1) \| k\right) \\ &= h(m_1 \| k) \\ &= MAC_k(m_1) \\ &= c_1. \end{aligned}$$

assuming k is a single block in length and the non-length strengthened version is used. Both of these assumptions can be relaxed, the details of which we leave to the reader.

To produce a secure MAC from a hash function one needs to be a little more clever. A MAC, called HMAC, occurring in a number of standards documents works as follows:

$$HMAC = h(k \| p_1 \| h(k \| p_2 \| M)),$$

where p_1 and p_2 are strings used to pad out the input to the hash function to a full block.

4.2. Producing MACs from block ciphers. Apart from ensuring the confidentiality of messages, block ciphers can also be used to protect the integrity of data. There are various types of MAC schemes based on block ciphers, but the best known and most widely used by far are the CBC-MACs. These are generated by a block cipher in CBC Mode. CBC-MACs are the subject of various international standards dating back to the early 1980s. These early standards specify the use of DES in CBC mode to produce a MAC, although one could really use any block cipher in place of DES.

Using an n-bit block cipher to give an m-bit MAC, where $m \leq n$, is done as follows:

- The data is padded to form a series of n-bit blocks.
- The blocks are encrypted using the block cipher in CBC Mode.
- Take the final block as the MAC, after an optional postprocessing stage and truncation (if $m < n$).

Hence, if the n-bit data blocks are

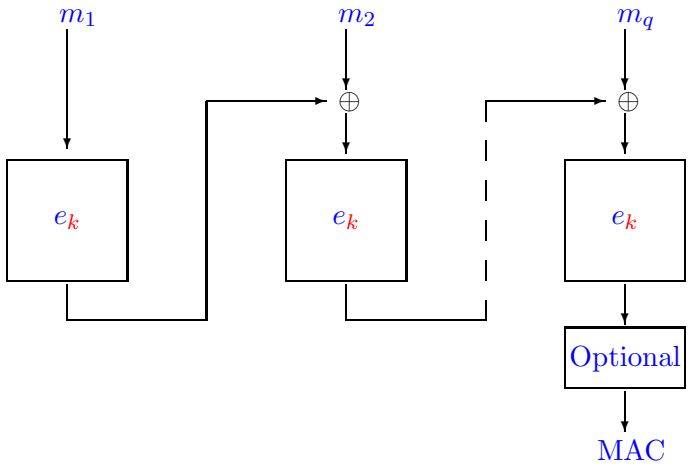
$$m_1, m_2, \ldots, m_q$$

then the MAC is computed by first setting $I_1 = m_1$ and $O_1 = e_k(I_1)$ and then performing the following for $i = 2, 3, \ldots, q$

$$I_i = m_i \oplus O_{i-1},$$
$$O_i = e_k(I_i).$$

The final value O_q is then subject to an optional processing stage. The result is then truncated to m bits to give the final MAC. This is all summarized in Fig. 1.

FIGURE 1. CBC-MAC: Flow diagram



Just as with hash functions one needs to worry about how one pads the message before applying the CBC-MAC. The three main padding methods proposed in the standards, are as follows, and are equivalent to those already considered for hash functions:

- **Method 1:** Add as many zeros as necessary to make a whole number of blocks. This method has a number of problems associated to it as it does not allow the detection of the addition or deletion of trailing zeros, unless the message length is known.
- **Method 2:** Add a single one to the message followed by as many zeros as necessary to make a whole number of blocks. The addition of the extra bit is used to signal the end of the message, in case the message ends with a string of zeros.
- **Method 3:** As method one but also add an extra block containing the length of the unpadded message.

Before we look at the "optional" post-processing steps let us first see what happens if no post-processing occurs. We first look at an attack which uses padding method one. Suppose we have a MAC M on a message

$$m_1, m_2, \ldots, m_q,$$

consisting of a whole number of blocks. Then one can the MAC M is also the MAC of the double length message

$$m_1, m_2, \ldots, m_q, M \oplus m_1, m_2, m_3, \ldots, m_q.$$

To see this notice that the input to the $(q+1)$'st block cipher envocation is equal to the value of the MAC on the original message, namely M, xor'd with the $(q+1)$'st block of the new message, namely $M \oplus m_1$. Thus the input to the $(q+1)$'st cipher envocation is equalk to m_1, and so the MAC on the double length message is also equal to M.

One could suspect that if you used padding method three above then attacks would be impossible. Let b denote the block length of the cipher and let $\mathbb{P}(n)$ denote the encoding within a block

of the number n. To MAC a single block message m_1 one then computes

$$M_1 = e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right).$$

Suppose one obtains the MAC's M_1 and M_2 on the single block messages m_1 and m_2. Then one requests the MAC on the three block message

$$m_1, \mathbb{P}(b), m_3$$

for some new block m_3. Suppose the recieved MAC is then equal to M_3, i.e.

$$M_3 = e_k\left(e_k\left(e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right) \oplus m_3\right) \oplus \mathbb{P}(3b)\right).$$

Now also consider the MAC on the three block message

$$m_2, \mathbb{P}(b), m_3 \oplus M_1 \oplus M_2.$$

This MAC is equal to M_3', where

$$\begin{aligned}
M_3' &= e_k\left(e_k\left(e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right) \oplus m_3 \oplus M_1 \oplus M_2\right) \oplus \mathbb{P}(3b)\right) \\
&= e_k\left(e_k\left(e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right) \oplus m_3 \oplus e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right) \oplus e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right)\right)\right. \\
&\quad \left.\oplus \mathbb{P}(3b)\right) \\
&= e_k\left(e_k\left(m_3 \oplus e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right)\right) \oplus \mathbb{P}(3b)\right) \\
&= e_k\left(e_k\left(e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right) \oplus m_3\right) \oplus \mathbb{P}(3b)\right) \\
&= M_3.
\end{aligned}$$

Hence, we see that on their own the non-trivial padding methods do not protect against MAC forgery attacks. This is one of the reasons for introducing the post processing steps. There are two popular post-processing steps, designed to make it more difficult for the cryptanalyst to perform an exhaustive key search and to protect against attacks such as the ones explained above:

(1) Choose a key k_1 and compute

$$O_q = e_k\left(d_{k_1}(O_q)\right).$$

(2) Choose a key k_1 and compute

$$O_q = e_{k_1}(O_q).$$

Both of these post-processing steps were invented when DES was the dominant cipher, and in such a situation the first of these is equivalent to processing the final block of the message using the 3DES algorithm.

Chapter Summary

- Hash functions are required which are both preimage, collision and second-preimage resistant.
- Due to the birthday paradox the output of the hash function should be at least twice the size of what one believes to be the limit of the computational ability of the attacker.
- More hash functions are iterative in nature, although most of the currently deployed ones have recently shown to be weaker than expected.
- A message authentication code is in some sense a keyed hash function.
- MACs can be created out of either block ciphers or hash functions.

Further Reading

A detailed description of both SHA-1 and the SHA-2 algorithms can be found in the FIPS standard below, this includes a set of test vectors as well. The recent work on the analysis of SHA-1, and references to the earlier attacks on MD4 and MD5 can be found in the papers og Wang et. al., of which we list only one below.

FIPS PUB 180-2, *Secure Hash Standard (including SHA-1, SHA-256, SHA-384, and SHA-512)*. NIST, 2005.

X. Wang, Y.L. Yin and H. Yu. *Finding Collisions in the Full SHA-1* In Advances in Cryptology – CRYPTO 2005, Springer-Verlag LNCS 3621, pp 17-36, 2005.

CHAPTER 11

Basic Public Key Encryption Algorithms

Chapter Goals

- To learn about public key encryption and the hard problems on which it is based.
- To understand the RSA algorithm and the assumptions on which its security relies.
- To understand the ElGamal encryption algorithm and it assumptions.
- To learn about the Rabin encryption algorithm and its assumptions.
- To learn about the Paillier encryption algorithm and its assumptions.

1. Public Key Cryptography

Recall that in symmetric key cryptography each communicating party needed to have a copy of the same secret key. This led to a very difficult key management problem. In public key cryptography we replace the use of identical keys with two keys, one public and one private.

The public key can be published in a directory along with the user's name. Anyone who then wishes to send a message to the holder of the associated private key will take the public key, encrypt a message under it and send it to the owner of the corresponding private key. The idea is that only the holder of the private key will be able to decrypt the message. More clearly, we have the transforms

$$\text{Message} + \text{Alice's public key} = \text{Ciphertext},$$
$$\text{Ciphertext} + \text{Alice's private key} = \text{Message}.$$

Hence anyone with Alice's public key can send Alice a secret message. But only Alice can decrypt the message, since only Alice has the corresponding private key.

Public key systems work because the two keys are linked in a mathematical way, such that knowing the public key tells you nothing about the private key. But knowing the private key allows you to unlock information encrypted with the public key. This may seem strange, and will require some thought and patience to understand. The concept was so strange it was not until 1976 that anyone thought of it. The idea was first presented in the seminal paper of Diffie and Hellman entitled *New Directions in Cryptography*. Although Diffie and Hellman invented the concept of public key cryptography it was not until a year or so later that the first (and most successful) system, namely RSA, was invented.

The previous paragraph is how the 'official' history of public key cryptography goes. However, in the late 1990s an unofficial history came to light. It turned out that in 1969, over five years before Diffie and Hellman invented public key cryptography, a cryptographer called James Ellis, working for the British government's communication headquarters GCHQ, invented the concept of public key cryptography (or non-secret encryption as he called it) as a means of solving the key distribution problem. Ellis, just like Diffie and Hellman, did not however have a system.

The problem of finding such a public key encryption system was given to a new recruit to GCHQ called Clifford Cocks in 1973. Within a day Cocks had invented what was essentially the RSA algorithm, although a full four years before Rivest, Shamir and Adleman. In 1974 another

employee at GCHQ, Malcolm Williamson, invented the concept of Diffie–Hellman key exchange, which we shall return to in Chapter 14. Hence, by 1974 the British security services had already discovered the main techniques in public key cryptography.

There is a surprisingly small number of ideas behind public key encryption algorithms, which may explain why once Diffie and Hellman or Ellis had the concept of public key encryption, an invention of essentially the same cipher, i.e. RSA, came so quickly. There are so few ideas because we require a mathematical operation which is easy to do one way, i.e. encryption, but which is hard to do the other way, i.e. decryption, without some special secret information, namely the private key. Such a mathematical function is called a trapdoor one-way function, since it is effectively a one-way function unless one knows the key to the trapdoor.

Luckily there are a number of possible one-way functions which have been well studied, such as factoring integers, computing discrete logarithms or computing square roots modulo a composite number. In the next section we shall study such one-way functions, before presenting some public key encryption algorithms later in the chapter. However, these are only computational one-way functions in that given enough computing power one can invert these functions faster than exhaustive search.

2. Candidate One-way Functions

The most important one-way function used in public key cryptography is that of factoring integers. By factoring an integer we mean finding its prime factors, for example

$$\begin{aligned} 10 &= 2 \cdot 5 \\ 60 &= 2^2 \cdot 3 \cdot 5 \\ 2^{113} - 1 &= 3391 \cdot 23\,279 \cdot 65\,993 \cdot 1\,868\,569 \cdot 1\,066\,818\,132\,868\,207 \end{aligned}$$

Finding the factors is an expensive computational operation. To measure the complexity of algorithms to factor an integer N we often use the function

$$L_N(\alpha, \beta) = \exp\left((\beta + o(1))(\log N)^{\alpha}(\log\log N)^{1-\alpha}\right).$$

Notice that if an algorithm to factor an integer has complexity $O(L_N(0, \beta))$, then it runs in polynomial time (recall the input size of the problem is $\log N$). However, if an algorithm to factor an integer has complexity $O(L_N(1, \beta))$ then it runs in exponential time. Hence, the function $L_N(\alpha, \beta)$ for $0 < \alpha < 1$ interpolates between polynomial and exponential time. An algorithm with complexity $O(L_N(\alpha, \beta))$ for $0 < \alpha < 1$ is said to have sub-exponential behaviour. Notice that multiplication, which is the inverse algorithm to factoring, is a very simple operation requiring time less than $O(L_N(0, 2))$.

There are a number of methods to factor numbers of the form

$$N = p \cdot q,$$

some of which we shall discuss in a later chapter. For now we just summarize the most well-known techniques.

- **Trial Division:** Try every prime number up to $\sqrt{N}$ and see if it is a factor of N. This has complexity $L_N(1, 1)$, and is therefore an exponential algorithm.
- **Elliptic Curve Method:** This is a very good method if $p < 2^{50}$, its complexity is $L_p(1/2, c)$, which is sub-exponential. Notice that the complexity is given in terms of the size of the unknown value p. If the number is a product of two primes of very unequal size then the elliptic curve method may be the best at finding the factors.
- **Quadratic Sieve:** This is probably the fastest method for factoring integers of between 80 and 100 decimal digits. It has complexity $L_N(1/2, 1)$.

- **Number Field Sieve:** This is currently the most successful method for numbers with more than 100 decimal digits. It can factor numbers of the size of $10^{155} \approx 2^{512}$ and has complexity $L_N(1/3, 1.923)$.

There are a number of other hard problems related to factoring which can be used to produce public key cryptosystems. Suppose you are given N but not its factors p and q, there are four main problems which one can try to solve:

- **FACTORING:** Find p and q.
- **RSA:** Given e such that
$$\gcd(e, (p-1)(q-1)) = 1$$
and c, find m such that
$$m^e = c \pmod{N}.$$
- **QUADRES:** Given a, determine whether a is a square modulo N.
- **SQRROOT:** Given a such that
$$a = x^2 \pmod{N},$$
find x.

Another important class of problems are those based on the discrete logarithm problem or its variants. Let $(G, \cdot)$ be a finite abelian group, such as the multiplicative group of a finite field or the set of points on an elliptic curve over a finite field. The discrete logarithm problem, or DLP, in G is given $g, h \in G$, find an integer x (if it exists) such that
$$g^x = h.$$
For some groups G this problem is easy. For example if we take G to be the integers modulo a number N under addition then given $g, h \in \mathbb{Z}/N\mathbb{Z}$ we need to solve
$$x \cdot g = h.$$
We have already seen in Chapter 1 that we can easily tell whether such an equation has a solution, and determine its solution when it does, using the extended Euclidean algorithm.

For certain other groups determining discrete logarithms is believed to be hard. For example in the multiplicative group of a finite field the best known algorithm for this task is the Number Field Sieve. The complexity of determining discrete logarithms in this case is given by
$$L_N(1/3, c)$$
for some constant c, depending on the type of the finite field, e.g. whether it is a large prime field or an extension field of characteristic two.

For other groups, such as elliptic curve groups, the discrete logarithm problem is believed to be even harder. The best known algorithm for finding discrete logarithms on a general elliptic curve defined over a finite field $\mathbb{F}_q$ is Pollard's Rho method which has complexity
$$\sqrt{q} = L_q(1, 1/2).$$
Hence, this is a fully exponential algorithm. Since determining elliptic curve discrete logarithms is harder than in the case of multiplicative groups of finite fields we are able to use smaller groups. This leads to an advantage in key size. Elliptic curve cryptosystems often have much smaller key sizes (say 160 bits) compared with those based on factoring or discrete logarithms in finite fields (where for both the 'equivalent' recommended key size is about 1024 bits).

Just as with the FACTORING problem, there are a number of related problems associated to discrete logarithms; again suppose we are given a finite abelian group $(G, \cdot)$ and $g \in G$.

- **DLP:** This is the discrete logarithm problem considered above. Namely given $g, h \in G$ such that $h = g^x$, find x.

- **DHP:** This is the Diffie–Hellman problem. Given $g \in G$ and
$$a = g^x \text{ and } b = g^y,$$
find c such that
$$c = g^{xy}.$$
- **DDH:** This is the decision Diffie–Hellman problem. Given $g \in G$ and
$$a = g^x,\ b = g^y \text{ and } c = g^z,$$
determine if $z = x \cdot y$.

When giving all these problems it is important to know how they are all related. This is done by giving complexity theoretic reductions from one problem to another. This allows us to say that 'Problem A is no harder than Problem B'. We do this by assuming an oracle (or efficient algorithm) to solve Problem B. We then use this oracle to give an efficient algorithm for Problem A. Hence, we reduce the problem of solving Problem A to inventing an efficient algorithm to solve Problem B. The algorithms which perform these reductions should be efficient, in that they run in polynomial time, where we treat each oracle query as a single step.

We can also show *equivalence* between two problems A and B, by showing an efficient reduction from A to B and an efficient reduction from B to A. If the two reductions are both polynomial-time reductions then we say that the two problems are *polynomial-time equivalent*.

As an example we first show how to reduce solving the Diffie–Hellman problem to the discrete logarithm problem.

LEMMA 11.1. *In an arbitrary finite abelian group G the DHP is no harder than the DLP.*

PROOF. Suppose I have an oracle $\mathcal{O}_{\text{DLP}}$ which will solve the DLP for me, i.e. on input of $h = g^x$ it will return x. To solve the DHP on input of $a = g^x$ and $b = g^y$ we compute

(1) $z = \mathcal{O}_{\text{DLP}}(a)$.
(2) $c = b^z$.
(3) Output c.

The above reduction clearly runs in polynomial time and will compute the true solution to the DHP, assuming the oracle returns the correct value, i.e.
$$z = x.$$
Hence, the DHP is no harder than the DLP. □

In some groups there is a more complicated argument to show that the DHP is in fact equivalent to the DLP.

We now show how to reduce the solution of the decision Diffie–Hellman problem to the Diffie–Hellman problem, and hence using our previous argument to the discrete logarithm problem.

LEMMA 11.2. *In an arbitrary finite abelian group G the DDH is no harder than the DHP.*

PROOF. Now suppose we have an oracle $\mathcal{O}_{\text{DHP}}$ which on input of g^x and g^y computes the value of g^{xy}. To solve the DDH on input of $a = g^x, b = g^y$ and $c = g^z$ we compute

(1) $d = \mathcal{O}_{\text{DHP}}(a, b)$.
(2) If $d = c$ output YES.
(3) Else output NO.

Again the reduction clearly runs in polynomial time, and assuming the output of the oracle is correct then the above reduction will solve the DDH. □

So the decision Diffie–Hellman problem is no harder than the computational Diffie–Hellman problem. There are however some groups in which one can solve the DDH in polynomial time but the fastest known algorithm to solve the DHP takes sub-exponential time.

Hence, of our three discrete logarithm based problems, the easiest is DDH, then comes DHP and finally the hardest problem is DLP.

We now turn to show reductions for the factoring based problems. The most important result is

LEMMA 11.3. *The FACTORING and SQRROOT problems are polynomial-time equivalent.*

PROOF. We first show how to reduce SQRROOT to FACTORING. Assume we are given a factoring oracle, we wish to show how to use this to extract square roots modulo a composite number N. Namely, given

$$z = x^2 \pmod{N}$$

we wish to compute x. First we factor N into its prime factors p_i using the factoring oracle. Then we compute

$$s_i = \sqrt{z} \pmod{p_i},$$

this can be done in expected polynomial time using Shanks' Algorithm. Then we compute the value of x using the Chinese Remainder Theorem on the data

$$s_i = \sqrt{z} \pmod{p_i}.$$

One has to be a little careful if powers of p_i greater than one divide N, but this is easy to deal with and will not concern us here. Hence, finding square roots modulo N is no harder than factoring.

We now show that FACTORING can be reduced to SQRROOT. Assume we are given an oracle for extracting square roots modulo a composite number N. We shall assume for simplicity that N is a product of two primes, which is the most difficult case. The general case is only slightly more tricky mathematically, but it is computationally easier since factoring numbers with three or more prime factors is usually easier than factoring numbers with two prime factors.

We wish to use our oracle for the problem SQRROOT to factor the integer N into its prime factors, i.e. given $N = p \cdot q$ we wish to compute p. First we pick a random $x \in (\mathbb{Z}/N\mathbb{Z})^*$ and compute

$$z = x^2 \pmod{N}.$$

Now we compute

$$y = \sqrt{z} \pmod{N}$$

using the SQRROOT oracle. There are four such square roots, since N is a product of two primes. With 50 percent probability we obtain

$$y \neq \pm x \pmod{N}.$$

If we do not obtain this inequality then we simply repeat the method. We expect after an average number of two repetitions we will obtain the desired inequality.

Now, since $x^2 = y^2 \pmod{N}$, we see that N divides

$$x^2 - y^2 = (x - y)(x + y).$$

But N does not divide either $x - y$ or $x + y$, since $y \neq \pm x \pmod{N}$. So the factors of N must be distributed over these later two pairs of numbers. This means we can obtain a non-trivial factor of N by computing $\gcd(x - y, N)$

Clearly both of the above reductions can be performed in expected polynomial time. Hence, the problems FACTORING and SQRROOT are polynomial-time equivalent. □

The above proof contains an important tool used in factoring algorithms, namely the construction of a difference of two squares. We shall return to this later in Chapter 12.

Before leaving the problem SQRROOT notice that QUADRES is easier than SQRROOT, since an algorithm to compute square roots modulo N can be used to determine quadratic residuosity.

Finally we end this section by showing that the RSA problem can be reduced to FACTORING. Recall the RSA problem is given $c = m^e \pmod N$, find m.

LEMMA 11.4. *The RSA problem is no harder than the FACTORING problem.*

PROOF. Using a factoring oracle we first find the factorization of N. We can now compute $\Phi = \phi(N)$ and then compute

$$d = 1/e \pmod \Phi.$$

Once d has been computed it is easy to recover m via

$$c^d = m^{ed} = m^{1 \pmod \Phi} = m \pmod N.$$

Hence, the RSA problem is no harder than FACTORING. □

There is some evidence, although slight, that the RSA problem may actually be easier than FACTORING for some problem instances. It is a major open question as to how much easier it is.

3. RSA

The RSA algorithm was the world's first public key encryption algorithm, and it has stood the test of time remarkably well. The RSA algorithm is based on the difficulty of the RSA problem considered in the previous section, and hence it is based on the difficulty of finding the prime factors of large integers. We have seen that it may be possible to solve the RSA problem without factoring, hence the RSA algorithm is not based completely on the difficulty of factoring.

Suppose Alice wishes to enable anyone to send her secret messages, which only she can decrypt. She first picks two large secret prime numbers p and q. Alice then computes

$$N = p \cdot q.$$

Alice also chooses an encryption exponent e which satisfies

$$\gcd(e, (p-1)(q-1)) = 1.$$

It is common to choose $e = 3, 17$ or $65\,537$. Now Alice's public key is the pair (N, e), which she can publish in a public directory. To compute the private key Alice applies the extended Euclidean algorithm to e and $(p-1)(q-1)$ to obtain the decryption exponent d, which should satisfy

$$e \cdot d \equiv 1 \pmod{(p-1)(q-1)}.$$

Alice keeps secret her private key, which is the triple (d, p, q). Actually, she could simply throw away p and q, and retain a copy of her public key which contains the integer N, but we shall see later that this is not efficient.

Now suppose Bob wishes to encrypt a message to Alice. He first looks up Alice's public key and represents the message as a number m which is strictly less than the public modulus N. The ciphertext is then produced by raising the message to the power of the public encryption exponent modulo the public modulus, i.e.

$$c = m^e \pmod N.$$

Alice on receiving c can decrypt the ciphertext to recover the message by exponentiating by the private decryption exponent, i.e.

$$m = c^d \pmod N.$$

This works since the group $(\mathbb{Z}/N\mathbb{Z})^*$ has order

$$\phi(N) = (p-1)(q-1)$$

and so, by Lagrange's Theorem,

$$x^{(p-1)(q-1)} \equiv 1 \pmod{N},$$

for all $x \in (\mathbb{Z}/N\mathbb{Z})^*$. For some integer s we have

$$ed - s(p-1)(q-1) = 1,$$

and so

$$\begin{aligned} c^d &= (m^e)^d \\ &= m^{ed} \\ &= m^{1+s(p-1)(q-1)} \\ &= m \cdot m^{s(p-1)(q-1)} \\ &= m. \end{aligned}$$

To make things clearer let's consider a baby example. Choose $p = 7$ and $q = 11$, and so $N = 77$ and $(p-1)(q-1) = 6 \cdot 10 = 60$. We pick as the public encryption exponent $e = 37$, since we have $\gcd(37, 60) = 1$. Then, applying the extended Euclidean algorithm we obtain $d = 13$ since

$$37 \cdot 13 = 481 = 1 \pmod{60}.$$

Suppose the message we wish to transmit is given by $m = 2$, then to encrypt m we compute

$$c = m^e \pmod{N} = 2^{37} \pmod{77} = 51,$$

whilst to decrypt the ciphertext c we compute

$$m = c^d \pmod{N} = 51^{13} \pmod{77} = 2.$$

3.1. RSA Encryption and the RSA Problem. The security of RSA on first inspection relies on the difficulty of finding the private encryption exponent d given only the public key, namely the public modulus N and the public encryption exponent e.

We have shown that the RSA problem is no harder than FACTORING, hence if we can factor N then we can find p and q and hence we can calculate d. Hence, if factoring is easy we can break RSA. Currently 500-bit numbers are the largest that have been factored and so it is recommended that one takes public moduli of size around 1024 bits to ensure medium-term security. For long-term security one would need to take a public modulus size of over 2048 bits.

In this chapter we shall consider security to be defined as being unable to recover the whole plaintext given the ciphertext. We shall argue in a later chapter that this is far too weak a definition of security for many applications. In addition in a later chapter we shall show that RSA, as we have described it, is not secure against a chosen ciphertext attack.

For a public key algorithm the adversary always has access to the encryption algorithm, hence she can always mount a chosen plaintext attack. RSA is secure against a chosen plaintext attack assuming our weak definition of security and that the RSA problem is hard. To show this we use the reduction arguments of the previous section. This example is rather trivial but we labour the point since these arguments are used over and over again in later chapters.

LEMMA 11.5. *If the RSA problem is hard then the RSA system is secure under a chosen plaintext attack, in the sense that an attacker is unable to recover the whole plaintext given the ciphertext.*

PROOF. We wish to give an algorithm which solves the RSA problem using an algorithm to break the RSA cryptosystem as an oracle. If we can show this then we can conclude that the breaking the RSA cryptosystem is no harder than solving the RSA problem.

Recall that the RSA problem is given $N = p \cdot q$, e and $y \in (\mathbb{Z}/N\mathbb{Z})^*$, compute an x such that $x^e \pmod N = y$. We use our oracle to break the RSA encryption algorithm to 'decrypt' the message corresponding to $c = y$, this oracle will return the plaintext message m. Then our RSA problem is solved by setting $x = m$ since, by definition,

$$m^e \pmod N = c = y.$$

So if we can break the RSA algorithm then we can solve the RSA problem. □

3.2. Knowledge of the Private Exponent and Factoring. Whilst it is unclear whether breaking RSA, in the sense of inverting the RSA function, is equivalent to factoring, determining the private key d given the public information N and e is equivalent to factoring.

LEMMA 11.6. *If one knows the RSA decryption exponent d corresponding to the public key (N, e) then one can efficiently factor N.*

PROOF. Recall that for some integer s

$$ed - 1 = s(p-1)(q-1).$$

We pick an integer $x \neq 0$, this is guaranteed to satisfy

$$x^{ed-1} = 1 \pmod N.$$

We now compute a square root y_1 of one modulo N,

$$y_1 = \sqrt{x^{ed-1}} = x^{(ed-1)/2},$$

which we can do since $ed - 1$ is known and will be even. We will then have the identity

$${y_1}^2 - 1 \equiv 0 \pmod N,$$

which we can use to recover a factor of N via computing

$$\gcd(y_1 - 1, N).$$

But this will only work when $y_1 \neq \pm 1 \pmod N$.

Now suppose we are unlucky and we obtain $y_1 = \pm 1 \pmod N$ rather than a factor of N. If $y_1 = -1 \pmod N$ we return to the beginning and pick another value of x. This leaves us with the case $y_1 = 1 \pmod N$, in which case we take another square root of one via,

$$y_2 = \sqrt{y_1} = x^{(ed-1)/4}.$$

Again we have

$${y_2}^2 - 1 = y_1 - 1 = 0 \pmod N.$$

Hence we compute

$$\gcd(y_2 - 1, N)$$

and see if this gives a factor of N. Again this will give a factor of N unless $y_2 = \pm 1$, if we are unlucky we repeat once more and so on.

This method can be repeated until either we have factored N or until $(ed-1)/2^t$ is no longer divisible by 2. In this latter case we return to the beginning, choose a new random value of x and start again. □

The algorithm in the above proof is an example of a Las Vegas Algorithm: It is probabilistic in nature in the sense that whilst it may not actually give an answer (or terminate), it is however guaranteed that when it does give an answer then that answer will always be correct.

We shall now present a small example of the previous method. Consider the following RSA parameters

$$N = 1\,441\,499, e = 17 \text{ and } d = 507\,905.$$

Recall we are assuming that the private exponent d is public knowledge. We will show that the previous method does in fact find a factor of N. Put

$$\begin{aligned} t_1 &= (ed-1)/2 = 4\,317\,192, \\ x &= 2. \end{aligned}$$

To compute y_1 we evaluate

$$\begin{aligned} y_1 &= x^{(ed-1)/2}, \\ &= 2^{t_1}, \\ &= 1 \pmod{N}. \end{aligned}$$

Since we obtain $y_1 = 1$ we need to set

$$\begin{aligned} t_2 &= t_1/2 = (ed-1)/4 = 2\,158\,596, \\ y_2 &= 2^{t_2}. \end{aligned}$$

We now compute y_2,

$$\begin{aligned} y_2 &= x^{(ed-1)/4}, \\ &= 2^{t_2}, \\ &= 1 \pmod{N}. \end{aligned}$$

So we need to repeat the method again, this time we obtain $t_3 = (ed-1)/8 = 1\,079\,298$. We compute y_3,

$$\begin{aligned} y_3 &= x^{(ed-1)/8}, \\ &= 2^{t_3}, \\ &= 119\,533 \pmod{N}. \end{aligned}$$

So

$$y_3{}^2 - 1 = (y_3 - 1)(y_3 + 1) \equiv 0 \pmod{N},$$

and we compute a prime factor of N by evaluating

$$\gcd(y_3 - 1, N) = 1423.$$

3.3. Knowledge of $\phi(N)$ and Factoring. We have seen that knowledge of d allows us to factor N. Now we will show that knowledge of $\Phi = \phi(N)$ also allows us to factor N.

LEMMA 11.7. *Given an RSA modulus N and the value of $\Phi = \phi(N)$ one can efficiently factor N.*

PROOF. We have

$$\Phi = (p-1)(q-1) = N - (p+q) + 1.$$

Hence, if we set $S = N + 1 - \Phi$, we obtain

$$S = p + q.$$

So we need to determine p and q from their sum S and product N. Define the polynomial

$$f(X) = (X - p) \cdot (X - q) = X^2 - SX + N.$$

So we can find p and q by solving $f(X) = 0$ using the standard formulae for extracting the roots of a quadratic polynomial,

$$p = \frac{S + \sqrt{S^2 - 4N}}{2},$$
$$q = \frac{S - \sqrt{S^2 - 4N}}{2}.$$

□

As an example consider the RSA public modulus $N = 18\,923$. Assume that we are given $\Phi = \phi(N) = 18\,648$. We then compute

$$S = p + q = N + 1 - \Phi = 276.$$

Using this we compute the polynomial

$$f(X) = X^2 - SX + N = X^2 - 276X + 18\,923$$

and find that its roots over the real numbers are

$$p = 149, q = 127$$

which are indeed the factors of N.

3.4. Use of a Shared Modulus. Since modular arithmetic is very expensive it can be very tempting for a system to be set up in which a number of users share the same public modulus N but use different public/private exponents, (e_i, d_i). One reason to do this could be to allow very fast hardware acceleration of modular arithmetic, specially tuned to the chosen shared modulus N. This is, however, a very silly idea since it can be attacked in one of two ways, either by a malicious insider or by an external attacker.

Suppose the bad guy is one of the internal users, say user number one. He can now compute the value of the decryption exponent for user number two, namely d_2. First user one computes p and q since they know d_1, via the algorithm in the proof of Lemma 11.6. Then user one computes $\phi(N) = (p-1)(q-1)$, and finally they can recover d_2 from

$$d_2 = \frac{1}{e_2} \pmod{\phi(N)}.$$

Now suppose the attacker is not one of the people who share the modulus. Suppose Alice sends the same message m to two of the users with public keys

$$(N, e_1) \text{ and } (N, e_2),$$

i.e. $N_1 = N_2 = N$. Eve, the external attacker, sees the messages c_1 and c_2 where

$$c_1 = m^{e_1} \pmod{N},$$
$$c_2 = m^{e_2} \pmod{N}.$$

Eve can now compute

$$t_1 = {e_1}^{-1} \pmod{e_2},$$
$$t_2 = (t_1 e_1 - 1)/e_2,$$

and can recover the message m from

$$\begin{aligned} c_1^{t_1} c_2^{-t_2} &= m^{e_1 t_1} m^{-e_2 t_2} \\ &= m^{1+e_2 t_2} m^{-e_2 t_2} \\ &= m^{1+e_2 t_2 - e_2 t_2} \\ &= m^1 = m. \end{aligned}$$

As an example of this external attack, take the public keys as

$$N = N_1 = N_2 = 18\,923,\ e_1 = 11 \text{ and } e_2 = 5.$$

Now suppose Eve sees the ciphertexts

$$c_1 = 1514 \text{ and } c_2 = 8189$$

corresponding to the same plaintext m. Then Eve computes $t_1 = 1$ and $t_2 = 2$, and recovers the message

$$m = c_1^{t_1} c_2^{-t_2} = 100 \pmod{N}.$$

3.5. Use of a Small Public Exponent. Fielded RSA systems often use a small public exponent e so as to cut down the computational cost of the sender. We shall now show that this can also lead to problems. Suppose we have three users all with different public moduli

$$N_1,\ N_2 \text{ and } N_3.$$

In addition suppose they all have the same small public exponent $e = 3$. Suppose someone sends them the same message m.

The attacker Eve sees the messages

$$\begin{aligned} c_1 &= m^3 \pmod{N_1}, \\ c_2 &= m^3 \pmod{N_2}, \\ c_3 &= m^3 \pmod{N_3}. \end{aligned}$$

Now the attacker, using the Chinese Remainder Theorem, computes the simultaneous solution to

$$X = c_i \pmod{N_i} \text{ for } i = 1, 2, 3,$$

to obtain

$$X = m^3 \pmod{N_1 N_2 N_3}.$$

But since $m^3 < N_1 N_2 N_3$ we must have $X = m^3$ identically over the integers. Hence we can recover m by taking the real cube root of X.

As a simple example of this attack take,

$$N_1 = 323,\ N_2 = 299 \text{ and } N_3 = 341.$$

Suppose Eve sees the ciphertexts

$$c_1 = 50,\ c_2 = 268 \text{ and } c_3 = 1,$$

and wants to determine the common value of m. Eve computes via the Chinese Remainder Theorem

$$X = 300\,763 \pmod{N_1 N_2 N_3}.$$

Finally, she computes over the integers

$$m = X^{1/3} = 67.$$

This attack and the previous one are interesting since we find the message without factoring the modulus. This is, albeit slight, evidence that breaking RSA is easier than factoring. The main lesson, however, from both these attacks is that plaintext should be randomly padded before transmission. That way the same 'message' is never encrypted to two different people. In addition

one should probably avoid very small exponents for encryption, $e = 65\,537$ is the usual choice now in use. However, small public exponents for RSA signatures (see later) do not seem to have any problems.

4. ElGamal Encryption

The simplest encryption algorithm based on the discrete logarithm problem is the ElGamal encryption algorithm. In the following we shall describe the finite field analogue of ElGamal encryption, we leave it as an exercise to write down the elliptic curve variant.

Unlike the RSA algorithm, in ElGamal encryption there are some public parameters which can be shared by a number of users. These are called the domain parameters and are given by

- p a 'large prime', by which we mean one with around 1024 bits, such that $p-1$ is divisible by another 'medium prime' q of around 160 bits.
- g an element of $\mathbb{F}_p^*$ of prime order q, i.e.

$$g = r^{(p-1)/q} \pmod{p} \neq 1 \text{ for some } r \in \mathbb{F}_p^*.$$

All the domain parameters do is create a public finite abelian group G of prime order q with generator g. Such domain parameters can be shared between a large number of users.

Once these domain parameters have been fixed, the public and private keys can then be determined. The private key is chosen to be an integer x, whilst the public key is given by

$$h = g^x \pmod{p}.$$

Notice that whilst each user in RSA needed to generate two large primes to set up their key pair (which is a costly task), for ElGamal encryption each user only needs to generate a random number and perform a modular exponentiation to generate a key pair.

Messages are assumed to be non-zero elements of the field $\mathbb{F}_p^*$. To encrypt a message $m \in \mathbb{F}_p^*$ we

- generate a random ephemeral key k,
- set $c_1 = g^k$,
- set $c_2 = m \cdot h^k$,
- output the ciphertext as $c = (c_1, c_2)$.

Notice that since each message has a different ephemeral key, encrypting the same message twice will produce different ciphertexts.

To decrypt a ciphertext $c = (c_1, c_2)$ we compute

$$\begin{aligned} \frac{c_2}{{c_1}^x} &= \frac{m \cdot h^k}{g^{xk}} \\ &= \frac{m \cdot g^{xk}}{g^{xk}} \\ &= m. \end{aligned}$$

As an example of ElGamal encryption consider the following. We first need to set up the domain parameters. For our small example we choose

$$q = 101,\ p = 809 \text{ and } g = 3.$$

Note that q divides $p-1$ and that g has order divisible by q in the multiplicative group of integers modulo p. The actual order of g is 808 since

$$3^{808} = 1 \pmod{p},$$

and no smaller power of g is equal to one. As a public private key pair we choose

- $x = 68$,
- $h = g^x = 65$.

Now suppose we wish to encrypt the message $m = 100$ to the user with the above ElGamal public key.

- We generate a random ephemeral key $k = 89$.
- Set $c_1 = g^k = 345$.
- Set $c_2 = m \cdot h^k = 517$.
- Output the ciphertext as $c = (345, 517)$.

The recipient can decrypt our ciphertext by computing

$$\begin{aligned} \frac{c_2}{{c_1}^x} &= \frac{517}{345^{68}} \\ &= 100. \end{aligned}$$

This last value is computed by first computing 345^{68}, taking the inverse modulo p of the result and then multiplying this value by 517.

In a later chapter we shall see that ElGamal encryption as it stands is not secure against a chosen ciphertext attack, so usually a modified scheme is used. However, ElGamal encryption is secure against a chosen plaintext attack, assuming the Diffie–Hellman problem is hard. Again, here we take a naive definition of what security means in that an encryption algorithm is secure if an adversary is unable to invert the encryption function.

LEMMA 11.8. *Assuming the Diffie–Hellman problem (DHP) is hard then ElGamal is secure under a chosen plaintext attack, where security means it is hard for the adversary, given the ciphertext, to recover the whole of the plaintext.*

PROOF. To see that ElGamal encryption is secure under a chosen plaintext attack assuming the Diffie–Hellman problem is hard, we first suppose that we have an oracle $\mathcal{O}$ to break ElGamal encryption. This oracle $\mathcal{O}(h, (c_1, c_2))$ takes as input a public key h and a ciphertext (c_1, c_2) and then returns the underlying plaintext. We will then show how to use this oracle to solve the DHP.

Suppose we are given

$$g^x \text{ and } g^y$$

and we are asked to solve the DHP, i.e. we need to compute g^{xy}.

We first set up an ElGamal public key which depends on the input to this Diffie–Hellman problem, i.e. we set

$$h = g^x.$$

Note, we do not know what the corresponding private key is. Now we write down the 'ciphertext'

$$c = (c_1, c_2),$$

where

- $c_1 = g^y$,
- c_2 is a random element of $\mathbb{F}_p^*$.

Now we input this ciphertext into our oracle which breaks ElGamal encryption so as to produce the corresponding plaintext, $m = \mathcal{O}(h, (c_1, c_2))$. We can now solve the original Diffie–Hellman problem by computing

$$\begin{aligned} \frac{c_2}{m} &= \frac{m \cdot h^y}{m} \quad \text{since } c_1 = g^y \\ &= h^y \\ &= g^{xy}. \end{aligned}$$

□

5. Rabin Encryption

There is another system, due to Rabin, based on the difficulty of factoring large integers. In fact it is actually based on the difficulty of extracting square roots modulo $N = p \cdot q$. Recall that these two problems are known to be equivalent, i.e.

- knowing the factors of N means we can extract square roots modulo N,
- extracting square roots modulo N means we can factor N.

Hence, in some respects such a system should be considered more secure than RSA. Encryption in the Rabin encryption system is also much faster than almost any other public key scheme. Despite these plus points the Rabin system is not used as much as the RSA system. It is, however, useful to study for a number of reasons, both historical and theoretical. The basic idea of the system is also used in some higher level protocols.

We first choose prime numbers of the form

$$p \equiv q \equiv 3 \pmod 4$$

since this makes extracting square roots modulo p and q very fast. The private key is then the pair (p, q). To compute the associated public key we generate a random integer $B \in \{0, \ldots, N-1\}$ and then the public key is

$$(N, B),$$

where N is the product of p and q.

To encrypt a message m, using the above public key, in the Rabin encryption algorithm we compute

$$c = m(m + B) \pmod N.$$

Hence, encryption involves one addition and one multiplication modulo N. Encryption is therefore much faster than RSA encryption, even when one chooses a small RSA encryption exponent.

Decryption is far more complicated, essentially we want to compute

$$m = \sqrt{\frac{B^2}{4} + c} - \frac{B}{2} \pmod N.$$

At first sight this uses no private information, but a moment's thought reveals that you need the factorization of N to be able to find the square root. There are however four possible square roots modulo N, since N is the product of two primes. Hence, on decryption you obtain four possible plaintexts. This means that we need to add redundancy to the plaintext before encryption in order to decide which of the four possible plaintexts corresponds to the intended one.

We still need to show why Rabin decryption works. Recall

$$c = m(m + B) \pmod N,$$

then

$$\begin{aligned}
\sqrt{\frac{B^2}{4} + c} - \frac{B}{2} &= \sqrt{\frac{B^2 + 4m(m+B)}{4}} - \frac{B}{2} \\
&= \sqrt{\frac{4m^2 + 4Bm + B^2}{4}} - \frac{B}{2} \\
&= \sqrt{\frac{(2m+B)^2}{4}} - \frac{B}{2} \\
&= \frac{2m + B}{2} - \frac{B}{2} \\
&= m,
\end{aligned}$$

of course assuming the 'correct' square root is taken.

We end with an example of Rabin encryption at work. Let the public and private keys be given by

- $p = 127$ and $q = 131$,
- $N = 16\,637$ and $B = 12\,345$.

To encrypt $m = 4410$ we compute

$$c = m(m + B) \pmod{N} = 4633.$$

To decrypt we first compute

$$t = B^2/4 + c \pmod{N} = 1500.$$

We then evaluate the square root of t modulo p and q

$$\sqrt{t} \pmod{p} = \pm 22,$$
$$\sqrt{t} \pmod{q} = \pm 37.$$

Now we apply the Chinese Remainder Theorem to both

$$\pm 22 \pmod{p} \text{ and } \pm 37 \pmod{q}$$

so as to find the square root of t modulo N,

$$s = \sqrt{t} \pmod{N} = \pm 3705 \text{ or } \pm 14\,373.$$

The four possible messages are then given by the four possible values of

$$s - \frac{B}{2} = s - \frac{12\,345}{2}.$$

This leaves us with the four messages

$$4410, 5851, 15\,078, \text{ or } 16\,519.$$

6. Paillier Encryption

There is another system, due to Paillier, based on the difficulty of factoring large integers. Paillier's scheme has a number of interesting properties, such as the fact that it is additively homomorphic (which means it has found application in electronic voting applications).

We first pick an RSA modulo $N = p \cdot q$, but instead of working with the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^*$ we work with $(\mathbb{Z}/N^2\mathbb{Z})^*$. The order of this last group is given by $\phi(N) = N \cdot (p-1) \cdot (q-1) = N \cdot \phi(N)$. Which means, by Lagrange's Theorem, that for all a with $\gcd(a, N) = 1$ we have

$$a^{N \cdot (p-1) \cdot (q-1)} \equiv 1 \pmod{N^2}.$$

The private key for Paillier's scheme is defined to be an integer d such that

$$d \equiv 1 \pmod{N},$$
$$d \equiv 0 \pmod{(p-1) \cdot (q-1)},$$

such a value of d can be found by the Chinese Remainder Theorem. The public key is just the integer N, whereas the private key is the integer d.

Messages are defined to be elements of $\mathbb{Z}/\mathbb{Z}$, to encrypt such a message the encryptor picks an integer $r \in \mathbb{Z}/N^2\mathbb{Z}$ and computes

$$c = (1 + N)^m \cdot r^N \pmod{N^2}.$$

To decrypt one first computes

$$\begin{aligned} t &= c^d \pmod{N^2} \\ &= (1+N)^{m\cdot d} \cdot r^{d\cdot N} \pmod{N^2} \\ &= (1+N)^{m\cdot d} \pmod{N^2} \text{ since } d \equiv 0 \pmod{(p-1)\cdot(q-1)} \\ &= 1 + m \cdot d \cdot N \pmod{N^2} \\ &= 1 + m \cdot N \pmod{N^2} \text{ since } d \equiv 1 \pmod{N}. \end{aligned}$$

Then to recover the message we compute

$$R = \frac{t-1}{N}.$$

Chapter Summary

- Public key encryption requires one-way functions. Examples of these are FACTORING, SQRROOT, DLP, DHP and DDH.
- There are a number of relationships between these problems. These relationships are proved by assuming an oracle for one problem and then using this in an algorithm to solve the other problem.
- RSA is the most popular public key encryption algorithm, but its security rests on the difficulty of the RSA problem and not quite on the difficulty of FACTORING.
- ElGamal encryption is a system based on the difficulty of the Diffie–Hellman problem (DHP).
- Rabin encryption is based on the difficulty of extracting square roots modulo a composite modulus. Since the problems SQRROOT and FACTORING are polynomial-time equivalent this means that Rabin encryption is based on the difficulty of FACTORING.
- Paillier encryption is a scheme which is based on the composite decision residuosity assumption.

Further Reading

Still the best quick introduction to the concept of public key cryptography can be found in the original paper of Diffie and Hellman. See also the original papers on ElGamal, Rabin and RSA encryption.

W. Diffie and M. Hellman. *New directions in cryptography.* IEEE Trans. on Info. Theory, **22**, 644–654, 1976.

T. ElGamal. *A public key cryptosystem and a signature scheme based on discrete logarithms.* IEEE Trans. Info. Theory, **31**, 469–472, 1985.

R.L. Rivest, A. Shamir and L.M. Adleman. *A method for obtaining digital signatures and public-key cryptosystems.* Comm. ACM, **21**, 120–126, 1978.

M. Rabin. *Digitized signatures and public key functions as intractable as factorization.* MIT/LCS/TR-212, MIT Laboratory for Computer Science, 1979.

CHAPTER 14

Key Exchange and Signature Schemes

Chapter Goals

- To introduce Diffie–Hellman key exchange.
- To introduce the need for digital signatures.
- To explain the two most used signature algorithms, namely RSA and DSA.
- To explain the need for cryptographic hash functions within signature schemes.
- To describe some other signature algorithms and key exchange techniques which have interesting properties.

1. Diffie–Hellman Key Exchange

Recall that the main drawback with the use of fast bulk encryption based on block or stream ciphers was the problem of key distribution. We have already seen a number of techniques to solve this problem, either using protocols which are themselves based on symmetric key techniques, or using a public key algorithm to transport a session key to the intended recipient. These, however, both have problems associated with them. For example, the symmetric key protocols were hard to analyse and required the use of already deployed long-term keys between each user and a trusted central authority.

A system is said to have forward secrecy, if the compromise of a long-term private key at some point in the future does not compromise the security of communications made using that key in the past. Key transport via public key encryption does not have *forward secrecy*. To see why this is important, suppose you bulk encrypt a video stream and then encrypt the session key under the recipient's RSA public key. Then suppose that some time in the future, the recipient's RSA private key is compromised. At that point your video stream is also compromised, assuming the attacker recorded this at the time it was transmitted.

In addition using key transport implies that the recipient trusts the sender to be able to generate, in a sensible way, the session key. Sometimes the recipient may wish to contribute some randomness of their own to the session key. However, this can only be done if both parties are online at the same moment in time. Key transport is more suited to the case where only the sender is online, as in applications like email for example.

The key distribution problem was solved in the same seminal paper by Diffie and Hellman as that in which they introduced public key cryptography. Their protocol for key distribution, called Diffie–Hellman Key Exchange, allows two parties to agree a secret key over an insecure channel without having met before. Its security is based on the discrete logarithm problem in a finite abelian group G.

In the original paper the group is taken to be $G = \mathbb{F}_p^*$, but now more efficient versions can be produced by taking G to be an elliptic curve group, where the protocol is called EC-DH. The basic message flows for the Diffie–Hellman protocol are given in the following diagram:

$$\begin{array}{ccc} \text{Alice} & & \text{Bob} \\ a & g^a \longrightarrow & g^a \\ g^b & \longleftarrow g^b & b \end{array}$$

The two parties each have their own ephemeral secrets a and b. From these secrets both parties can agree on the same secret session key:

- Alice can compute $K = (g^b)^a$, since she knows a and was sent g^b by Bob,
- Bob can also compute $K = (g^a)^b$, since he knows b and was sent g^a by Alice.

Eve, the attacker, can see the messages

$$g^a \text{ and } g^b$$

and then needs to recover the secret key

$$K = g^{ab}$$

which is exactly the Diffie–Hellman problem considered in Chapter 11. Hence, the security of the above protocol rests not on the difficulty of solving the discrete logarithm problem, DLP, but on the difficulty of solving the Diffie–Hellman problem, DHP. Recall that it may be the case that it is easier to solve the DHP than the DLP, although no one believes this to be true for the groups that are currently used in real-life protocols.

Notice that the Diffie–Hellman protocol can be performed both online (in which case both parties contribute to the randomness in the shared session key) or offline, where one of the parties uses a long-term key of the form g^a instead of an ephemeral key. Hence, the Diffie–Hellman protocol can be used as a key exchange or as a key transport protocol.

The following is a very small example, in real life one takes $p \approx 2^{1024}$, but for our purposes we let the domain parameters be given by

$$p = 2\,147\,483\,659 \text{ and } g = 2.$$

Then the following diagram indicates a possible message flow for the Diffie–Hellman protocol:

$$\begin{array}{lcl} \text{Alice} & & \text{Bob} \\ a = 12\,345 & & b = 654\,323 \\ A = g^a = 428\,647\,416 & \longrightarrow & A = 428\,647\,416 \\ B = 450\,904\,856 & \longleftarrow & B = g^b = 450\,904\,856 \end{array}$$

The shared secret key is then computed via

$$\begin{aligned} A^b &= 428\,647\,416^{654\,323} \pmod p, \\ &= 1\,333\,327\,162, \\ B^a &= 450\,904\,856^{12\,345} \pmod p, \\ &= 1\,333\,327\,162. \end{aligned}$$

Notice that group elements are transmitted in the protocol, hence when using a finite field such as $\mathbb{F}_p^*$ for the Diffie–Hellman protocol the communication costs are around 1024 bits in each direction, since it is prudent to choose $p \approx 2^{1024}$. However, when one uses an elliptic curve group $E(\mathbb{F}_q)$ one can choose $q \approx 2^{160}$, and so the communication costs are much less, namely around 160 bits in each direction. In addition the group exponentiation step for elliptic curves can be done more efficiently than that for finite prime fields.

As a baby example of EC-DH consider the elliptic curve

$$E : Y^2 = X^3 + X - 3$$

over the field $\mathbb{F}_{199}$. Let the base point be given by $G = (1, 76)$, then a possible message flow is given by

Alice		Bob
$a = 23$		$b = 86$
$A = [a]G = (2, 150)$	$\longrightarrow$	$A = (2, 150)$
$B = (123, 187)$	$\longleftarrow$	$B = [b]G = (123, 187)$

The shared secret key is then computed via

$$\begin{aligned} [b]A &= [86](2, 150) \\ &= (156, 75), \\ [a]B &= [23](123, 187) \\ &= (156, 75). \end{aligned}$$

The shared key is then taken to be the x-coordinate 156 of the computed point. In addition, instead of transmitting the points, we transmit the compression of the point, which results in a significant saving in bandwidth.

So we seem to have solved the key distribution problem. But there is an important problem: you need to be careful *who* you are agreeing a key with. Alice has no assurance that she is agreeing a key with Bob, which can lead to the following (wo)man in the middle attack:

Alice		Eve		Bob
a	$\longrightarrow$	g^a		
g^m	$\longleftarrow$	m		
g^{am}		g^{am}		
		n	$\longrightarrow$	g^n
		g^b	$\longleftarrow$	b
		g^{bn}		g^{bn}

In the man in the middle attack

- Alice agrees a key with Eve, thinking it is Bob she is agreeing a key with,
- Bob agrees a key with Eve, thinking it is Alice,
- Eve can now examine communications as they pass through her, she acts as a router. She does not alter the plaintext, so her actions go undetected.

So we can conclude that the Diffie–Hellman protocol on its own is not enough. For example how does Alice know who she is agreeing a key with? Is it Bob or Eve?

2. Digital Signature Schemes

One way around the man in the middle attack on the Diffie–Hellman protocol is for Alice to sign her message to Bob and Bob to sign his message to Alice. In that way both parties know who they are talking to. Signatures are an important concept of public key cryptography, they also were invented by Diffie and Hellman in the same 1976 paper, but the first practical system was due to Rivest, Shamir and Adleman.

The basic idea behind public key signatures is as follows:

$$\begin{aligned} \text{Message} + \text{Alice's private key} &= \text{Signature}, \\ \text{Message} + \text{Signature} + \text{Alice's public key} &= \text{YES/NO}. \end{aligned}$$

The above is called a signature scheme with appendix, since the signature is appended to the message before transmission, the message needs to be input into the signature verification procedure. Another variant is the signature scheme with message recovery, where the message is output by the signature verification procedure, as described in

$$\text{Message} + \text{Alice's private key} = \text{Signature},$$
$$\text{Signature} + \text{Alice's public key} = \text{YES/NO} + \text{Message}.$$

The main idea is that only Alice can sign a message, which could only come from her since only Alice has access to the private key. On the other hand anyone can verify Alice's signature, since everyone can have access to her public key.

The main problem is how are the public keys to be trusted? How do you know a certain public key is associated to a given entity? You may think a public key belongs to Alice, but it may belong to Eve. Eve can therefore sign cheques etc., and you would think they come from Alice. We seem to have the same key management problem as in symmetric systems, albeit now the problem is not one of keeping the keys secret, but making sure they are authentic. We shall return to this problem later.

A digital signature scheme consists more formally of two transformations:

- a secret signing transform S,
- a public verification transform V.

In the following discussion, we assume a signature with message recovery. For an appendix based scheme a simple change to the following will suffice.

Alice, sending a message m, calculates

$$s = S(m)$$

and then transmits s, where s is the digital signature on the message m. Note, we are not interested in keeping the message secret here, since we are only interested in knowing who it comes from. If confidentiality of the message is important then the signature s could be encrypted using, for example, the public key of the receiver.

The receiver of the signature s applies the public verification transform V to s. The output is then the message m and a bit v. The bit v indicates valid or invalid, i.e. whether the signature is good or not. If v is valid the recipient gets a guarantee of three important security properties:

- message integrity – the message has not been altered in transit,
- message origin – the message was sent by Alice,
- non-repudiation – Alice cannot claim she did not send the message.

Note, the first two of these properties are also provided by message authentication codes, MACs. However, the last property of non-repudiation is not provided by MACs and has important applications in e-commerce. To see why non-repudiation is so important, consider what would happen if you could sign a cheque and then say you did not sign it.

The RSA encryption algorithm is particularly interesting since it can be used directly as a signature algorithm with message recovery.

- The sender applies the RSA decryption transform to generate the signature, by taking the message and raising it to the private exponent d

$$s = m^d \pmod{N}.$$

- The receiver then applies the RSA encryption transform to recover the original message

$$m = s^e \pmod{N}.$$

But this raises the question as to how do we check for validity of the signature? If the original message is in a natural language such as English then one can verify that the extracted message is also in the same natural language. But this is not a solution for all possible messages. Hence one needs to add redundancy to the message.

One way of doing this is as follows. Suppose the message D is t bits long and the RSA modulus N is k bits long, with $t < k - 32$. We first pad D to the right by zeros to produce a string of length a multiple of eight. We then add $(k - t)/8$ bytes to the left of D to produce a byte-string

$$m = 00\|01\|FF\|FF \ldots \|FF\|00\|D.$$

The signature is then computed via

$$m^d \pmod{N}.$$

When verifying the signature we ensure that the recovered value of m has the correct padding.

But not all messages will be so short so as to fit into the above method. Hence, naively to apply the RSA signature algorithm to a long message m we need to break it into blocks and sign each block in turn. This is very time consuming for long messages. Worse than this, we must add serial numbers and more redundancy to each message otherwise an attacker could delete parts of the long message without us knowing, just as happened when encrypting using a block cipher in ECB Mode. This problem arises because our signature model is one giving message recovery, i.e. the message is recovered from the signature and the verification process. If we used a system using a signature scheme with appendix then we could produce a hash of the message to be signed and then just sign the hash.

3. The Use of Hash Functions In Signature Schemes

Using a cryptographic hash function h, such as those described in Chapter 10, it is possible to make RSA into a signature scheme without message recovery, which is much more efficient for long messages.

Suppose we are given a long message m for signing, we first compute $h(m)$ and then apply the RSA signing transform to $h(m)$, i.e. the signature is given by

$$s = h(m)^d \pmod{N}.$$

The signature and message are then transmitted together as the pair (m, s). Verifying a message/signature pair (m, s) generated using a hash function involves three steps.

- 'Encrypt' s using the RSA encryption function to recover h', i.e.

$$h' = s^e \pmod{N}.$$

- Compute $h(m)$ from m.
- Check whether $h' = h(m)$. If they agree accept the signature as valid, otherwise the signature should be rejected.

Actually in practice one also needs padding, as a hash function usually does not have output the whole of the integers modulo N. You could use the padding scheme given earlier when we discussed RSA with message recovery.

Recall that a cryptographic hash function needs to satisfy the following three properties:

(1) **Preimage Resistant:** It should be hard to find a message with a given hash value.
(2) **Collision Resistant:** It should be hard to find two messages with the same hash value.
(3) **Second Preimage Resistant:** Given one message it should be hard to find another message with the same hash value.

So why do we need to use a hash function which has these properties within the above signature scheme? We shall address these issues below:

3.1. Requirement for preimage resistance. The one-way property stops a cryptanalyst from cooking up a message with a given signature. For example, suppose we are using the RSA scheme with appendix just described but with a hash function which does not have the one-way property. We then have the following attack.

- Eve computes
$$h' = r^e \pmod{N}$$
for some random integer r.
- Eve also computes the pre-image of h' under h (recall we are assuming that h does not have the one-way property) i.e. Eve computes
$$m = h^{-1}(h').$$

Eve now has your signature (m, r) on the message m. Such a forgery is called an existential forgery in that the attacker may not have any control over the contents of the message on which they have obtained a digital signature.

3.2. Requirement for collision resistance. This is needed to avoid the following attack, which is performed by the legitimate signer.

- The signer chooses two messages m and m' with $h(m) = h(m')$.
- They sign m and output the signature (m, s).
- Later they repudiate this signature, saying it was really a signature on the message m'.

As a concrete example one could have that m is an electronic cheque for 1000 euros whilst m' is an electronic cheque for 10 euros.

3.3. Requirement for second preimage resistance. This property is needed to stop the following attack.

- An attacker obtains your signature (m, s) on a message m.
- The attacker finds another message m' with $h(m') = h(m)$.
- The attacker now has your signature (m', s) on the message m'.

Note, the security of any signature scheme which uses a cryptographic hash function, depends both on the security of the underlying hard mathematical problem, such as factoring or the discrete logarithm problem, and the security of the underlying hash function.

4. The Digital Signature Algorithm

We have already presented one digital signature scheme RSA. You may ask why do we need another one?

- What if someone breaks the RSA algorithm or finds that factoring is easy?
- RSA is not suited to some applications since signature generation is a very costly operation.
- RSA signatures are very large, some applications require smaller signature footprints.

One algorithm which addresses all of these concerns is the Digital Signature Algorithm, or DSA. One sometimes sees this referred to as the DSS, or Digital Signature Standard. Although originally designed to work in the group $\mathbb{F}_p^*$, where p is a large prime, it is now common to see it used using elliptic curves, in which case it is called EC-DSA. The elliptic curve variants of DSA run very fast and have smaller footprints and key sizes than almost all other signature algorithms.

We shall first describe the basic DSA algorithm as it applies to finite fields. In this variant the security is based on the difficulty of solving the discrete logarithm problem in the field $\mathbb{F}_p$.

DSA is a signature with appendix algorithm and the signature produced consists of two 160-bit integers r and s. The integer r is a function of a 160-bit random number k called the ephemeral key which changes with every message. The integer s is a function of

- the message,
- the signer's private key x,
- the integer r,
- the ephemeral key k.

Just as with the ElGamal encryption algorithm there are a number of domain parameters which are usually shared amongst a number of users. The DSA domain parameters are all public information and are much like those found in the ElGamal encryption algorithm. First a 160-bit prime number q is chosen, one then selects a large prime number p such that

- p has between 512 and 2048 bits,
- q divides $p-1$.

Finally we generate a random integer h less than p and compute

$$g = h^{(p-1)/q}.$$

If $g = 1$ then we pick a new value of h until we obtain $g \neq 1$. This ensures that g is an element of order q in the group $\mathbb{F}_p^*$, i.e.

$$g^q = 1 \pmod{p}.$$

After having decided on the domain parameters (p, q, g), each user generates their own private signing key x such that

$$0 < x < q.$$

The associated public key is y where

$$y = g^x \pmod{p}.$$

Notice that key generation for each user is much simpler than with RSA, since we only require a single modular exponentiation to generate the public key.

To sign a message m the user performs the following steps:

- Compute the hash value $h = H(m)$.
- Choose a random ephemeral key, $0 < k < q$.
- Compute
$$r = (g^k \pmod{p}) \pmod{q}.$$
- Compute
$$s = (h + xr)/k \pmod{q}.$$

The signature on m is then the pair (r, s), notice that this signature is therefore around 320 bits long.

To verify the signature (r, s) on the message m the verifier performs the following steps.

- Compute the hash value $h = H(m)$.
- $a = h/s \pmod{q}$.
- $b = r/s \pmod{q}$.
- Compute, where y is the public key of the sender,
$$v = (g^a y^b \pmod{p}) \pmod{q}.$$
- Accept the signature if and only if $v = r$.

As a baby example of DSA consider the following domain parameters

$$q = 13,\ p = 4q + 1 = 53 \text{ and } g = 16.$$

Suppose the public/private key pair of the user is given by $x = 3$ and

$$y = g^3 \pmod{p} = 15.$$

Now, if we wish to sign a message which has hash value $h = 5$, we first generate the ephemeral secret key $k = 2$ and then compute

$$r = (g^k \pmod{p}) \pmod{q} = 5,$$
$$s = (h + xr)/k \pmod{q} = 10.$$

To verify this signature the recipient computes

$$a = h/s \pmod{q} = 7,$$
$$b = r/s \pmod{q} = 7,$$
$$v = (g^a y^b \pmod{p}) \pmod{q} = 5.$$

Note $v = r$ and so the signature verifies correctly.

The DSA algorithm uses the subgroup of $\mathbb{F}_p^*$ of order q which is generated by g. Hence the discrete logarithm problem really is in the cyclic group $\langle g \rangle$ of order q. For security we insisted that we have

- $p > 2^{512}$, although $p > 2^{1024}$ may be more prudent, to avoid attacks via the Number Field Sieve,
- $q > 2^{160}$ to avoid attacks via the Baby-Step/Giant-Step method.

Hence, to achieve the rough equivalent of 80 bits of DES strength we need to operate on integers of roughly 1024 bits in length. This makes DSA slower even than RSA, since the DSA operation is more complicated than RSA. The verification operation in RSA requires only one exponentiation modulo a 1024-bit number, and even that is an exponentiation by a small number. For DSA, verification requires two exponentiations modulo a 1024-bit number, rather than one as in RSA. In addition the signing operation for DSA is more complicated due to the need to compute the value of s.

The main problem is that the DSA algorithm really only requires to work in a finite abelian group of size 2^{160}, but since the integers modulo p are susceptible to an attack from the Number Field Sieve we are required to work with group elements of 1024 bits in size. This produces a significant performance penalty.

Luckily we can generalize DSA to an arbitrary finite abelian group in which the discrete logarithm problem is hard. We can then use a group which provides a harder instance of the discrete logarithm problem, for example the group of points on an elliptic curve over a finite field.

We write $G = \langle g \rangle$ for a group generated by g, we assume that

- g has prime order $q > 2^{160}$,
- the discrete logarithm problem with respect to g is hard,
- there is a public function f such that

$$f : G \longrightarrow \mathbb{Z}/q\mathbb{Z}.$$

We summarize the different choices between DSA and EC-DSA in the following table:

Quantity	DSA	EC-DSA
G	$\langle g \rangle < \mathbb{F}_p^*$	$\langle P \rangle < E(\mathbb{F}_p)$
g	$g \in \mathbb{F}_p^*$	$P \in E(\mathbb{F}_p)$
y	g^x	$[x]P$
f	$\cdot \pmod{q}$	x-coord$(P) \pmod{q}$

For this generalized form of DSA each user again generates a secret signing key, x. The public key is again give by y where

$$y = g^x.$$

Signatures are computed via the steps

- Compute the hash value $h = H(m)$.
- Chooses a random ephemeral key, $0 < k < q$.
- Compute
$$r = f(g^k).$$
- Compute
$$s = (h + xr)/k \pmod{q}.$$

The signature on m is then the pair (r, s).

To verify the signature (r, s) on the message m the verifier performs the following steps.

- Compute the hash value $h = H(m)$.
- $a = h/s \pmod{q}$.
- $b = r/s \pmod{q}$.
- Compute, where y is the public key of the sender,
$$v = f(g^a y^b).$$
- Accept the signature if and only if $v = r$.

You should compare this signature and verification algorithm with that given earlier for DSA and spot where they differ. When used for EC-DSA the above generalization is written additively.

As a baby example of EC-DSA take the following elliptic curve

$$Y^2 = X^3 + X + 3,$$

over the field $\mathbb{F}_{199}$. The number of elements in $E(\mathbb{F}_{199})$ is equal to $q = 197$ which is a prime, the elliptic curve group is therefore cyclic and as a generator we can take

$$P = (1, 76).$$

As a private key let us take $x = 29$, and so the associated public key is given by

$$Y = [x]P = [29](1, 76) = (113, 191).$$

Suppose the holder of this public key wishes to sign a message with hash value $H(m)$ equal to 68. They first produce a random ephemeral key, which we shall take to be $k = 153$ and compute

$$\begin{aligned} r &= x\text{-coord}\left([k]P\right) \\ &= x\text{-coord}\left([153](1, 76)\right) \\ &= x\text{-coord}\left((185, 35)\right) \\ &= 185. \end{aligned}$$

Now they compute

$$\begin{aligned} s &= (H(m) + x \cdot r)/k \pmod{q} \\ &= (68 + 29 \cdot 185)/153 \pmod{197} \\ &= 78. \end{aligned}$$

The signature is then the pair $(r, s) = (185, 78)$.

To verify this signature we compute

$$\begin{aligned} a &= H(m)/s \pmod{q} \\ &= 68/78 \pmod{197} \\ &= 112, \\ b &= r/s \pmod{q} \\ &= 185/78 \pmod{197} \\ &= 15. \end{aligned}$$

We then compute

$$\begin{aligned} Z &= [a]P + [b]Y \\ &= [112](1,76) + [15](113,191) \\ &= (111,60) + (122,140) \\ &= (185,35). \end{aligned}$$

The signature now verifies since we have

$$r = 185 = x\text{-coord}(Z).$$

5. Schnorr Signatures

There are many variants of signature schemes based on discrete logarithms. A particularly interesting one is that of Schnorr signatures. We present the algorithm in the general case and allow the reader to work out the differences between the elliptic curve and finite field variants.

Suppose G is a public finite abelian group generated by an element g of prime order q. The public/private key pairs are just the same as in DSA, namely

- The private key is an integer x in the range $0 < x < q$.
- The public key is the element
$$y = g^x.$$

To sign a message m using the Schnorr signature algorithm we:

(1) Choose an ephemeral key k in the range $0 < k < q$.
(2) Compute the associated ephemeral public key
$$r = g^k.$$
(3) Compute $e = h(m\|r)$. Notice how the hash function depends both on the message and the ephemeral public key.
(4) Compute
$$s = k + x \cdot e \pmod{q}.$$

The signature is then given by the pair (e, s).

The verification step is very simple, we first compute

$$r = g^s y^{-e}.$$

The signature is accepted if and only if $e = h(m\|r)$.

As an example of Schnorr signatures in a finite field we take the domain parameters

$$q = 101, p = 607 \text{ and } g = 601.$$

As the public/private key pair we assume $x = 3$ and

$$y = g^x \pmod{p} = 391.$$

Then to sign a message we generate an ephemeral key $k = 65$ and compute

$$r = g^k \pmod{p} = 223.$$

We now need to compute the hash value

$$e = h(m\|r) \pmod{q}.$$

Let us assume that we compute $e = 93$, then the second component of the signature is given by

$$\begin{aligned} s &= k + x \cdot e \pmod{q} \\ &= 65 + 3 \cdot 93 \pmod{101} \\ &= 41. \end{aligned}$$

In a later chapter we shall see that Schnorr signatures are able to be proved to be secure, assuming that discrete logarithms are hard to compute, whereas no proof of security is known for DSA signatures.

Schnorr signatures have been suggested to be used for challenge response mechanisms in smart cards since the response part of the signature (the value of s) is particularly easy to evaluate since it only requires the computation of a single modular multiplication and a single modular addition. No matter what group we choose this final phase only requires arithmetic modulo a relatively small prime number.

To see how one uses Schnorr signatures in a challenge response situation we give the following scenario. A smart card wishes to authenticate you to a building or ATM machine. The card reader has a copy of your public key y, whilst the card has a copy of your private key x. Whilst you are walking around the card is generating commitments, which are ephemeral public keys of the form

$$r = g^k.$$

When you place your card into the card reader the card transmits to the reader the value of one of these precomputed commitments. The card reader then responds with a challenge message e. Your card then only needs to compute

$$s = k + xe \pmod{q},$$

and transmit it to the reader which then verifies the 'signature', by checking that

$$g^s = ry^e.$$

Notice that the only online computation needed by the card is the computation of the value of e and s, which are both easy to perform.

In more detail, if we let C denote the card and R denote the card reader then we have

$$\begin{aligned} &C \longrightarrow R : r = g^k, \\ &R \longrightarrow C : e, \\ &C \longrightarrow R : s = k + xe \pmod{q}. \end{aligned}$$

The point of the initial commitment is to stop either the challenge being concocted so as to reveal your private key, or your response being concocted so as to fool the reader. A three-phase protocol consisting of

$$\text{commitment} \longrightarrow \text{challenge} \longrightarrow \text{response}$$

is a common form of authentication protocols, we shall see more protocols of this nature when we discuss zero-knowledge proofs in Chapter 25.

6. Nyberg–Rueppel Signatures

What happens when we want to sign a general message which is itself quite short. It may turn out that the signature could be longer than the message. Recall that RSA can be used either as a scheme with appendix or as a scheme with message recovery. So far none of our discrete logarithm based schemes can be used with message recovery. We shall now give an example scheme which does have the message recovery property, called the Nyberg–Rueppel signature scheme, which is based on discrete logarithms in some public finite abelian group G.

All signature schemes with message recovery require a public redundancy function R. This function maps actual messages over to the data which is actually signed. This acts rather like a hash function does in the schemes based on signatures with appendix. However, unlike a hash function the redundancy function must be easy to invert. As a simple example we could take R to be the function

$$R : \begin{cases} \{0,1\}^{n/2} \longrightarrow \{0,1\}^n \\ m \longmapsto m\|m. \end{cases}$$

We assume that the codomain of R can be embedded into the group G. In our description we shall use the integers modulo p, i.e. $G = \mathbb{F}_p^*$, and as usual we assume that a large prime q divides $p-1$ and that g is a generator of the subgroup of order q.

Once again the public/private key pair is given as a discrete logarithm problem

$$(y = g^x, x).$$

Nyberg–Rueppel signatures are then produced as follows:

(1) Select a random $k \in \mathbb{Z}/q\mathbb{Z}$ and compute
$$r = g^k \pmod{p}.$$
(2) Compute
$$e = R(m) \cdot r \pmod{p}.$$
(3) Compute
$$s = x \cdot e + k \pmod{q}.$$

The signature is then the pair (e, s). From this pair, which is a group element and an integer modulo q, we need to

- verify that the signature comes from the user with public key y,
- recover the message m from the pair (e, s).

Verification for a Nyberg–Rueppel signature takes the signature (e, s) and the sender's public key $y = g^x$ and then computes

(1) Set
$$u_1 = g^s y^{-e} = g^{s-ex} = g^k \pmod{p}.$$
(2) Now compute
$$u_2 = e/u_1 \pmod{p}.$$
(3) Verify that u_2 lies in the range of the redundancy function, e.g. we must have
$$u_2 = R(m) = m\|m.$$
If this does not hold then reject the signature.
(4) Recover the message $m = R^{-1}(u_2)$ and accept the signature.

As an example we take the domain parameters

$$q = 101, p = 607 \text{ and } g = 601.$$

As the public/private key pair we assume $x = 3$ and

$$y = g^x \pmod{p} = 391.$$

To sign the message $m = 12$, where m must lie in $[0, \ldots, 15]$, we compute an ephemeral key $k = 45$ and

$$r = g^k \pmod{p} = 143.$$

Suppose

$$R(m) = m + 2^4 \cdot m$$

then we have $R(m) = 204$. We then compute

$$e = R(m) \cdot r \pmod{p} = 36,$$
$$s = x \cdot e + k \pmod{q} = 52.$$

The signature is then the pair $(e, s) = (36, 52)$. We now show how this signature is verified and the message recovered. We first compute

$$u_1 = g^s y^{-e} = 143.$$

Notice how the verifier has computed u_1 to be the same as the value of r computed by the signer. The verifier now computes

$$u_2 = e/u_1 \pmod{p} = 204.$$

The verifier now checks that $u_2 = 204$ is of the form

$$m + 2^4 m$$

for some value of $m \in [0, \ldots, 15]$. We see that u_2 is of this form and so the signature is valid. The message is then recovered by solving for m in

$$m + 2^4 m = 204,$$

from which we obtain $m = 12$.

7. Authenticated Key Agreement

Now we know how to perform digital signatures we can solve the problem with Diffie–Hellman key exchange. Recall that the man in the middle attack worked because each end did not know who they were talking to. We can now authenticate each end by requiring the parties to digitally sign their messages.

We will still obtain forward secrecy, since the long-term signing key is only used to provide authentication and is not used to perform a key transport operation.

We also have two choices of Diffie–Hellman protocol, namely one based on the discrete logarithms in a finite field DH and one based on elliptic curves EC-DH. There are also at least three possible signing algorithms RSA, DSA and EC-DSA. Assuming security sizes of 1024 bits for RSA, 1024 bits for the prime in DSA and 160 bits for the group order in both DSA and EC-DSA we obtain the following message sizes for our signed Diffie–Hellman protocol.

Algorithms	DH size	Signature size	Total size
DH+DSA	1024	320	1344
DH+RSA	1024	1024	2048
ECDH+RSA	160	1024	1184
ECDH+ECDSA	160	320	480

This is still an awfully large amount of overhead to simply agree what could be only a 128-bit session key.

To make the messages smaller Menezes, Qu and Vanstone invented the following protocol, called the MQV protocol based on the DLOG problem in a group G generated by g. One can use this protocol either in finite fields or in elliptic curves to obtain authenticated key exchange with message size of

Protocol	Message size
DL-MQV	1024
EC-MQV	160

Thus the MQV protocol gives us a considerable saving on the earlier message sizes. The protocol works by assuming that both parties, Alice and Bob, generate first a long-term public/private key pair which we shall denote by

$$(A = g^a, a) \text{ and } (B = g^b, b).$$

We shall assume that Bob knows that A is the authentic public key belonging to Alice and that Alice knows that B is the authentic public key belonging to Bob. This authentication of the public keys can be ensured by using some form of public key certification, described in a later chapter.

Assume Alice and Bob now want to agree on a secret session key to which they both contribute a random nonce. The use of the nonces provides them with forward secrecy and means that neither party has to trust the other in producing their session keys. So Alice and Bob now generate a public/private ephemeral key pair each

$$(C = g^c, c) \text{ and } (D = g^d, d).$$

They then exchange C and D. These are the only message flows in the MQV protocol, namely

$$\begin{aligned} \text{Alice} &\longrightarrow \text{Bob} : g^c, \\ \text{Bob} &\longrightarrow \text{Alice} : g^d. \end{aligned}$$

Hence, to some extent this looks like a standard Diffie–Hellman protocol with no signing. However, the trick is that the final session key will also depend on the long-term public keys A and B.

Assume you are Alice, so you know

$$A, B, C, D, a \text{ and } c.$$

Let l denote half the bit size of the order of the group G, for example if we are using a group with order $q \approx 2^{160}$ then we set $l = 160/2 = 80$. To determine the session key, Alice now computes

(1) Convert C to an integer i.
(2) Put $s_A = (i \pmod{2^l}) + 2^l$.
(3) Convert D to an integer j.
(4) Put $t_A = (j \pmod{2^l}) + 2^l$.
(5) Put $h_A = c + s_A a$.
(6) Put $P_A = (DB^{t_A})^{h_A}$.

Bob runs the same protocol but with the roles of the public and private keys swapped around in the obvious manner, namely

(1) Convert D to an integer i.
(2) Put $s_B = (i \pmod{2^l}) + 2^l$.
(3) Convert C to an integer j.
(4) Put $t_B = (j \pmod{2^l}) + 2^l$.
(5) Put $h_B = d + s_B b$.
(6) Put $P_B = (CA^{t_B})^{h_B}$.

Then $P_A = P_B$ is the shared secret. To see why the P_A computed by Alice and the P_B computed by Bob are the same we notice that the s_A and t_A seen by Alice, are swapped when seen by Bob, i.e. $s_A = t_B$ and $s_B = t_A$. Setting $\log(P)$ to be the discrete logarithm of P to the base g, we see

that

$$\begin{aligned}
\log(P_A) &= \log\left((DB^{t_A})^{h_A}\right) \\
&= (d + bt_A)h_A \\
&= d(c + s_A a) + bt_A(c + s_A a) \\
&= d(c + t_B a) + bs_B(c + t_B a) \\
&= c(d + s_B b) + at_B(d + s_B b) \\
&= (c + at_B)h_B \\
&= \log\left((CA^{t_B})^{h_B}\right) \\
&= \log(P_B).
\end{aligned}$$

Chapter Summary

- Diffie–Hellman key exchange can be used for two parties to agree on a secret key over an insecure channel. However, Diffie–Hellman is susceptible to the man in the middle attack and so requires some form of authentication of the communicating parties.
- Digital signatures provide authentication for both long-term and short-term purposes. They come in two variants either with message recovery or as a signature with appendix.
- The RSA encryption algorithm can be used in reverse to produce a public key signature scheme, but one needs to combine the RSA algorithm with a hash algorithm to obtain security for both short and long messages.
- DSA is a signature algorithm based on discrete logarithms, it has reduced bandwidth compared with RSA but is slower. EC-DSA is the elliptic curve variant of DSA, it also has the benefit of reduced bandwidth compared to DSA, but is more efficient than DSA.
- Other discrete logarithm based signature algorithms exist, all with different properties. Two we have looked at are Schnorr signatures and Nyberg–Rueppel signatures.
- Another way of using a key exchange scheme, without the need for digital signatures, is to use the MQV system. This has very small bandwidth requirements. It obtains implicit authentication of the agreed key, by combining the ephemeral exchanged key with the long-term static public key of each user, so as to obtain a new session key.

Further Reading

Details on more esoteric signature schemes such as one-time signatures, fail-stop signatures and undeniable signatures can be found in the books by Stinson and Schneier. These are also good places to look for further details about hash functions and message authentication codes, although by far the best reference in this area is *HAC*.

B. Schneier. *Applied Cryptography.* Wiley, 1996.

D. Stinson. *Cryptography: Theory and Practice.* CRC Press, 1995.

CHAPTER 15

Implementation Issues

Chapter Goals

- To show how exponentiation algorithms are implemented.
- To explain how modular arithmetic can be implemented efficiently on large numbers.
- To show how certain tricks can be used to speed up RSA and DSA operations.
- To show how finite fields of characteristic two can be implemented efficiently.

1. Introduction

In this chapter we examine how one actually implements cryptographic operations. We shall mainly be concerned with public key operations since those are the most complex to implement. For example, in RSA or DSA we have to perform a modular exponentiation with respect to a modulus of a thousand or more bits. This means we need to understand the implementation issues involved with both modular arithmetic and exponentiation algorithms.

There is another reason to focus on public key algorithms rather than private key ones: in general public key schemes run much slower than symmetric schemes. In fact they can be so slow that their use can seriously slow down networks and web servers. Hence, efficient implementation is crucial unless one is willing to pay a large performance penalty.

Since RSA is the easiest system to understand we will concentrate on this, although where special techniques exist for other schemes we will mention these as well. The chapter focuses on algorithms used in software, for hardware based algorithms one often uses different techniques entirely, but these alternative techniques are related to those used in software, so an understanding of software techniques is important.

2. Exponentiation Algorithms

So far in this book we have assumed that computing

$$a^b \pmod{c}$$

is an easy operation. We need this operation both in RSA and in systems based on discrete logarithms such as ElGamal encryption and DSA. In this section we concentrate on the exponentiation algorithms and assume that we can perform modular arithmetic efficiently. In a later section we shall discuss how to perform modular arithmetic.

As we have already stressed, the main operation in RSA and DSA is modular exponentiation

$$M = C^d \pmod{N}.$$

Firstly note it does not make sense to perform this via

- compute $R = C^d$,
- then compute $R \pmod{N}$.

To see this, consider

$$123^5 \pmod{511} = 28\,153\,056\,843 \pmod{511} = 359.$$

With this naive method one obtains a huge intermediate result, in our small case above this is

$$28\,153\,056\,843.$$

But in a real 1024-bit RSA multiplication this intermediate result would be in general

$$2^{1024} \cdot 1024$$

bits long. Such a number requires 10^{301} gigabytes simply to write down.

To stop this explosion in the size of any intermediate results we use the fact that we are working modulo N. But even here one needs to be careful, a naive algorithm would compute the above example by computing

$$\begin{aligned} x &= 123, \\ x^2 &= x \times x \pmod{511} = 310, \\ x^3 &= x \times x^2 \pmod{511} = 316, \\ x^4 &= x \times x^3 \pmod{511} = 32, \\ x^5 &= x \times x^4 \pmod{511} = 359. \end{aligned}$$

This requires four modular multiplications, which seems fine for our small example. But for a general RSA exponentiation by a 1024-bit exponent using this method would require around 2^{1024} modular multiplications. If each such multiplication could be done in under one millionth of a second we would still require 10^{294} years to perform an RSA decryption operation.

However, it is easy to see that, even in our small example, we can reduce the number of required multiplications by being a little more clever:

$$\begin{aligned} x &= 123, \\ x^2 &= x \times x \pmod{511} = 310, \\ x^4 &= x^2 \times x^2 \pmod{511} = 32, \\ x^5 &= x \times x^4 \pmod{511} = 359. \end{aligned}$$

Which only requires three modular multiplications rather than the previous four. To understand why we only require three modular multiplications notice that the exponent 5 has binary representation $0b101$ and so

- has bit length $t = 3$,
- has Hamming weight $h = 2$.

In the above example we required $1 = (h - 1)$ general multiplications and $2 = (t - 1)$ squarings. This fact holds in general, in that a modular exponentiation can be performed using

- $(h - 1)$ multiplications,
- $(t - 1)$ squarings,

where t is the bit length of the exponent and h is the Hamming weight. The average Hamming weight of an integer is $t/2$ so the number of multiplications and squarings is on average

$$t + t/2 - 1.$$

For a 1024-bit RSA modulus this means that the average number of modular multiplications needed to perform exponentiation by a 1024-bit exponent is at most 2048 and on average 1535.

The method used to achieve this improvement in performance is called the binary exponentiation method. This is because it works by reading each bit of the binary representation of the exponent

in turn, starting with the least significant bit and working up to the most significant bit. Algorithm 15.1 explains the method by computing

$$y = x^d \pmod{n}.$$

Algorithm 15.1: Binary exponentiation : Right-to-Left variant

$y = 1$
while $d \neq 0$ **do**
  if $(d \mod 2) \neq= 0$ **then**
    $y = (y \cdot x) \mod n$
    $d = d - 1$
  end
  $d = d/2$
  $x = (x \cdot x) \mod n$
end

The above binary exponentiation algorithm has a number of different names, some authors call it the *square and multiply* algorithm, since it proceeds by a sequence of squarings and multiplications, other authors call it the *indian exponentiation* algorithm. The above algorithm is called a *right to left* exponentiation algorithm since it processes the bits of d from the least significant bit up to the most significant bit.

Most of the time it is faster to perform a squaring operation than a general multiplication. Hence to reduce time even more one tries to reduce the total number of modular multiplications even further. This is done using window techniques which trade off precomputations (i.e. storage) against the time in the main loop.

To understand window methods better we first examine the binary exponentiation method again. But this time instead of a right to left variant, we process the exponent from the most significant bit first, thus producing a *left to right* binary exponentiation algorithm, see Algorithm 15.2. Again we assume we wish to compute

$$y = x^d \pmod{n}.$$

We first give a notation for the binary representation of the exponent

$$d = \sum_{i=0}^{t} d_i 2^i,$$

where $d_i \in \{0, 1\}$.

Algorithm 15.2: Binary exponentiation : Left-to-Right variant

$y = 1$
for $i = t$ **downto** 0 **do**
  $y = (y \cdot y) \mod n$
  if $d_i = 1$ **then** $y = (y \cdot x) \mod n$
end

The above algorithm processes a single bit of the exponent on every iteration of the loop. Again the number of squarings is equal to t and the expected number of multiplications is equal to $t/2$.

In a window method we process w bits of the exponent at a time, as in Algorithm 15.3. We first precompute a table

$$x_i = x^i \pmod{n} \text{ for } i = 0, \ldots, 2^w - 1.$$

Then we write our exponent out, but this time taking w bits at a time,

$$d = \sum_{i=0}^{t/w} d_i 2^{iw},$$

where $d_i \in \{0, 1, 2, \ldots, 2^w - 1\}$.

Algorithm 15.3: Window exponentiation method

```
y = 1
for i = t/w downto 0 do
    for j = 0 to w − 1 do
        y = (y · y) mod n
    end
    j = d_i
    y = (y · x_j) mod n
end
```

Let us see this algorithm in action by computing

$$y = x^{215} \pmod{n}$$

with a window width of $w = 3$. We compute the d_i as

$$215 = 3 \cdot 2^6 + 2 \cdot 2^3 + 7.$$

Hence, our iteration to compute $x^{215} \pmod{n}$ computes in order

$$\begin{aligned}
y &= 1, \\
y &= y \cdot x^3 = x^3, \\
y &= y^8 = x^{24}, \\
y &= y \cdot x^2 = x^{26}, \\
y &= y^8 = y^{208}, \\
y &= y \cdot x^7 = x^{215}.
\end{aligned}$$

With a window method as above, we still perform t squarings but the number of multiplications reduces to t/w on average. One can do even better by adopting a sliding window method, where we now encode our exponent as

$$d = \sum_{i=0}^{l} d_i 2^{e_i}$$

where $d_i \in \{1, 3, 5, \ldots, 2^w - 1\}$ and $e_{i+1} - e_i \geq w$. By choosing only odd values for d_i and having a variable window width we achieve both decreased storage for the precomputed values and improved efficiency. After precomputing $x_i = x^i$ for $i = 1, 3, 5, \ldots, 2^w - 1$, we execute Algorithm 15.4.

The number of squarings remains again at t, but now the number of multiplications reduces to l, which is about $t/(w + 1)$ on average. In our example of computing $y = x^{215} \pmod{n}$ we have

$$215 = 2^7 + 5 \cdot 2^4 + 7,$$

Algorithm 15.4: Sliding window exponentiation

$y = 1$
for $i = l$ **downto** 0 **do**
  for $j = 0$ **to** $e_{i+1} - e_i - 1$ **do** $y = (y \cdot y) \mod n$
  $j = d_i$
  $y = (y \cdot x_j) \mod n$
end
for $j = 0$ **to** $e_0 - 1$ **do** $y = (y \cdot y) \mod n$

and so we execute the steps

$$\begin{aligned} y &= 1, \\ y &= y \cdot x = x^1, \\ y &= y^8 = x^8, \\ y &= y \cdot x^5 = x^{13}, \\ y &= y^{16} = x^{208}, \\ y &= y \cdot x^7 = x^{215}. \end{aligned}$$

Notice that all of the above window algorithms apply to exponentiation in any abelian group and not just the integers modulo n. Hence, we can use these algorithms to compute

$$\alpha^d$$

in a finite field or to compute

$$[d]P$$

on an elliptic curve, in the latter case we call this point multiplication rather than exponentiation.

An advantage with elliptic curve variants is that negation comes for free, in that given P it is easy to compute $-P$. This leads to the use of signed binary and signed window methods. We only present the signed window method. We precompute

$$P_i = [i]P \text{ for } i = 1, 3, 5, \ldots, 2^{w-1} - 1,$$

which requires only half the storage of the equivalent sliding window method or one quarter of the storage of the equivalent standard window method. We now write our multiplicand d as

$$d = \sum_{i=0}^{l} d_i 2^{e_i}$$

where $d_i \in \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$. The signed sliding window method for elliptic curves is then given by Algorithm 15.5

3. Exponentiation in RSA

To speed up RSA exponentiation even more, a number of tricks are used which are special to RSA. The tricks used are different depending on whether we are performing an encryption/verification operation with the public key or a decryption/signing operation with the private key.

Algorithm 15.5: Signed sliding window method

$Q = 0$
for $i = l$ **downto** 0 **do**
  for $j = 0$ **to** $e_{i+1} - e_i - 1$ **do** $Q = [2]Q$
  $j = d_i$
  if $j > 0$ **then** $Q = Q + P_j$
  else $Q = Q - P_{-j}$
end
for $j = 0$ **to** $e_0 - 1$ **do** $Q = [2]Q$

3.1. RSA Encryption/Verification. As already remarked in earlier chapters one often uses a small public exponent, for example $e = 3, 17$ or $65\,537$. The reason for these particular values is that they have small Hamming weight, in fact the smallest possible for an RSA public key, namely two. This means that the binary method, or any other exponentiation algorithm, will require only one general multiplication, but it will still need k squarings where k is the bit size of the public exponent. For example

$$\begin{aligned} M^3 &= M^2 \times M, \\ M^{17} &= M^{16} \times M, \\ &= (((M^2)^2)^2)^2 \times M. \end{aligned}$$

3.2. RSA Decryption/Signing. In the case of RSA decryption or signing the exponent will be a general 1000-bit number. Hence, we need some way of speeding up the computation. Luckily, since we are considering a private key operation we have access to the private key, and hence the factorization of the modulus,

$$N = p \cdot q.$$

Supposing we are decrypting a message, we therefore wish to compute

$$M = C^d \pmod{N}.$$

We speed up the calculation by first computing M modulo p and q:

$$\begin{aligned} M_p &= C^d \pmod{p} = C^{d \pmod{p-1}} \pmod{p}, \\ M_q &= C^d \pmod{q} = C^{d \pmod{q-1}} \pmod{q}. \end{aligned}$$

Since p and q are 512-bit numbers, the above calculation requires two exponentiations modulo 512-bit moduli and 512-bit exponents. This is faster than a single exponentiation modulo a 1024-bit number with a 1024-bit exponent.

But we now need to recover M from M_p and M_q, which is done using the Chinese Remainder Theorem as follows: We compute $T = p^{-1} \pmod{q}$ and store it with the private key. The message M can then be recovered from M_p and M_q via

- $u = (M_q - M_p)T \pmod{q}$,
- $M = M_p + up$.

This is why in Chapter 11 we said that when you generate a private key it is best to store p and q even though they are not mathematically needed.

4. Exponentiation in DSA

Recall that in DSA verification one needs to compute

$$r = g^a y^b.$$

This can be accomplished by first computing g^a and then y^b and then multiplying the results together. However, often it is easier to perform the two exponentiations simultaneously. There are a number of techniques to accomplish this, using various forms of window techniques etc. But all are essentially based on the following idea, called Shamir's trick.

We first compute the following look-up table

$$G_i = g^{i_0} y^{i_1}$$

where $i = (i_1, i_0)$ is the binary representation of i, for $i = 0, 1, 2, 3$. We then compute an exponent array from the two exponents a and b. This is a 2 by t array, where t is the maximum bit length of a and b. The rows of this array are the binary representation of the exponents a and b. We then let I_j, for $j = 1, \ldots, t$, denote the integers whose binary representation is given by the columns of this array. The exponentiation is then computed by setting $r = 1$ and computing

$$r = r^2 \cdot G_{I_j}$$

for $j = 1$ to t.

As an example suppose we wish to compute

$$r = g^{11} y^7,$$

hence we have $t = 4$. We precompute

$$G_0 = 1,\ G_1 = g,\ G_2 = y,\ G_3 = g \cdot y.$$

Since the binary representation of 11 and 7 is given by 1011 and 111, our exponent array is given by

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

The integers I_j then become

$$I_1 = 1,\ I_2 = 2,\ I_3 = 3,\ I_4 = 3.$$

Hence, the four steps of our algorithm become

$$\begin{aligned} r &= G_1 = g, \\ r &= r^2 \cdot G_2 = g^2 \cdot y, \\ r &= r^2 \cdot G_3 = (g^4 \cdot y^2) \cdot (g \cdot y) = g^5 \cdot y^3, \\ r &= r^2 \cdot G_3 = (g^{10} \cdot y^6) \cdot (g \cdot y) = g^{11} \cdot y^7. \end{aligned}$$

Note, elliptic curve analogues of Shamir's trick and its variants can be made which make use of signed representations for the exponent. We do not give these here, but leave them for the interested reader to investigate.

5. Multi-precision Arithmetic

We shall now explain how to perform modular arithmetic on 1024-bit numbers. We show how this is accomplished using modern processors, and why naive algorithms are usually replaced with a special technique due to Montgomery.

In a cryptographic application it is common to focus on a fixed length for the integers in use, for example 1024 bits in an RSA/DSA implementation or 200 bits for an ECC implementation. This leads to different programming choices than when one implements a general purpose multi-precision arithmetic library. For example one no longer needs to worry so much about dynamic memory allocation, and one can now concentrate on particular performance enhancements for the integer sizes one is dealing with.

It is common to represent all integers in little-wordian format. This means that if a large integer is held in memory locations, $x_0, x_1, \ldots, x_n$, then x_0 is the least significant word and x_n is the most

significant word. For a 32-bit machine and 64-bit numbers we would represent x and y as $[x_0, x_1]$ and $[y_0, y_1]$ where

$$x = x_1 2^{32} + x_0,$$
$$y = y_1 2^{32} + y_0.$$

5.1. Addition. Most modern processors have a carry flag which is set by any overflow from an addition operation. Also most have a special instruction, usually called something like **addc**, which adds two integers together and adds on the contents of the carry flag. So if we wish to add our two 64-bit integers given earlier then we need to compute

$$z = x + y = z_2 2^{64} + z_1 2^{32} + z_0.$$

The values of z_0, z_1 and z_2 are then computed via

```
z0 <- add   x0,y0
z1 <- addc  x1,y1
z2 <- addc  0,0
```

Note that the value held in z_2 is at most one, so the value of z could be a 65-bit integer. The above technique for adding two 64-bit integers can clearly be scaled to adding integers of any fixed length, and can also be made to work for subtraction of large integers.

5.2. School-book Multiplication. We now turn to the next simplest arithmetic operation, after addition and subtraction, namely multiplication. Notice that two 32 -bit words multiply together to form a 64-bit result, and so most modern processors have an instruction which will do this operation.

$$w_1 \cdot w_2 = (High, Low) = (H(w_1 \cdot w_2), L(w_1 \cdot w_2)).$$

When we use school-book long multiplication, for our two 64-bit numbers, we obtain something like

		x_1	x_0
	$\times$	y_1	y_0
		$H(x_0 \cdot y_0)$	$L(x_0 \cdot y_0)$
	$H(x_0 \cdot y_1)$	$L(x_0 \cdot y_1)$	
	$H(x_1 \cdot y_0)$	$L(x_1 \cdot y_0)$	
$H(x_1 \cdot y_1)$	$L(x_1 \cdot y_1)$		

Then we add up the four rows to get the answer, remembering we need to take care of the carries. This then becomes, for

$$z = x \cdot y,$$

something like the following pseudo-code

```
(z1,z0) <- mul x0,y0
(z3,z2) <- mul x1,y1
(h,l)   <- mul x1,y0
z1      <- add z1,l
z2      <- addc z2,h
z3      <- addc z3,0
(h,l)   <- mul x0,y1
z1      <- add z1,l
z2      <- addc z2,h
z3      <- addc z3,0
```

If n denotes the bit size of the integers we are operating on, the above technique for multiplying large integers together clearly requires $O(n^2)$ bit operations, whilst it requires $O(n)$ bit operations to add or subtract integers. It is a natural question as to whether one can multiply integers faster than $O(n^2)$.

5.3. Karatsuba Multiplication. One technique to speed up multiplication is called Karatsuba multiplication. Suppose we have two n-bit integers x and y that we wish to multiply. We write these integers as

$$x = x_0 + 2^{n/2}x_1,$$
$$y = y_0 + 2^{n/2}y_2,$$

where $0 \leq x_0, x_1, y_0, y_1 < 2^{n/2}$. We then multiply x and y by computing

$$A = x_0 \cdot y_0,$$
$$B = (x_0 + x_1) \cdot (y_0 + y_1),$$
$$C = x_1 \cdot y_1.$$

The product $x \cdot y$ is then given by

$$\begin{aligned} C2^n + (B - A - C)2^{n/2} + A &= x_1y_1 2^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0 \\ &= (x_0 + 2^{n/2}x_1) \cdot (y_0 + 2^{n/2}y_1) \\ &= x \cdot y. \end{aligned}$$

Hence, this multiplication technique to multiply two n-bit numbers requires three $n/2$-bit multiplications, two $n/2$-bit additions and three n-bit additions/subtractions. If we denote the cost of an n-bit multiplication by $M(n)$ and the cost of an n-bit addition/subtraction by $A(n)$ then this becomes

$$M(n) = 3M(n/2) + 2A(n/2) + 3A(n).$$

Now if we make the approximation that $A(n) \approx n$ then

$$M(n) \approx 3M(n/2) + 4n.$$

If the multiplication of the $n/2$-bit numbers is accomplished in a similar fashion then to obtain the final complexity of multiplication we need to solve the above recurrence relation to obtain

$$\begin{aligned} M(n) &\approx 9n^{\frac{\log(3)}{\log(2)}} \text{ as } n \longrightarrow \infty \\ &= 9n^{1.58}. \end{aligned}$$

So we obtain an algorithm with asymptotic complexity $O(n^{1.58})$. Karatsuba multiplication becomes faster than the $O(n^2)$ method for integers of sizes greater than a few hundred bits. However, one can do even better for very large integers since the fastest known multiplication algorithm takes time

$$O(n \log n \log \log n).$$

But neither this latter technique nor Karatsuba multiplication are used in many cryptographic applications. The reason for this will become apparent as we discuss integer division.

5.4. Division. After having looked at multiplication we are left with the division operation, which is the hardest of all the basic algorithms. Division is required in order to be able to compute the remainder on division, which is after all a basic operation in RSA. Given two large integers x and y we wish to be able to compute q and r such that

$$x = qy + r$$

where $0 \leq r < y$, such an operation is called a Euclidean division.

If we write our two integers x and y in the little-wordian format

$$x = (x_0, \ldots, x_n) \text{ and } y = (y_0, \ldots, y_t)$$

where the base for the representation is $b = 2^w$ then the Euclidean division can be performed by Algorithm 15.6. We let $t \ll_w v$ denote a large integer t shifted to the left by v words, in other words the result of multiplying t by b^v.

As one can see this is a complex operation, hence one should try and avoid divisions as much as possible.

5.5. Montgomery Arithmetic. That division is a complex operation means our cryptographic operations run very slowly if we use standard division operations as above. Recall that virtually all of our public key systems make use of arithmetic modulo another number. What we require is the ability to compute remainders (i.e. to perform modular arithmetic) without having to perform any costly division operations. This at first sight may seem a state of affairs which is impossible to reach, but it can be achieved using a special form of arithmetic called Montgomery arithmetic.

Montgomery arithmetic works by using an alternative representation of integers, called the Montgomery representation. Let us fix some notation, we let b denote 2 to the power of the word size of our computer, for example $b = 2^{32}$ or 2^{64}. To perform arithmetic modulo N we choose an integer R which satisfies

$$R = b^t > N.$$

Now instead of holding the value of the integer x in memory, we instead hold the value

$$x \cdot R \pmod{N}.$$

Again this is usually held in a little-wordian format. The value $x \cdot R \pmod{N}$ is called the Montgomery representation of the integer $x \pmod{N}$.

Adding two elements in Montgomery representation is easy. If

$$z = x + y \pmod{N}$$

then given $x \cdot R \pmod{N}$ and $y \cdot R \pmod{N}$ we need to compute $z \cdot R \pmod{N}$.

Let us take a simple example with

$$N = 1\,073\,741\,827,$$
$$b = R = 2^{32} = 4\,294\,967\,296.$$

The following is the map from the normal to Montgomery representation of the integers $1, 2$ and 3.

$$1 \longrightarrow 1 \cdot R \pmod{N} = 1\,073\,741\,815,$$
$$2 \longrightarrow 2 \cdot R \pmod{N} = 1\,073\,741\,803,$$
$$3 \longrightarrow 3 \cdot R \pmod{N} = 1\,073\,741\,791.$$

We can now verify that addition works since we have in the standard representation

$$1 + 2 = 3$$

whilst this is mirrored in the Montgomery representation as

$$1\,073\,741\,815 + 1\,073\,741\,803 = 1\,073\,741\,791 \pmod{N}.$$

Now we look at multiplication in Montgomery arithmetic. If we simply multiply two elements in Montgomery representation we will obtain

$$(xR) \cdot (yR) = xyR^2 \pmod{N}$$

but we want $xyR \pmod{N}$. Hence, we need to divide the result of the standard multiplication by R. Since R is a power of 2 we hope this should be easy.

The process of given y and computing

$$z = y/R \pmod{N}$$

Algorithm 15.6: Euclidean division algorithm

$r = x$

/* Cope with the trivial case */
if $t > n$ **then**
  $q = 0$
  return
end
$q = 0, s = 0$

/* Normalise the divisor */
while $y_t < b/2$ **do**
  $y = 2y$
  $r = 2r$
  $s = s + 1$
end
if $r_{n+1} \neq 0$ **then** $n = n + 1$

/* Get the msw of the quotient */
while $r \geq (y \ll_w (n - t))$ **do**
  $q_{n-t} = q_{n-t} + 1$
  $r = r - (y \ll_w n - t)$
end

/* Deal with the rest */
for $i = n$ **to** $t + 1$ **do**
  if $r_i = y_t$ **then** $q_{i-t-1} = b - 1$
  else $q_{i-t-1} =$floor$((r_i b + r_{i-1})/y_t)$
  if $t \neq 0$ **then** $h_m = y_t b + y_{t-1}$
  else $h_m = y_t b$
  $h = q_{i-t-1} h_m$
  if $i \neq 1$ **then** $l = r_i b^2 + r_{i-1} b + r_{i-2}$
  else $l = r_i b^2 + r_{i-1} b$
  while $h > l$ **do**
    $q[i - t - 1] = q[i - t - 1] - 1$
    $h = h - h_m$
  end
  $r = r - (q_{i-t-1} y) \ll_w (i - t - 1)$
  if $r < 0$ **then**
    $r = r + (y \ll_w i - t - 1)$
    $q_{i-t-1} = q_{i-t-1} - 1$
  end
end

/* Renormalise */
for $i = 0$ **to** $s - 1$ **do** $r = r/2$

Algorithm 15.7: Addition in Montgomery representation

$zR = xR + yR$
if $zR \geq N$ **then** $zR = zR - N$

given the earlier choice of R, is called Montgomery reduction. We first precompute the integer $q = 1/N \pmod R$, which is simple to perform with no divisions using the binary Euclidean algorithm. Then, performing a Montgomery reduction is done using Algorithm 15.8.

Algorithm 15.8: Montgomery reduction

$u = (-y \cdot q) \mod R$;
$z = (y + u \cdot N)/R$;
if $z \geq N$ **then** $z = z - N$;

Note that the reduction modulo R in the first line is easy, we compute $y \cdot q$ using standard algorithms, the reduction modulo R being achieved by truncating the result. This latter trick works since R is a power of b. The division by R in the second line can also be simply achieved, since $y + u \cdot N = 0 \pmod R$, we simply shift the result to the right by t words, again since $R = b^t$.

As an example we again take

$$N = 1\,073\,741\,827,$$
$$b = R = 2^{32} = 4\,294\,967\,296.$$

We wish to compute $2 \cdot 3$ in Montgomery representation. Recall

$$2 \longrightarrow 2 \cdot R \pmod N = 1\,073\,741\,803 = x,$$
$$3 \longrightarrow 3 \cdot R \pmod N = 1\,073\,741\,791 = y.$$

We then compute, using a standard multiplication algorithm that

$$w = x \cdot y = 1\,152\,921\,446\,624\,789\,173 = 2 \cdot 3 \cdot R^2.$$

We now need to pass this value of w into our technique for Montgomery reduction, so as to find the Montgomery representation of $x \cdot y$. We find

$$w = 1\,152\,921\,446\,624\,789\,173,$$
$$q = (1/N) \pmod R = 1\,789\,569\,707,$$
$$u = -w \cdot q \pmod R = 3\,221\,225\,241,$$
$$z = (w + u \cdot N)/R = 1\,073\,741\,755.$$

So the multiplication of x and y in Montgomery arithmetic should be

$$1\,073\,741\,755.$$

We can check that this is the correct value by computing

$$6 \cdot R \pmod N = 1\,073\,741\,755.$$

Hence, we see that Montgomery arithmetic allows us to add and multiply integers modulo an integer N without the need for costly division algorithms.

Our above method for Montgomery reduction requires two full multi-precision multiplications. So to multiply two numbers in Montgomery arithmetic we require three full multi-precision multiplications. If we are multiplying 1024-bit numbers, this means the intermediate results can grow to be 2048-bit numbers. We would like to do better, and we can.

Suppose y is given in little-wordian format

$$y = (y_0, y_1, \ldots, y_{2t-2}, y_{2t-1}).$$

Then a better way to perform Montgomery reduction is to first precompute

$$N' = -1/N \pmod{b}$$

which is easy and only requires operations on word-sized quantities, and then to execute Algorithm 15.9

Algorithm 15.9: Word oriented Montgomery reduction

$z = y$
for $i = 0$ **to** $t - 1$ **do**
  $u = (z_i \cdot N') \mod b$
  $z = z + u \cdot N$
  $z = z \cdot b$
end
$z = z/R$
if $z \geq N$ **then** $z = z - N$

Note, since we are reducing modulo b in the first line of the for loop we can execute this initial multiplication using a simple word multiplication algorithm. The second step of the for loop requires a shift by one word (to multiply by b) and a single *word* $\times$ *bigint* multiply. Hence, we have reduced the need for large intermediate results in the Montgomery reduction step.

We can also interleave the multiplication with the reduction to perform a single loop to produce

$$Z = XY/R \pmod{N}.$$

So if $X = xR$ and $Y = yR$ this will produce

$$Z = (xy)R.$$

This procedure is called Montgomery multiplication and allows us to perform a multiplication in Montgomery arithmetic without the need for larger integers, as in Algorithm 15.10.

Algorithm 15.10: Montgomery multiplication

$Z = 0$
for $i = 0$ **to** $t - 1$ **do**
  $u = ((z_0 + X_i \cdot Y_0) \cdot N') \mod b$
  $Z = (Z + X_i \cdot Y + u \cdot N)/b$
end
if $Z \geq N$ **then** $Z = Z - N$

Whilst Montgomery multiplication has complexity $O(n^2)$ as opposed to the $O(n^{1.58})$ of Karatsuba multiplication, it is still preferable to use Montgomery arithmetic since it deals more efficiently with modular arithmetic.

6. Finite Field Arithmetic

Apart from the integers modulo a large prime p the other type of finite field used in cryptography are those based on fields of characteristic two. These occur in the Rijndael algorithm and in certain elliptic curve systems. In Rijndael the field is so small that one can use look-up tables or special circuits to perform the basic arithmetic tasks, so in this section we shall concentrate on fields of large degree over $\mathbb{F}_2$, like those used with elliptic curves. In addition we shall concern ourselves with software implementations only. Fields of characteristic two can have special types of hardware implementations based on things called optimal normal bases, but we shall not concern ourselves with these.

Recall that to define a finite field of characteristic two we first pick an irreducible polynomial $f(x)$ over $\mathbb{F}_2$ of degree n. The field is defined to be

$$\mathbb{F}_{2^n} = \mathbb{F}_2[x]/f(x),$$

i.e. we look at binary polynomials modulo $f(x)$. Elements of this field are usually represented as bit strings, which represent a binary polynomial. For example the bit string

$$101010111$$

represents the polynomial

$$x^8 + x^6 + x^4 + x^2 + x + 1.$$

Addition and subtraction of elements in $\mathbb{F}_{2^n}$ is accomplished by simply performing a bitwise XOR between the two bitstrings. Hence, the difficult tasks are multiplication and division.

It turns out that division, although slower than multiplication, is easier to describe, so we start with division. To compute

$$\alpha/\beta,$$

where $\alpha, \beta \in \mathbb{F}_{2^n}$, we first compute

$$\beta^{-1}$$

and then perform the multiplication

$$\alpha \cdot \beta^{-1}.$$

So division is reduced to multiplication and the computation of β^{-1}. One way of computing β^{-1} is to use Lagrange's Theorem which tells us for $\beta \neq 0$ that we have

$$\beta^{2^n-1} = 1.$$

But this means that

$$\beta \cdot \beta^{2^n-2} = 1,$$

or in other words

$$\beta^{-1} = \beta^{2^n-2} = \beta^{2(2^{n-1}-1)}.$$

Another way of computing β^{-1} is to use the binary Euclidean algorithm. We take the polynomial f and the polynomial b which represents β and then perform Algorithm 15.11, which is a version of the binary Euclidean algorithm, where $\mathrm{lsb}(b)$ refers to the least significant bit of b (in other words the coefficient of x^0),

We now turn to the multiplication operation. Unlike the case of integers modulo N or p, where we use a special method of Montgomery arithmetic, in characteristic two we have the opportunity to choose a polynomial $f(x)$ which has 'nice' properties. Any irreducible polynomials of degree n can be used to implement the finite field $\mathbb{F}_{2^n}$, we just need to select the best one.

Almost always one chooses a value of $f(x)$ which is either a trinomial

$$f(x) = x^n + x^k + 1$$

Algorithm 15.11: Binary extended Euclidean algorithm for polynomials over $\mathbb{F}_2$

```
a = f
B = 0
D = 1
/* At least one of a and b now has a constant term on every
execution of the loop. */
while a ≠ 0 do
    while lsb(a) = 0 do
        a = a ≫ 1
        if lsb(B) ≠ 0 then B = B ⊕ f
        B = B ≫ 1
    end
    while lsb(b) = 0 do
        b = b ≫ 1
        if lsb(D) ≠ 0 then D = D ⊕ f
        D = D ≫ 1
    end
    /* Now both a and b have a constant term */
    if deg(a) ≥ deg(b) then
        a = a ⊕ b
        B = B ⊕ D
    else
        b = a ⊕ b
        D = D ⊕ B
    end
end
return D
```

or a pentanomial

$$f(x) = x^n + x^{k_3} + x^{k_2} + x^{k_1} + 1.$$

It turns out that for all fields of degree less than 10 000 we can always find such a trinomial or pentanomial to make the multiplication operation very efficient. Table 1 at the end of this chapter gives a list for all values of n between 2 and 500 of an example pentanomial or trinomial which defines the field $\mathbb{F}_{2^n}$. In all cases where a trinomial exists we give one, otherwise we present a pentanomial.

Now to perform a multiplication of α by β we first multiply the polynomials representing α and β together to form a polynomial $\gamma(x)$ of degree at most $2n-2$. Then we reduce this polynomial by taking the remainder on division by the polynomial $f(x)$.

We show how this remainder on division is efficiently performed for trinomials, and leave the pentanomial case for the reader. We write

$$\gamma(x) = \gamma_1(x)x^n + \gamma_0(x).$$

Hence, $\deg(\gamma_1(x)), \deg(\gamma_0(x)) \leq n-1$. We can then write

$$\gamma(x) \pmod{f(x)} = \gamma_0(x) + (x^k + 1)\gamma_1(x).$$

The right-hand side of this equation can be computed from the bit operations

$$\delta = \gamma_0 \oplus \gamma_1 \oplus (\gamma_1 \ll k).$$

Now δ, as a polynomial, will have degree at most $n-1+k$. So we need to carry out this procedure again by first writing

$$\delta(x) = \delta_1(x)x^n + \delta_0(x),$$

where $\deg(\delta_0(x)) \leq n-1$ and $\deg(\delta_1(x)) \leq k-1$. We then compute as before that γ is equivalent to

$$\delta_0 \oplus \delta_1 \oplus (\delta_1 \ll k).$$

This latter polynomial will have degree $\max(n-1, 2k-1)$, so if we may choose in our trinomial

$$k \leq n/2,$$

then Algorithm 15.12 will perform our division by remainder step. Let g denote the polynomial of degree $2n-2$ that we wish to reduce modulo f, where we assume a bit representation for these polynomials.

Algorithm 15.12: Reduction by a trinomial

$g_1 = g \gg n$
$g_0 = g[n-1 \ldots 0]$
$g = g_0 \oplus g_1 \oplus (g_1 \ll k)$
$g_1 = g \gg n$
$g_0 = g[n-1 \ldots 0]$
$g = g_0 \oplus g_1 \oplus (g_1 \ll k)$

So to complete our description of how to multiply elements in $\mathbb{F}_{2^n}$ we need to explain how to perform the multiplication of two binary polynomials of large degree $n-1$.

Again one can use a naive multiplication algorithm. Often however one uses a look-up table for polynomial multiplication of polynomials of degree less than eight, i.e. for operands which fit into one byte. Then multiplication of larger degree polynomials is reduced to multiplication of polynomials of degree less than eight by using a variant of the standard long multiplication algorithm from school. This algorithm will have complexity $O(n^2)$, where n is the degree of the polynomials involved.

Suppose we have a routine which uses a look-up table to multiply two binary polynomials of degree less than eight, returning a binary polynomial of degree less than sixteen. This function we denote by $Mult\_Tab(a, b)$ where a and b are 8-bit integers representing the input polynomials.

To perform a multiplication of two n-bit polynomials represented by two n-bit integers x and y we perform Algorithm 15.13, where $y \gg 8$ (resp. $y \ll 8$) represents shifting to the right (resp. left) by 8 bits.

Just as with integer multiplication one can use a divide and conquer technique based on Karatsuba multiplication, which again will have a complexity of $O(n^{1.58})$. Suppose the two polynomials we wish to multiply are given by

$$a = a_0 + x^{n/2}a_1,$$
$$b = b_0 + x^{n/2}b_1,$$

where a_0, a_1, b_0, b_1 are polynomials of degree less than $n/2$. We then multiply a and b by computing

$$A = a_0 \cdot b_0,$$
$$B = (a_0 + a_1) \cdot (b_0 + b_1),$$
$$C = a_1 \cdot b_1.$$

Algorithm 15.13: Multiplication of two n-bit polynomials over $\mathbb{F}_2$

$i = 0, a = 0$
while $x \neq 0$ **do**
  $u = y, j = 0$
  while $u \neq 0$ **do**
    $w = Mult\_Tab(x\&255, u\&255)$
    $w = w \ll (8(i + j))$
    $a = a \oplus w$
    $u = u \gg 8, j = j + 1$
  end
  $x = x \gg 8, i = i + 1$
end
return (a)

The product $a \cdot b$ is then given by

$$\begin{aligned} Cx^n + (B - A - C)x^{n/2} + A &= a_1b_1x^n + (a_1b_0 + a_0b_1)x^{n/2} + a_0b_0 \\ &= (a_0 + x^{n/2}a_1) \cdot (b_0 + x^{n/2}b_1) \\ &= a \cdot b. \end{aligned}$$

Again to multiply a_0 and b_0 etc. we use the Karatsuba multiplication method recursively. Once we reduce to the case of multiplying two polynomials of degree less than eight we resort to using our look-up table to perform the polynomial multiplication. Unlike the integer case we now find that Karatsuba multiplication is more efficient than the school-book method even for polynomials of quite small degree, say $n \approx 100$.

One should note that squaring polynomials in characteristic two is particularly easy. Suppose we have a polynomial

$$a = a_0 + a_1x + a_2x^2 + a_3x^3,$$

where $a_i = 0$ or 1. Then to square a we simply 'thin out' the coefficients as follows:

$$a^2 = a_0 + a_1x^2 + a_2x^4 + a_3x^6.$$

This means that squaring an element in a finite field of characteristic two is very fast compared with a multiplication operation.

Chapter Summary

- Modular exponentiation, or exponentiation in any group, can be computed using the binary exponentiation method. Often it is more efficient to use a window based method, or to use a signed exponentiation method in the case of elliptic curves.
- For RSA special optimizations are performed. In the case of the public exponent we choose one which is both small and has very low Hamming weight. For the exponentiation by the private exponent we use knowledge of the prime factorization of the modulus and the Chinese Remainder Theorem.

- For DSA verification there is a method based on simultaneous exponentiation which is often more efficient than performing two single exponentiations and then combining the result.
- Modular arithmetic is usually implemented using the technique of Montgomery representation. This allows us to avoid costly division operations by replacing the division with simple shift operations. This however is at the expense of using a non-standard representation for the numbers.
- Finite fields in characteristic two can also be implemented efficiently. But now the modular reduction operation can be made simple by choosing a special polynomial $f(x)$. Inversion is also particular simple using a variant of the binary Euclidean algorithm, although often inversion is still 3–10 times slower than multiplication.

Further Reading

The standard reference work for the type of algorithms considered in this chapter is Volume 2 of Knuth. A more gentle introduction can be found in the book by Bach and Shallit, whilst for more algorithms one should consult the book by Cohen. The first chapter of Cohen gives a number of lessons learnt in the development of the PARI/GP calculator which can be useful, whilst Bach and Shallit provides an extensive bibliography and associated commentary.

E. Bach and S. Shallit. *Algorithmic Number Theory, Volume 1: Efficient Algorithms*. MIT Press, 1996.

H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993.

D. Knuth. *The Art of Computing Programming, Volume 2 : Seminumerical Algorithms*. Addison-Wesley, 1975.