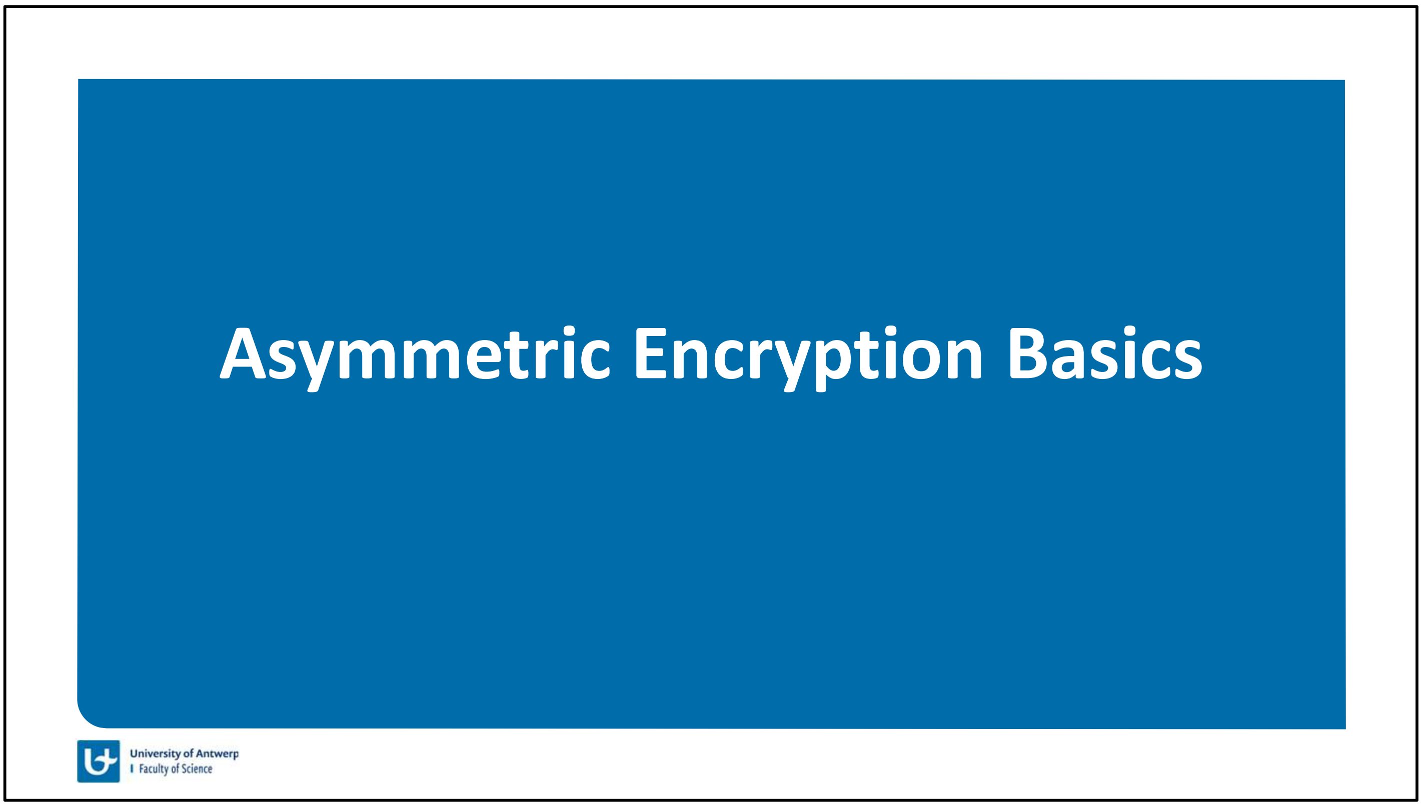
Computer and Network Security (2023-2024)

Part 4: Asymmetric Encryption

Jeroen Famaey
jeroen.famaey@uantwerpen.be

University of Antwerp
Faculty of Science



Asymmetric Encryption Basics

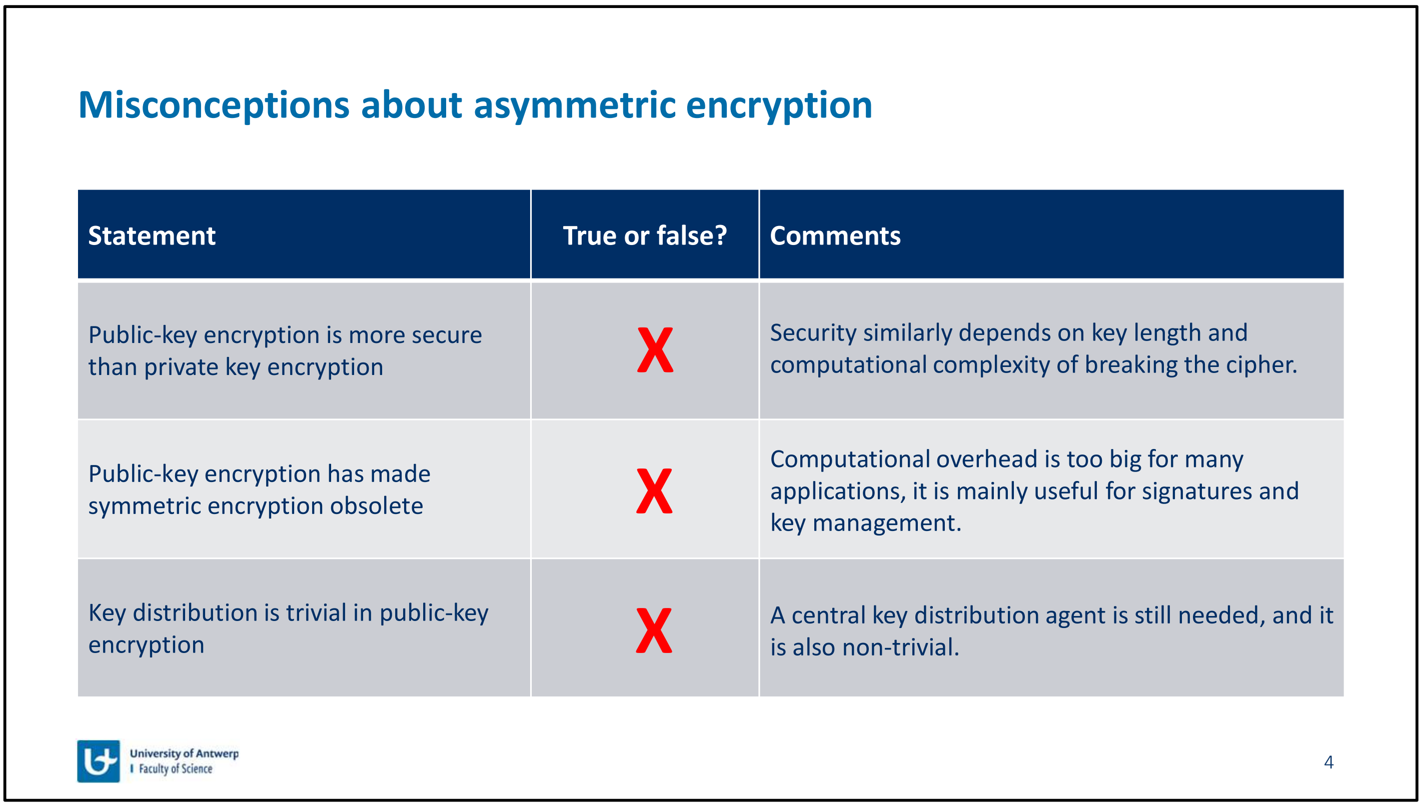
University of Antwerp
Faculty of Science

Some facts about asymmetric encryption

- Also known as **public key cryptography**, as it uses a public and private (i.e., secret) key
- Based on mathematical functions, not substitution and permutation
- Addresses two concerns with symmetric encryption
  - Secret keys need to be distributed using a trusted key distribution center
  - It does not support digital signatures
- Idea first made public by Diffie and Hellman in 1976

University of Antwerp
Faculty of Science

Public-key cryptography provides a radical departure from all that has gone before. For one thing, public-key algorithms are based on mathematical functions rather than on substitution and permutation. More important, public-key cryptography is asymmetric, involving the use of two separate keys, in contrast to symmetric encryption, which uses only one key. The use of two keys has profound consequences in the areas of confidentiality, key distribution, and authentication, as we shall see.



Misconceptions about asymmetric encryption

Statement	True or false?	Comments
Public-key encryption is more secure than private key encryption	X	Security similarly depends on key length and computational complexity of breaking the cipher.
Public-key encryption has made symmetric encryption obsolete	X	Computational overhead is too big for many applications, it is mainly useful for signatures and key management.
Key distribution is trivial in public-key encryption	X	A central key distribution agent is still needed, and it is also non-trivial.

University of Antwerp
Faculty of Science

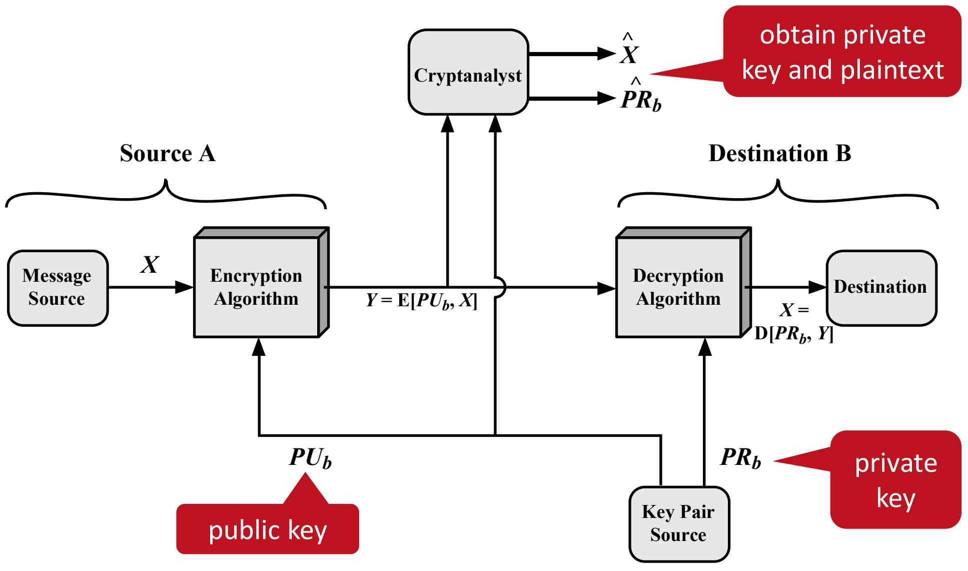
4

The first misconception is that public-key encryption is more secure from cryptanalysis than is symmetric encryption. In fact, the security of any encryption scheme depends on the length of the key and the computational work involved in breaking a cipher. There is nothing in principle about either symmetric or public-key encryption that makes one superior to another from the point of view of resisting cryptanalysis.

A second misconception is that public-key encryption is a general-purpose technique that has made symmetric encryption obsolete. On the contrary, because of the computational overhead of current public-key encryption schemes, there seems no foreseeable likelihood that symmetric encryption will be abandoned. As one of the inventors of public-key encryption has put it, “the restriction of public-key cryptography to key management and signature applications is almost universally accepted.”

Finally, there is a feeling that key distribution is trivial when using public-key encryption, compared to the rather cumbersome handshaking involved with key distribution centers for symmetric encryption. In fact, some form of protocol is needed, generally involving a central agent, and the procedures involved are not simpler nor any more efficient than those required for symmetric encryption.

Confidentiality using asymmetric encryption



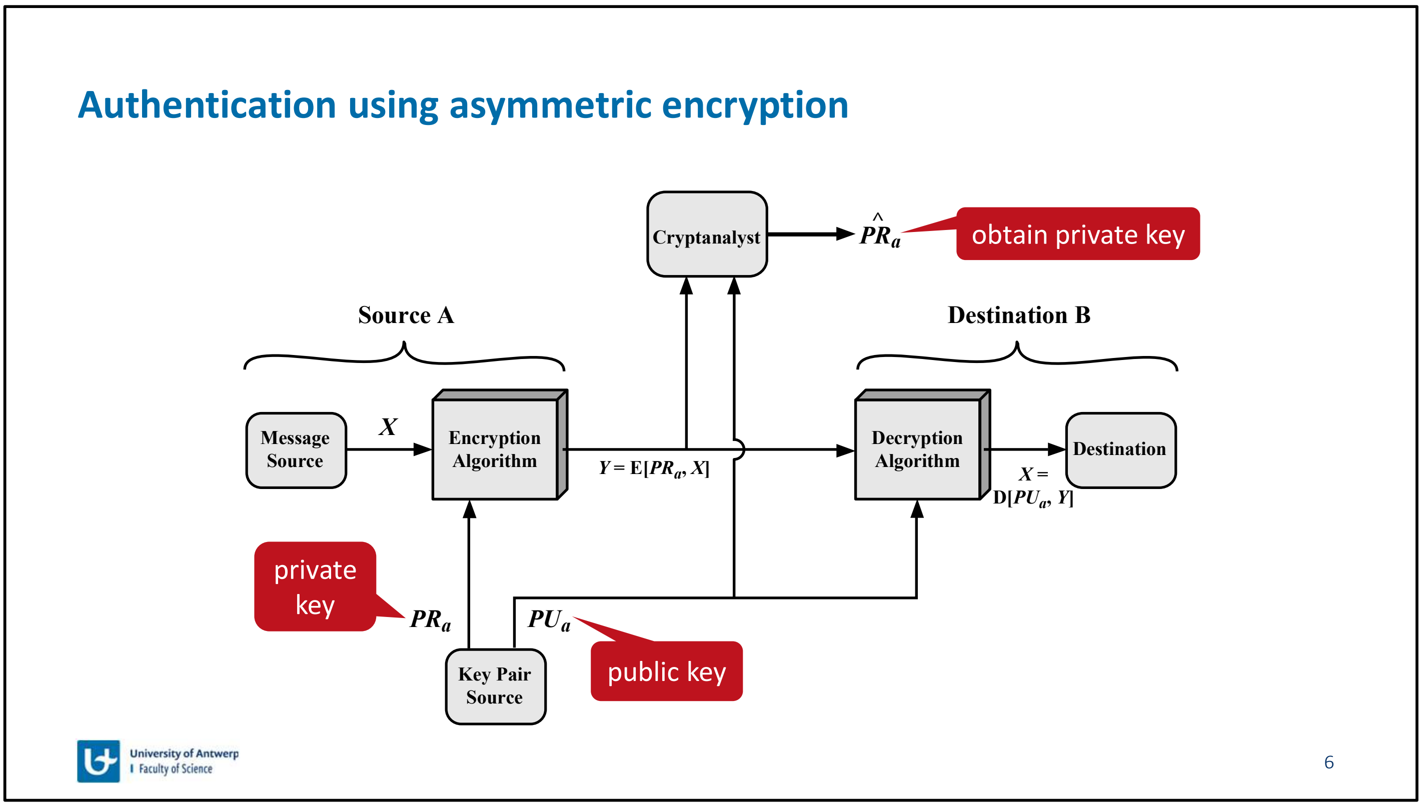
Let us take a closer look at the essential elements of a public-key encryption scheme. There is some source A that produces a message in plaintext, $X = [X_1, X_2, \ldots , X_M]$. The M elements of X are letters in some finite alphabet. The message is intended for destination B. B generates a related pair of keys: a public key, PU_b, and a private key, PR_b. PR_b is known only to B, whereas PU_b is publicly available and therefore accessible by A.
With the message X and the encryption key PU_b as input, A forms the ciphertext $Y = [Y_1, Y_2, \ldots , Y_N]$:

$$Y = E(PU_b, X)$$

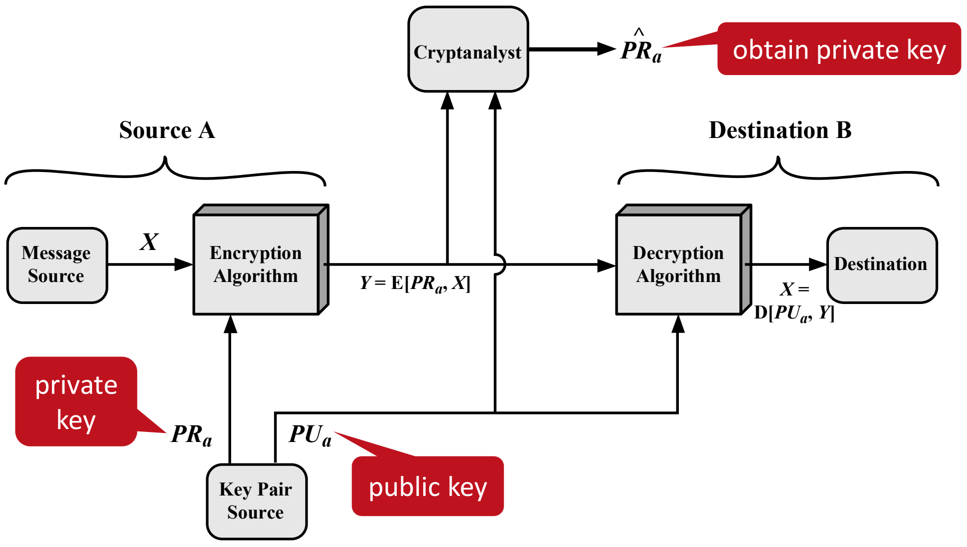
The intended receiver, in possession of the matching private key, is able to invert the transformation:

$$X = D(PR_b, Y)$$

An adversary, observing Y and having access to PU_b, but not having access to PR_b or X, must attempt to recover X and/or PR_b. It is assumed that the adversary does have knowledge of the encryption (E) and decryption (D) algorithms. If the adversary is interested only in this particular message, then the focus of effort is to recover X by generating a plaintext estimate $\hat{X}$. Often, however, the adversary is interested in being able to read future messages as well, in which case an attempt is made to recover PR_b by generating an estimate $\widehat{PR}_b$.



Authentication using asymmetric encryption

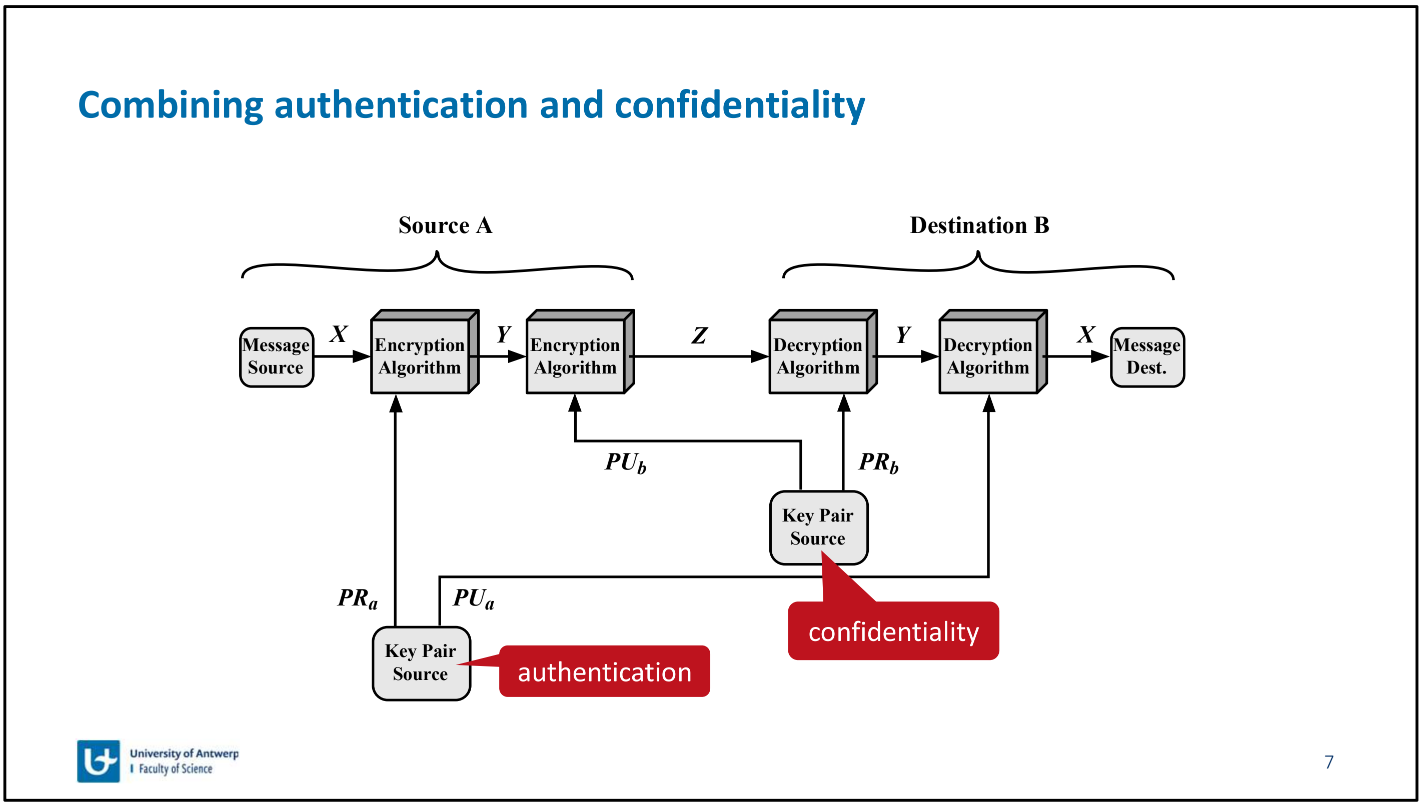


University of Antwerp | Faculty of Science

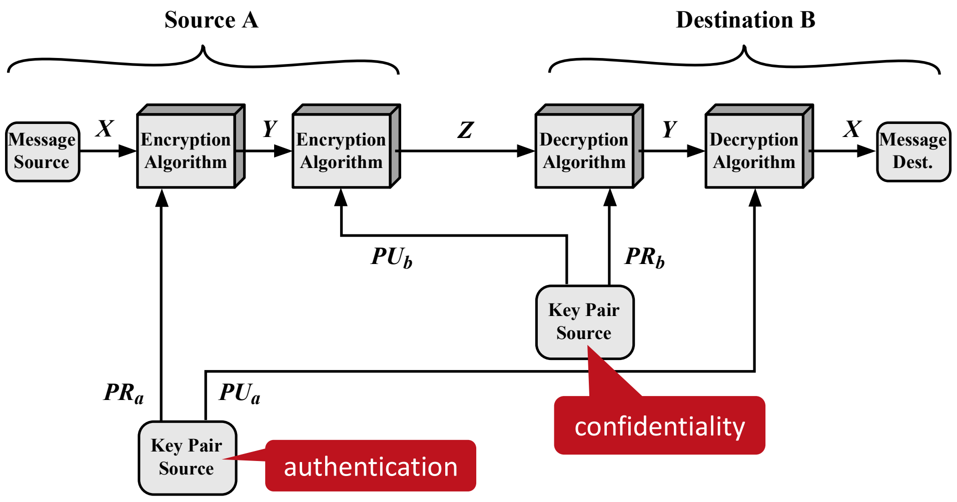
6

In this case, A prepares a message to B and encrypts it using A’s private key before transmitting it. B can decrypt the message using A’s public key. Because the message was encrypted using A’s private key, only A could have prepared the message. Therefore, the entire encrypted message serves as a **digital signature**. In addition, it is impossible to alter the message without access to A’s private key, so the message is authenticated both in terms of source and in terms of data integrity.

In this scheme, the entire message is encrypted, which, although validating both author and contents, requires a great deal of storage. Each document must be kept in plaintext to be used for practical purposes. A copy also must be stored in ciphertext so that the origin and contents can be verified in case of a dispute. A more efficient way of achieving the same results is to encrypt a small block of bits that is a function of the document. Such a block, called an authenticator, must have the property that it is infeasible to change the document without changing the authenticator. If the authenticator is encrypted with the sender’s private key, it serves as a signature that verifies origin, content, and sequencing.



Combining authentication and confidentiality



University of Antwerp | Faculty of Science

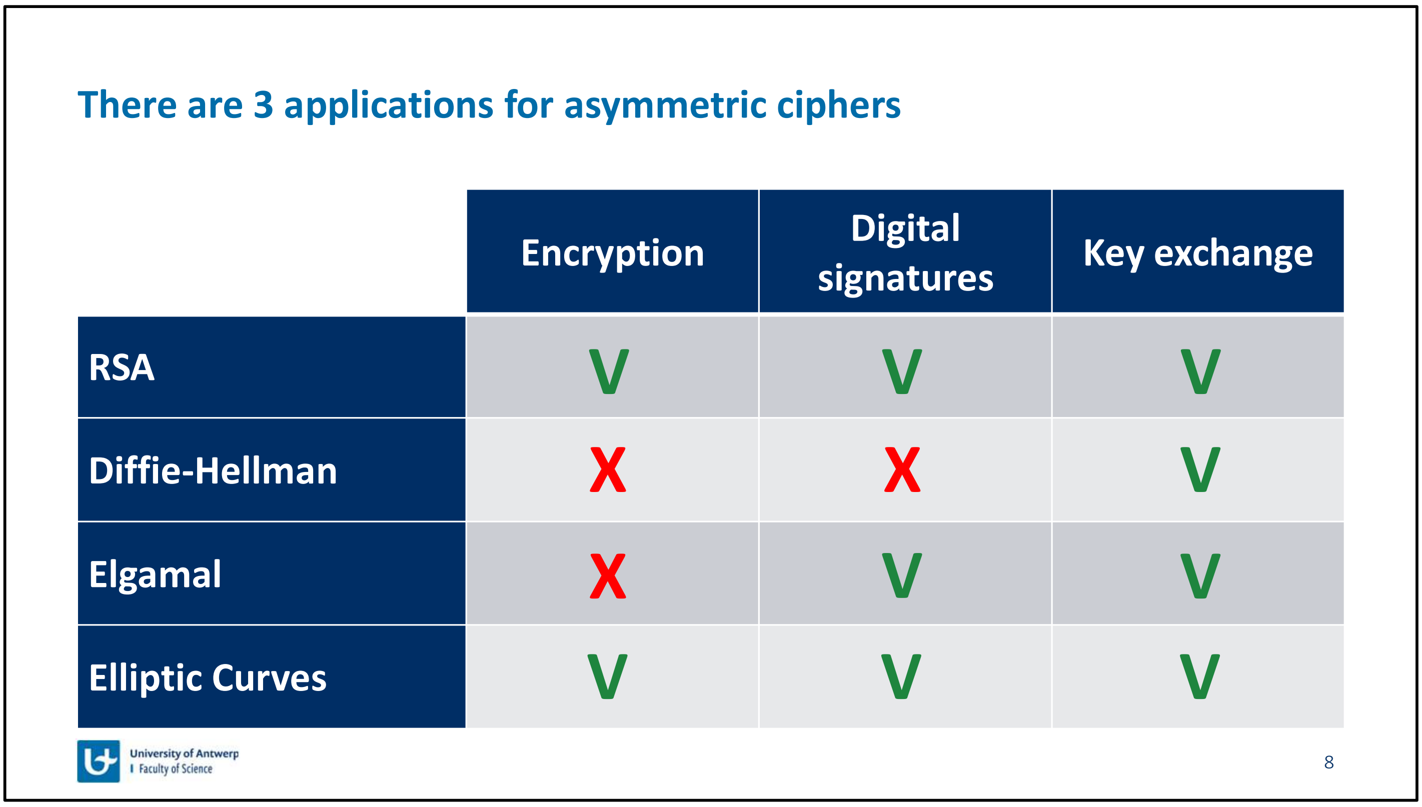
7

It is possible to provide both the authentication function and confidentiality by a double use of the public-key scheme:

$Z = \mathrm{E}(PU_b, \mathrm{E}(PR_a, X))$

$X = \mathrm{D}(PU_a, \mathrm{D}(PR_b, Z))$

In this case, we begin as before by encrypting a message, using the sender's private key. This provides the digital signature. Next, we encrypt again, using the receiver's public key. The final ciphertext can be decrypted only by the intended receiver, who alone has the matching private key. Thus, confidentiality is provided. The disadvantage of this approach is that the public-key algorithm, which is complex, must be exercised four times rather than two in each communication.



There are 3 applications for asymmetric ciphers

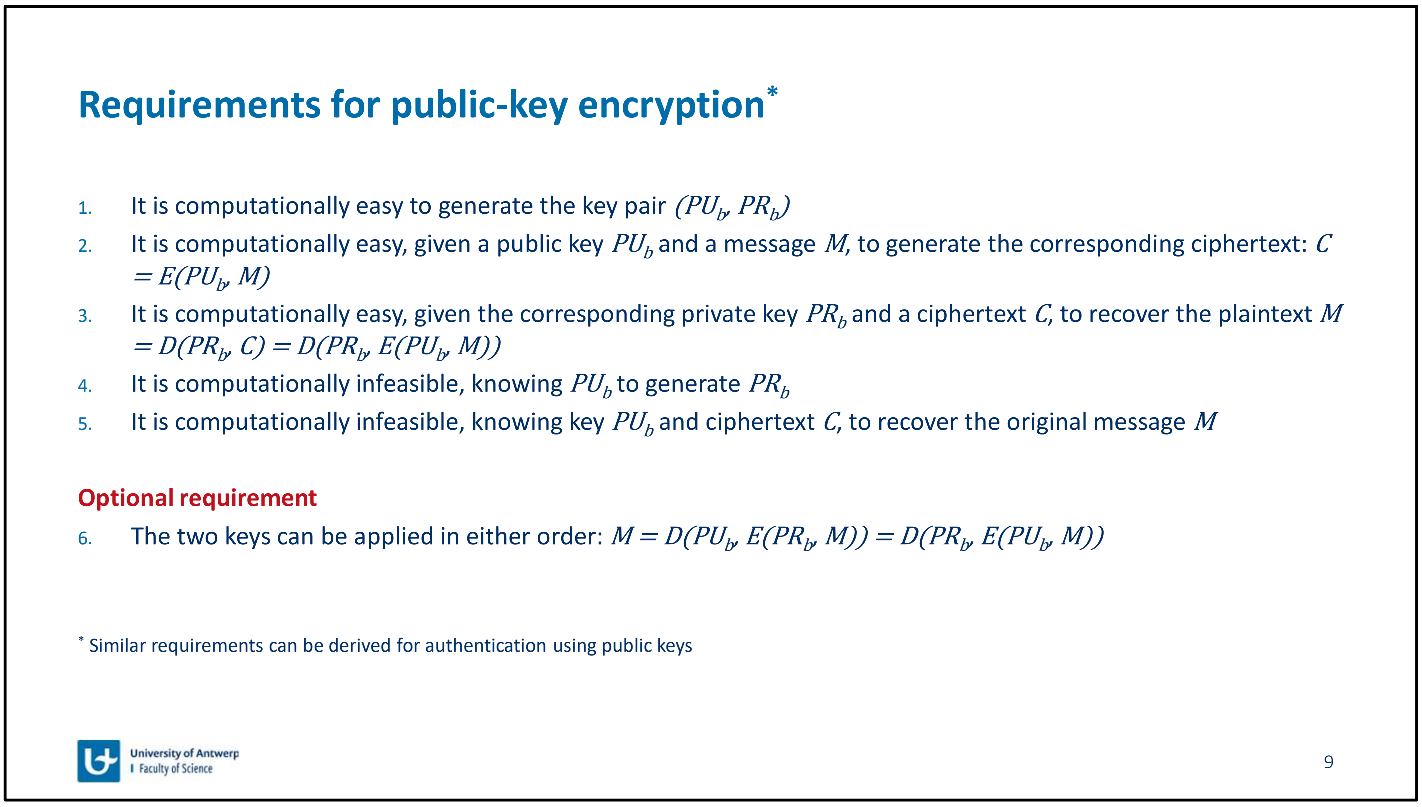
	Encryption	Digital signatures	Key exchange
RSA	V	V	V
Diffie-Hellman	X	X	V
Elgamal	X	V	V
Elliptic Curves	V	V	V

University of Antwerp
Faculty of Science

In broad terms, we can classify the use of **public-key cryptosystems** into three categories

- **Encryption/decryption:** The sender encrypts a message with the recipient's public key, and the recipient decrypts the message with the recipient's private key.
- **Digital signature:** The sender "signs" a message with its private key. Signing is achieved by a cryptographic algorithm applied to the message or to a small block of data that is a function of the message.
- **Key exchange:** Two sides cooperate to exchange a session key, which is a secret key for symmetric encryption generated for use for a particular transaction (or session) and valid for a short period of time. Several different approaches are possible, involving the private key(s) of one or both parties.

Some algorithms are suitable for all three applications, whereas others can be used only for one or two of these applications.



Requirements for public-key encryption*

1. It is computationally easy to generate the key pair (PU_b, PR_b)
2. It is computationally easy, given a public key PU_b and a message M, to generate the corresponding ciphertext: $C = E(PU_b, M)$
3. It is computationally easy, given the corresponding private key PR_b and a ciphertext C, to recover the plaintext $M = D(PR_b, C) = D(PR_b, E(PU_b, M))$
4. It is computationally infeasible, knowing PU_b to generate PR_b
5. It is computationally infeasible, knowing key PU_b and ciphertext C, to recover the original message M

Optional requirement

6. The two keys can be applied in either order: $M = D(PU_b, E(PR_b, M)) = D(PR_b, E(PU_b, M))$

* Similar requirements can be derived for authentication using public keys

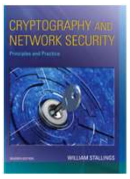
University of Antwerp
Faculty of Science

9

These are formidable requirements, as evidenced by the fact that only a few algorithms (RSA, elliptic curve cryptography, Diffie–Hellman, DSS) have received widespread acceptance in the several decades since the concept of public-key cryptography was proposed.

Summary on asymmetric encryption basics

- Uses a public and a private key
- Can be used for **confidentiality** and **authentication**
- Too **computationally expensive** for encrypting large data blocks
- Three **applications**: encryption, digital signatures and key exchange
- 4 **algorithms** exist that satisfy the requirements of asymmetric encryption:
  - RSA and Elliptic Curves can be used for all 3 applications
  - Diffie-Hellman can be used for key exchange
  - Elgamal encryption can be used for digital signatures and key exchange



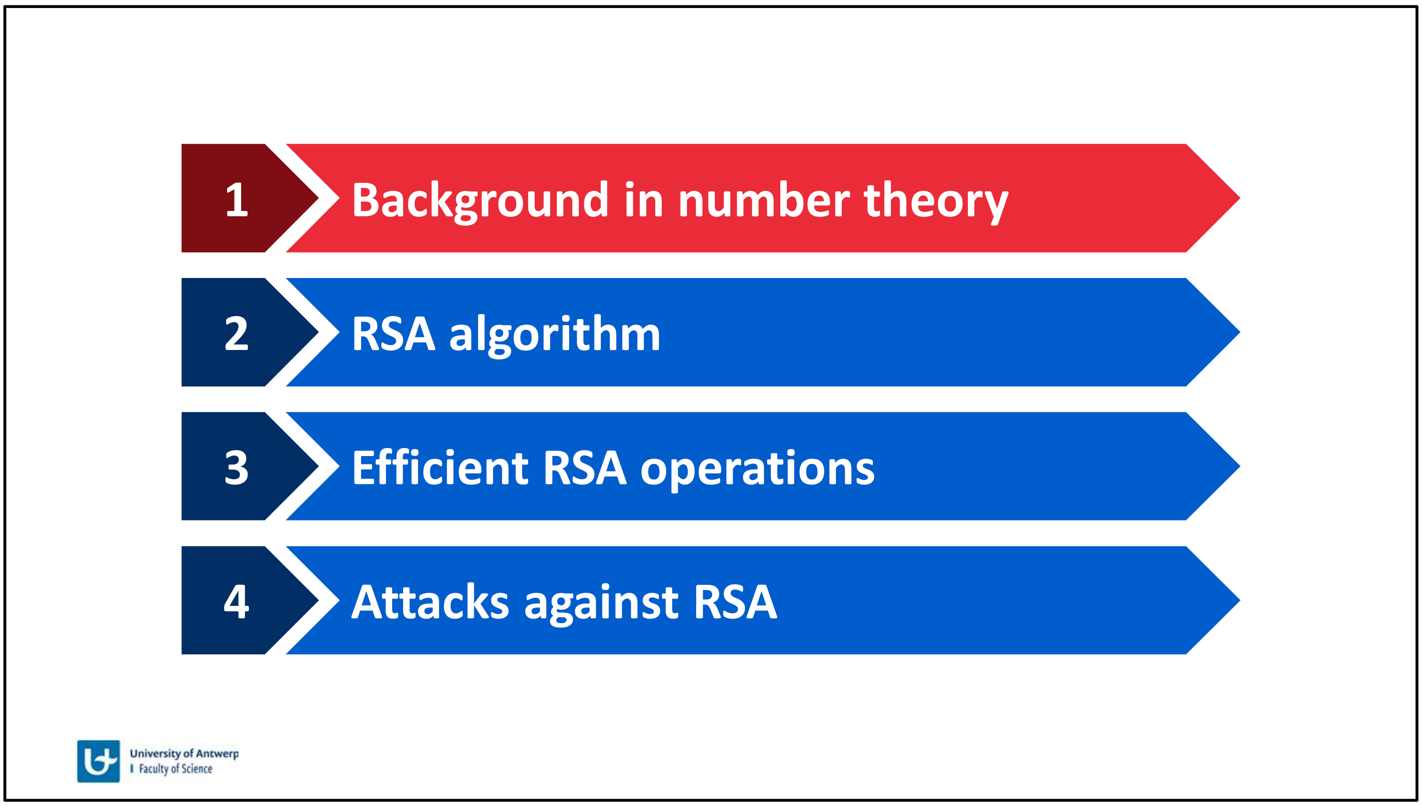
Link with the book
- Chapter 9 (Section 9.1)

Link with the videos
- Week 6 (Encryption from trapdoor permutations)

University of Antwerp
Faculty of Science

Rivest-Shamir-Adleman (RSA)

University of Antwerp
Faculty of Science



1. Background in number theory
2. RSA algorithm
3. Efficient RSA operations
4. Attacks against RSA

University of Antwerp | Faculty of Science

Divisibility

- We say that a nonzero b **divides** a (also written as b | a, or b is a **divisor** of a)
  - if $a = m \times b$ for some m
  - where a, b, and m are integers

- Properties of divisibility
  - If a | 1, then a = ±1
  - If a | b and b | a, then a = ±b
  - Any b ≠ 0 divides 0
  - If a | b and b | c, then a | c
  - If b | g and b | h, then b | (m g + n h) for arbitrary integers m and n

Greatest common divisor (GCD)

- The **greatest common divisor** of a and b is the largest integer that divides both a and b
  - Written as gcd(a, b)
- The greatest common divisor should be positive
  - gcd(a,b) = gcd(a,-b) = gcd(-a,b) = gcd(-a,-b)
  - Or, gcd(a,b) = gcd(|a|, |b|)
- Because all nonzero integers divide 0, gcd(a, 0) = |a|
- Two integers a and b are **relatively prime** if gcd(a, b) = 1

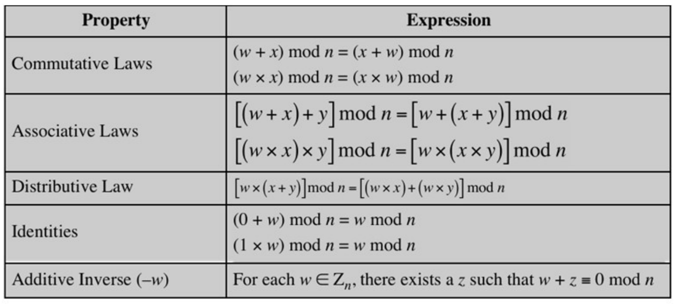
Modulus definition

- If a is an integer and n is a positive integer, we define **a mod n** to be the remainder when a is divided by n; the integer n is called the **modulus**
- Thus, for any integer a:
  - a = qn + r 0 ≤ r < n; q = [a/ n]
  - a = [a/ n] * n + (a mod n)

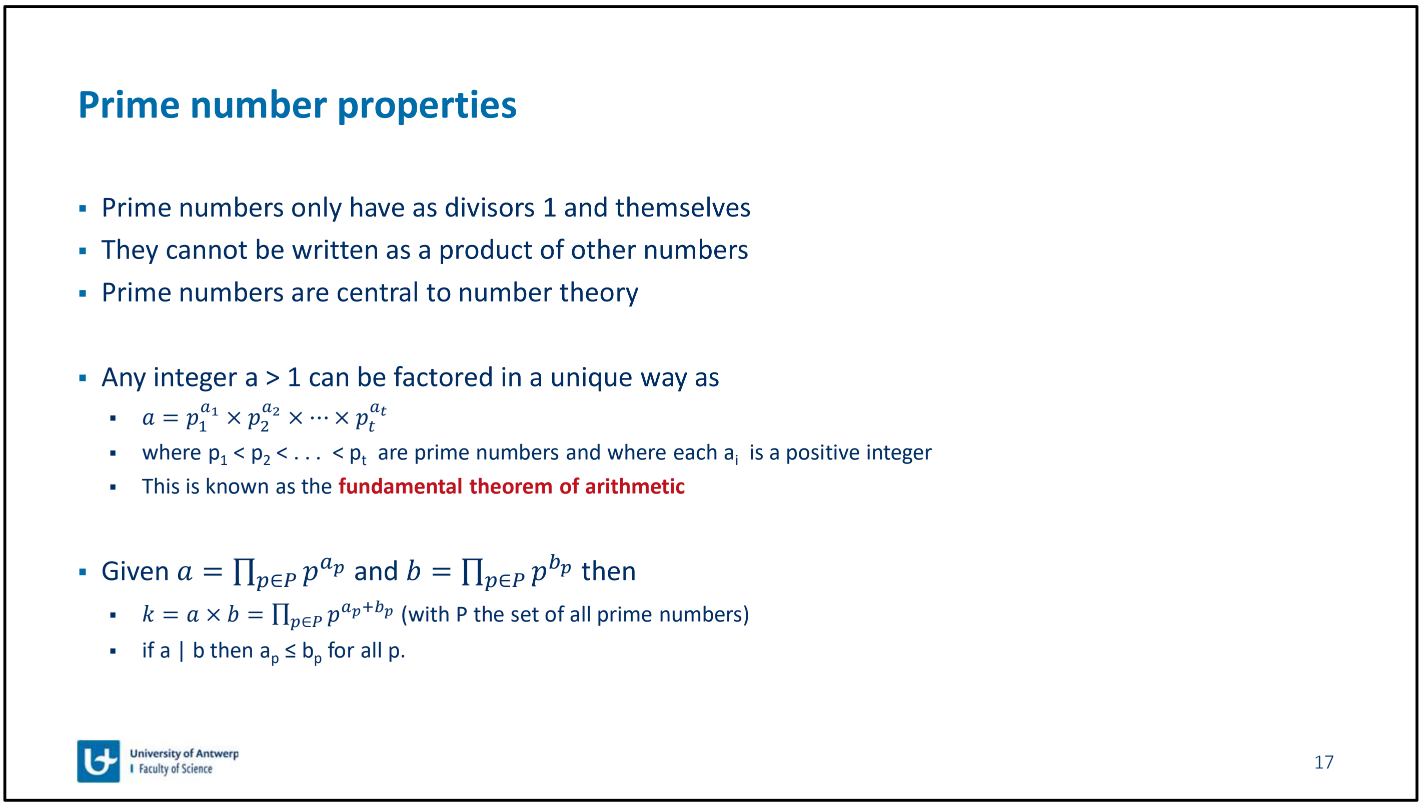
- Two integers a and b are said to be **congruent modulo n** if (a mod n) = (b mod n)
  - This is written as a ≡ b (mod n)
  - Note that if a ≡ 0 (mod n), then n | a
- **Congruences** have the following properties:
  - a ≡ b (mod n) if n = (a – b)
  - a ≡ b (mod n) implies b ≡ a (mod n)
  - a ≡ b (mod n) and b ≡ c (mod n) imply a ≡ c (mod n)

Additional properties of modular arithmetic

- [(a mod n) + (b mod n)] mod n = (a + b) mod n
- [(a mod n) - (b mod n)] mod n = (a - b) mod n
- [(a mod n) * (b mod n)] mod n = (a * b) mod n
- Inverse
  - a is the additive inverse of b if $(a + b) \bmod n = 0$
  - a is the multiplicative inverse of b if $a \times b \equiv 1 \pmod{n}$



Property	Expression
Commutative Laws	$(w + x) \bmod n = (x + w) \bmod n$ $(w \times x) \bmod n = (x \times w) \bmod n$
Associative Laws	$[(w + x) + y] \bmod n = [w + (x + y)] \bmod n$ $[(w \times x) \times y] \bmod n = [w \times (x \times y)] \bmod n$
Distributive Law	$[w \times (x + y)] \bmod n = [(w \times x) + (w \times y)] \bmod n$
Identities	$(0 + w) \bmod n = w \bmod n$ $(1 \times w) \bmod n = w \bmod n$
Additive Inverse $(-w)$	For each $w \in Z_n$, there exists a z such that $w + z \equiv 0 \bmod n$



Prime number properties

- Prime numbers only have as divisors 1 and themselves
- They cannot be written as a product of other numbers
- Prime numbers are central to number theory

- Any integer a > 1 can be factored in a unique way as
  - $a = p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_t^{a_t}$
  - where $p_1 < p_2 < . . . < p_t$ are prime numbers and where each a_i is a positive integer
  - This is known as the **fundamental theorem of arithmetic**

- Given $a = \prod_{p \in P} p^{a_p}$ and $b = \prod_{p \in P} p^{b_p}$ then
  - $k = a \times b = \prod_{p \in P} p^{a_p + b_p}$ (with P the set of all prime numbers)
  - if a | b then $a_p \leq b_p$ for all p.

University of Antwerp
Faculty of Science

17

Fermat's theorem

- Fermat's and Euler's theorems are very important in public key cryptography
- If p is prime and a is a positive integer not divisible by p, then
$$a^{p-1} \equiv 1 \pmod{p}$$
- (alternative form) If p is a prime and a is a positive integer then
$$a^{p} \equiv a \pmod{p}$$

Euler’s theorem

- **Euler’s Totient Function $\boldsymbol{\phi(n)}$** is defined as the number of positive integers less than n that are relatively prime to n

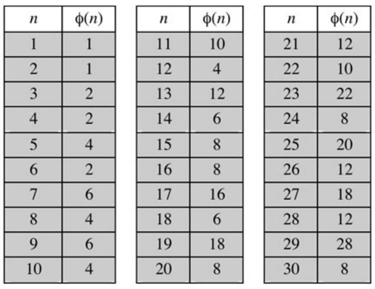
- For every a and n that are relatively prime

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

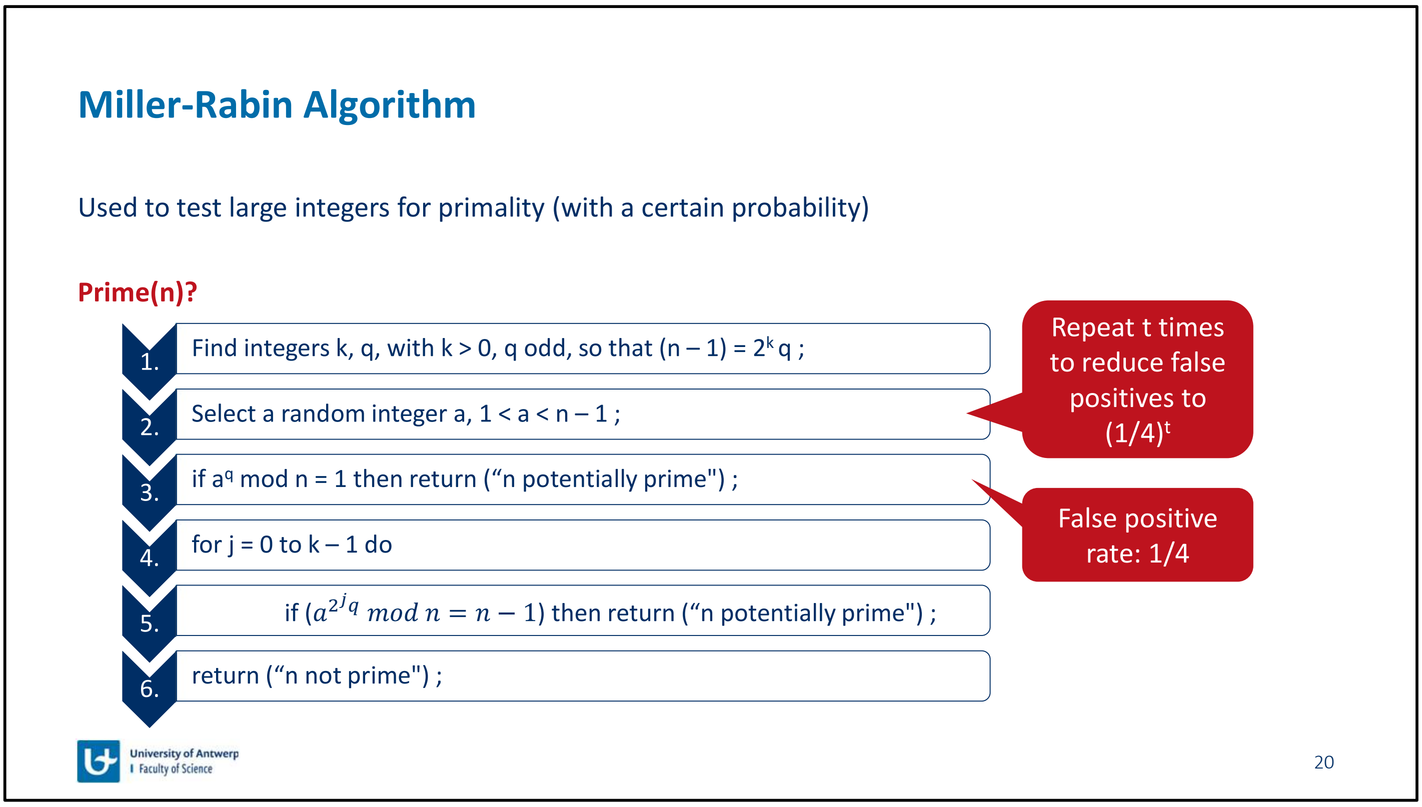
- Alternative form

$$a^{\phi(n)+1} \equiv a \pmod{n}$$

- Examples of Euler’s Totient Function:



n	$\phi(n)$	n	$\phi(n)$	n	$\phi(n)$
1	1	11	10	21	12
2	1	12	4	22	10
3	2	13	12	23	22
4	2	14	6	24	8
5	4	15	8	25	20
6	2	16	8	26	12
7	6	17	16	27	18
8	4	18	6	28	12
9	6	19	18	29	28
10	4	20	8	30	8



Miller-Rabin Algorithm

Used to test large integers for primality (with a certain probability)

Prime(n)?

```
1. Find integers k, q, with k > 0, q odd, so that $(n – 1) = 2^k q$ ;
2. Select a random integer a, $1 < a < n – 1$ ;
3. if $a^q \bmod n = 1$ then return (“n potentially prime") ;
4. for j = 0 to k – 1 do
5.         if $(a^{2^j q} \bmod n = n - 1)$ then return (“n potentially prime") ;
6. return (“n not prime") ;
```

Repeat t times to reduce false positives to $(1/4)^t$

False positive rate: 1/4

University of Antwerp | Faculty of Science

20

Chinese remainder theorem (CRT)

Let

$$M = \prod_{i=1}^{k} m_i$$

where the m_i are pairwise relatively prime

We can represent any integer $A \in \mathbb{Z}_M$ by a k-tuple whose elements are in $\mathbb{Z}_{m_i}$ using the correspondence

$$A \leftrightarrow (a_1, a_2, \ldots, a_k)$$

where $a_i \in \mathbb{Z}_{m_i}$ and $a_i = A \bmod m_i$

$\mathbb{Z}_x$ is the set of integers modulo x

Calculating A based on its unique bijection

For every A there is a unique mapping to a k-tuple, called a bijection

We can calculate A from the k-tuple (a_1, …, a_k) as follows:

$$A \equiv \left(\sum_{i=1}^{k} a_i c_i\right)(mod\ M)$$

with

$$c_i = M_i \times \left(M_i^{-1} mod\ m_i\right)$$

and

$$M_i = {}^{M}\!/_{m_i}$$

Operations on A can be performed on bijection

Operations performed on A can be equivalently performed individually on each element of the corresponding k-tuple

Given:

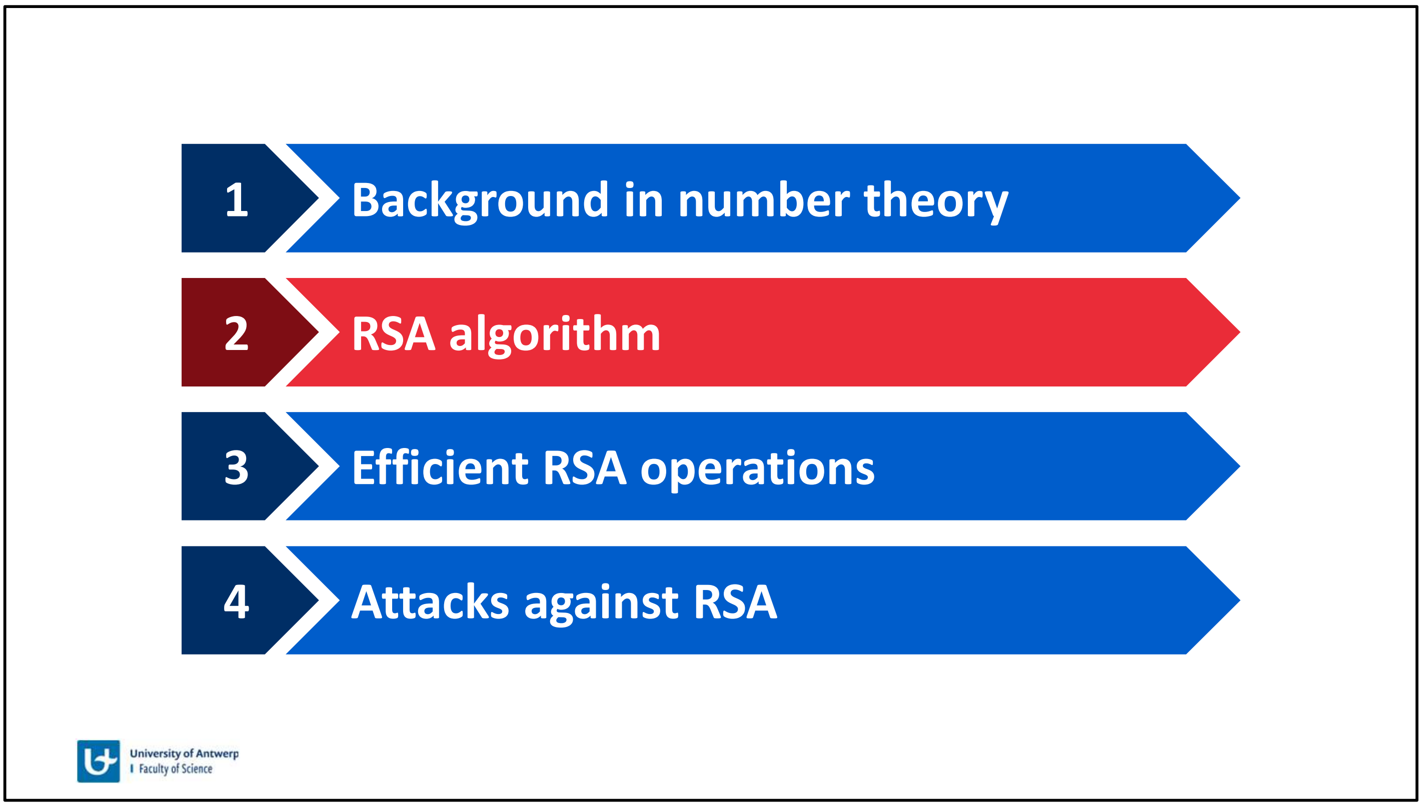
$$A \leftrightarrow (a_1, a_2, \dots, a_k) \text{ and } B \leftrightarrow (b_1, b_2, \dots, b_k)$$

then

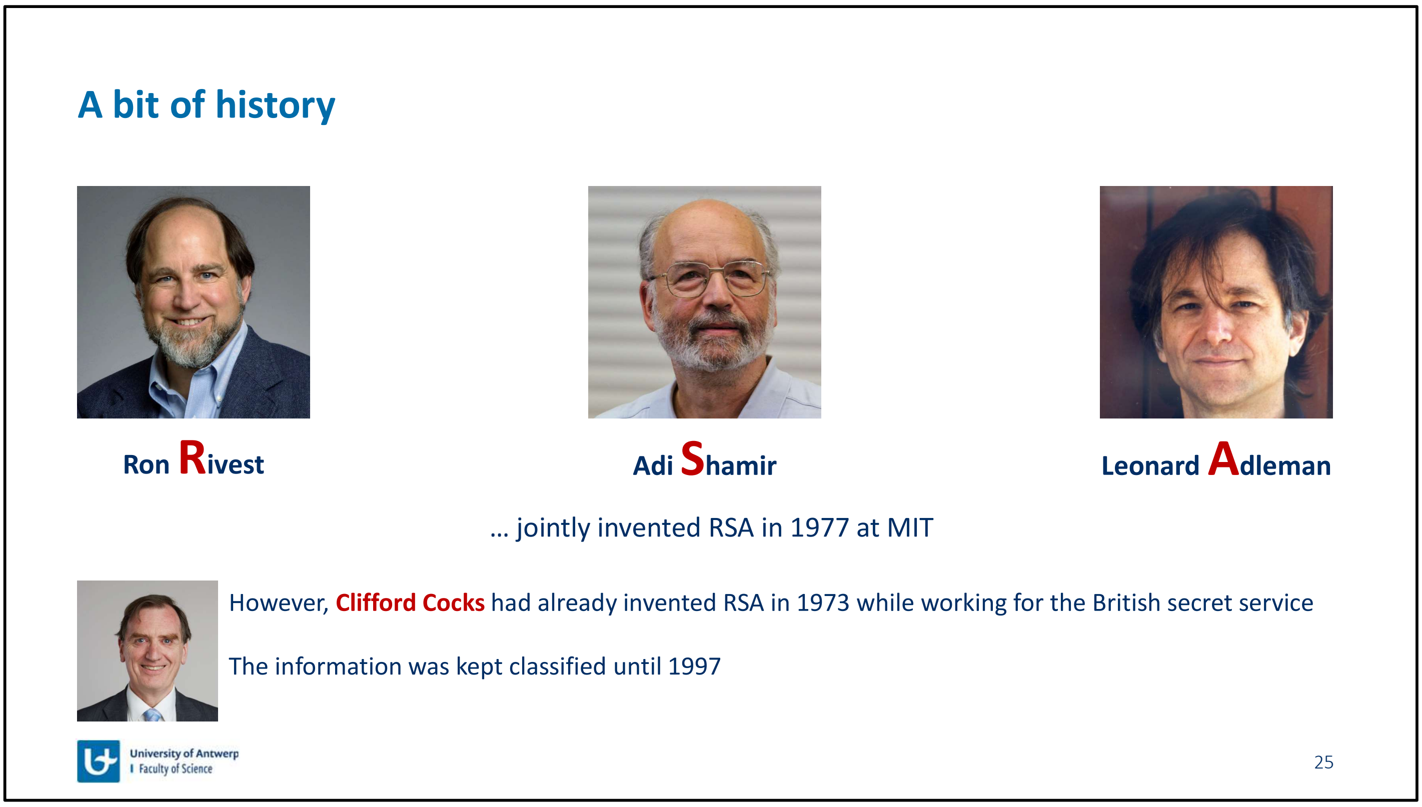
$$(A + B) \; mod \; M \; \leftrightarrow \left((a_1 + b_1) \; mod \; m_1, \cdots, (a_k + b_k) \; mod \; m_k\right)$$
$$(A - B) \; mod \; M \; \leftrightarrow \left((a_1 - b_1) \; mod \; m_1, \cdots, (a_k - b_k) \; mod \; m_k\right)$$
$$(A \times B) \; mod \; M \; \leftrightarrow \left((a_1 \times b_1) \; mod \; m_1, \cdots, (a_k \times b_k) \; mod \; m_k\right)$$

As such, the CRT provides a way to **manipulate large numbers mod M** in terms of tuples of smaller numbers.

University of Antwerp
Faculty of Science



University of Antwerp
Faculty of Science



A bit of history

Ron **R**ivest

Adi **S**hamir

Leonard **A**dleman

… jointly invented RSA in 1977 at MIT

However, **Clifford Cocks** had already invented RSA in 1973 while working for the British secret service

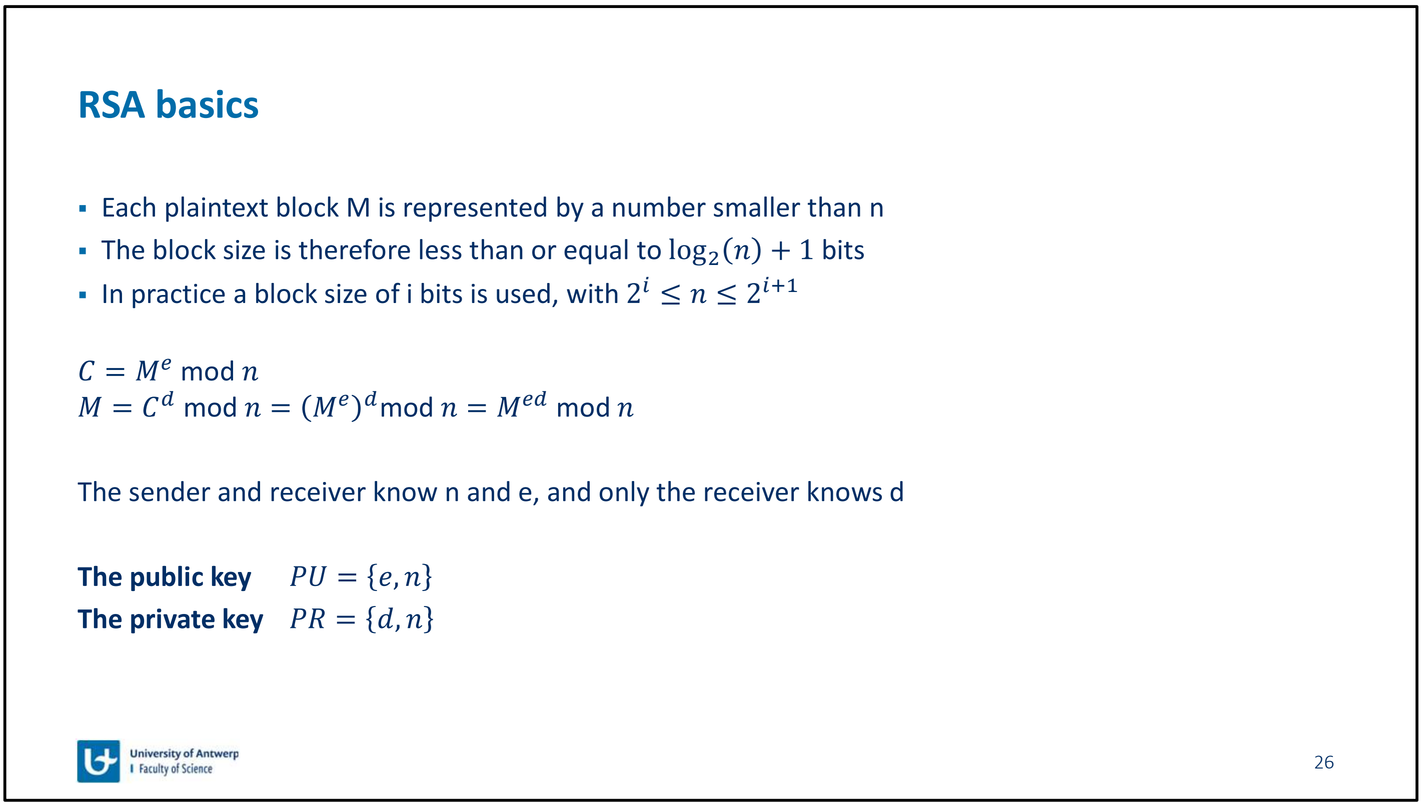
The information was kept classified until 1997

University of Antwerp
Faculty of Science

25

The pioneering paper by Diffie and Hellman introduced a new approach to cryptography and, in effect, challenged cryptologists to come up with a cryptographic algorithm that met the requirements for public-key systems. A number of algorithms have been proposed for public-key cryptography. Some of these, though initially promising, turned out to be breakable.

One of the first successful responses to the challenge was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978. The Rivest-Shamir-Adleman (RSA) scheme has since that time reigned supreme as the most widely accepted and implemented general-purpose approach to public-key encryption.le



RSA basics

- Each plaintext block M is represented by a number smaller than n
- The block size is therefore less than or equal to $\log_2(n) + 1$ bits
- In practice a block size of i bits is used, with $2^i \leq n \leq 2^{i+1}$

$C = M^e \text{ mod } n$
$M = C^d \text{ mod } n = (M^e)^d \text{mod } n = M^{ed} \text{ mod } n$

The sender and receiver know n and e, and only the receiver knows d

The public key $PU = \{e, n\}$
The private key $PR = \{d, n\}$

University of Antwerp | Faculty of Science

26

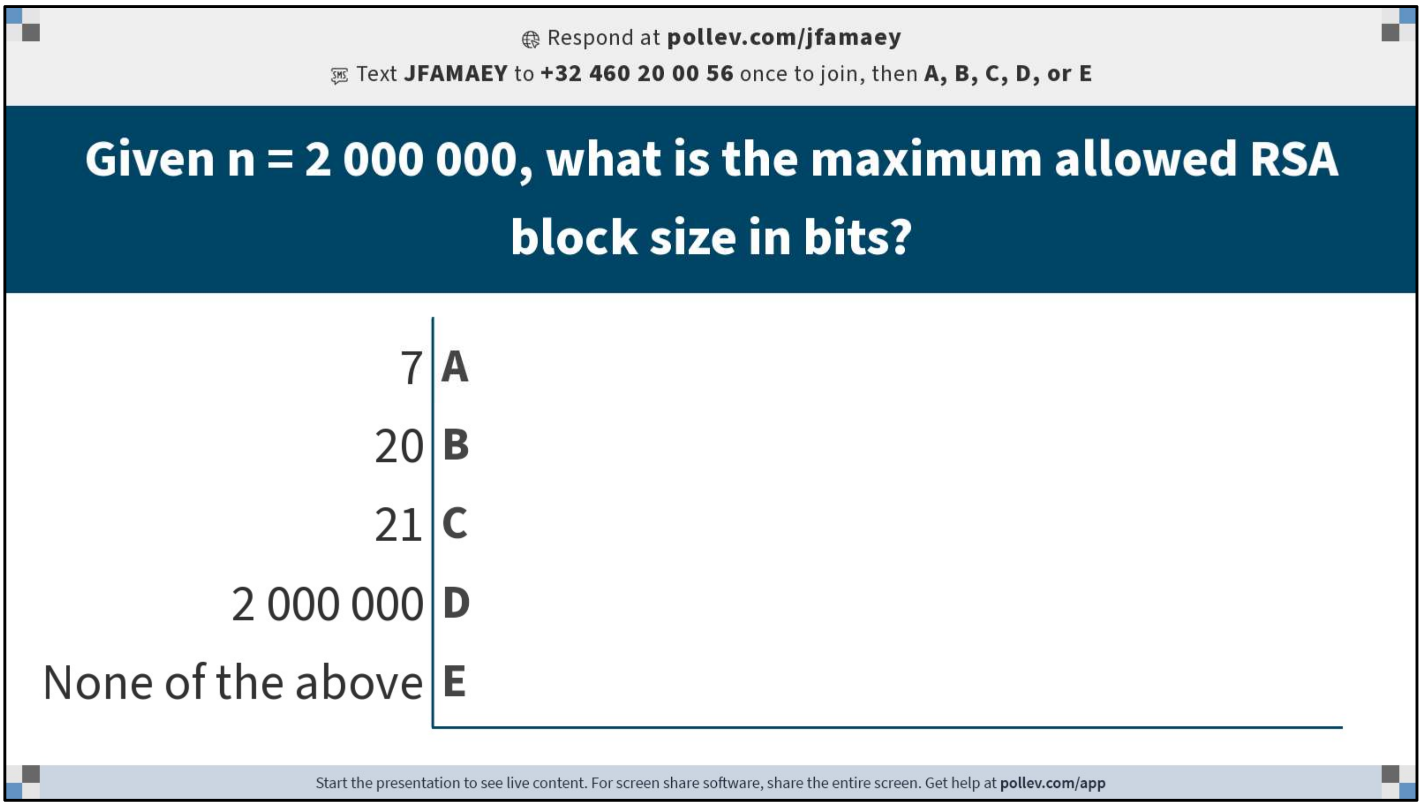
RSA makes use of an expression with exponentials. Plaintext is encrypted in blocks, with each block having a binary value less than some number *n*. That is, the block size must be less than or equal to $\log_2(n) + 1$; in practice, the block size is *i* bits, where $2^i <= n <= 2^{i+1}$. Encryption and decryption are of the following form, for some plaintext block *M* and ciphertext block *C*.

$C = M^e \text{ mod } n$
$M = C^d \text{ mod } n = (M^e)^d \text{ mod } n = M^{ed} \text{ mod } n$

Both sender and receiver must know the value of *n*. The sender knows the value of *e*, and only the receiver knows the value of *d*. Thus, this is a public-key encryption algorithm with a public key of *PU* = {*e*, *n*} and a private key of *PR* = {*d*, *n*}.

For this algorithm to be satisfactory for public-key encryption, the following requirements must be met:

1. It is possible to find values of *e*, *d*, and *n* such that $M^{ed} \text{ mod } n = M$ for all $M < n$.
2. It is relatively easy to calculate $M^e \text{ mod } n$ and $C^d \text{ mod } n$ for all values of $M < n$.
3. It is infeasible to determine *d* given *e* and *n*.


Respond at **pollev.com/jfamaey**

Text **JFAMAEY** to **+32 460 20 00 56** once to join, then **A, B, C, D, or E**

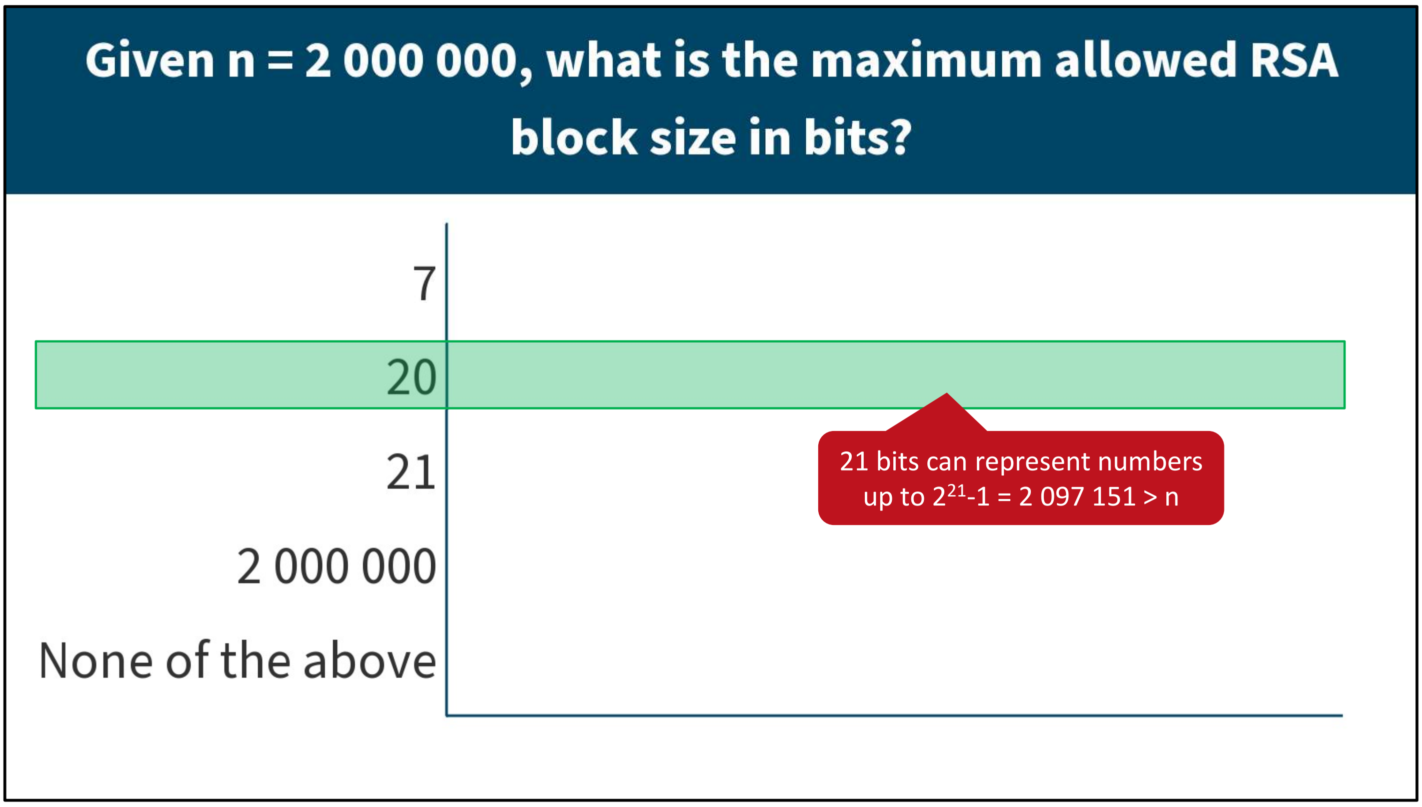
Given n = 2 000 000, what is the maximum allowed RSA block size in bits?

7	**A**
20	**B**
21	**C**
2 000 000	**D**
None of the above	**E**

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

Given n = 2 000 000, what is the maximum allowed RSA block size in bits?
https://www.polleverywhere.com/multiple_choice_polls/TeQgwpkoblgTWrw?state=opened&flow=Default&onscreen=persist

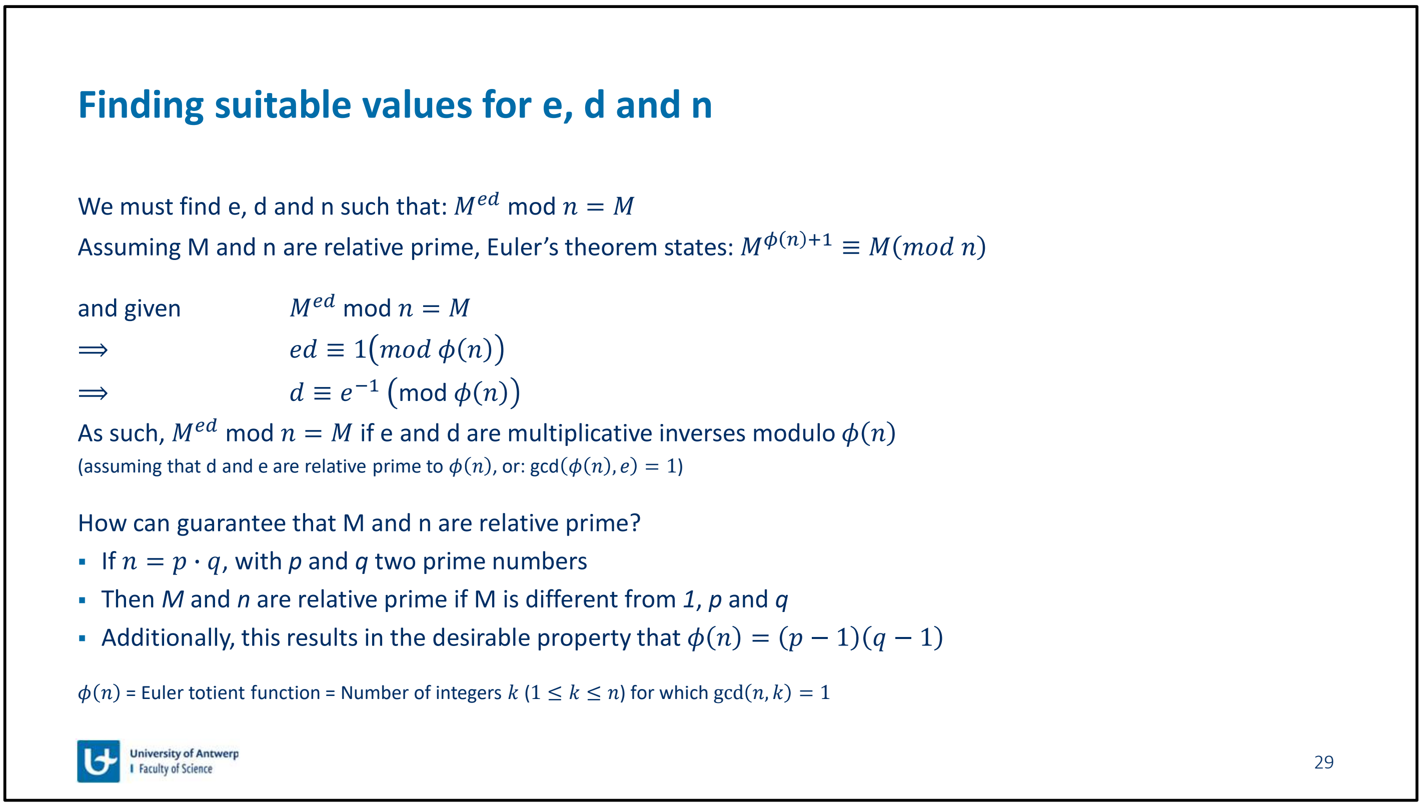


Given n = 2 000 000, what is the maximum allowed RSA block size in bits?

- 7
- 20 [✓]
- 21
- 2 000 000
- None of the above

21 bits can represent numbers up to 2^{21}-1 = 2 097 151 > n

Poll Title: Given n = 2 000 000, what is the maximum allowed RSA block size in bits?



Finding suitable values for e, d and n

We must find e, d and n such that: M^{ed} mod $n = M$

Assuming M and n are relative prime, Euler's theorem states: $M^{\phi(n)+1} \equiv M(mod\ n)$

and given M^{ed} mod $n = M$

$\Longrightarrow$ $ed \equiv 1\big(mod\ \phi(n)\big)$

$\Longrightarrow$ $d \equiv e^{-1}\ \big(\text{mod}\ \phi(n)\big)$

As such, M^{ed} mod $n = M$ if e and d are multiplicative inverses modulo $\phi(n)$

(assuming that d and e are relative prime to $\phi(n)$, or: gcd$(\phi(n), e) = 1$)

How can guarantee that M and n are relative prime?

- If $n = p \cdot q$, with *p* and *q* two prime numbers
- Then *M* and *n* are relative prime if M is different from *1*, *p* and *q*
- Additionally, this results in the desirable property that $\phi(n) = (p-1)(q-1)$

$\phi(n)$ = Euler totient function = Number of integers k $(1 \leq k \leq n)$ for which gcd$(n, k) = 1$

University of Antwerp I Faculty of Science

29

For now, we focus on the first requirement and consider the other questions later. We need to find a relationship of the form:

M^{ed} mod $n = M$

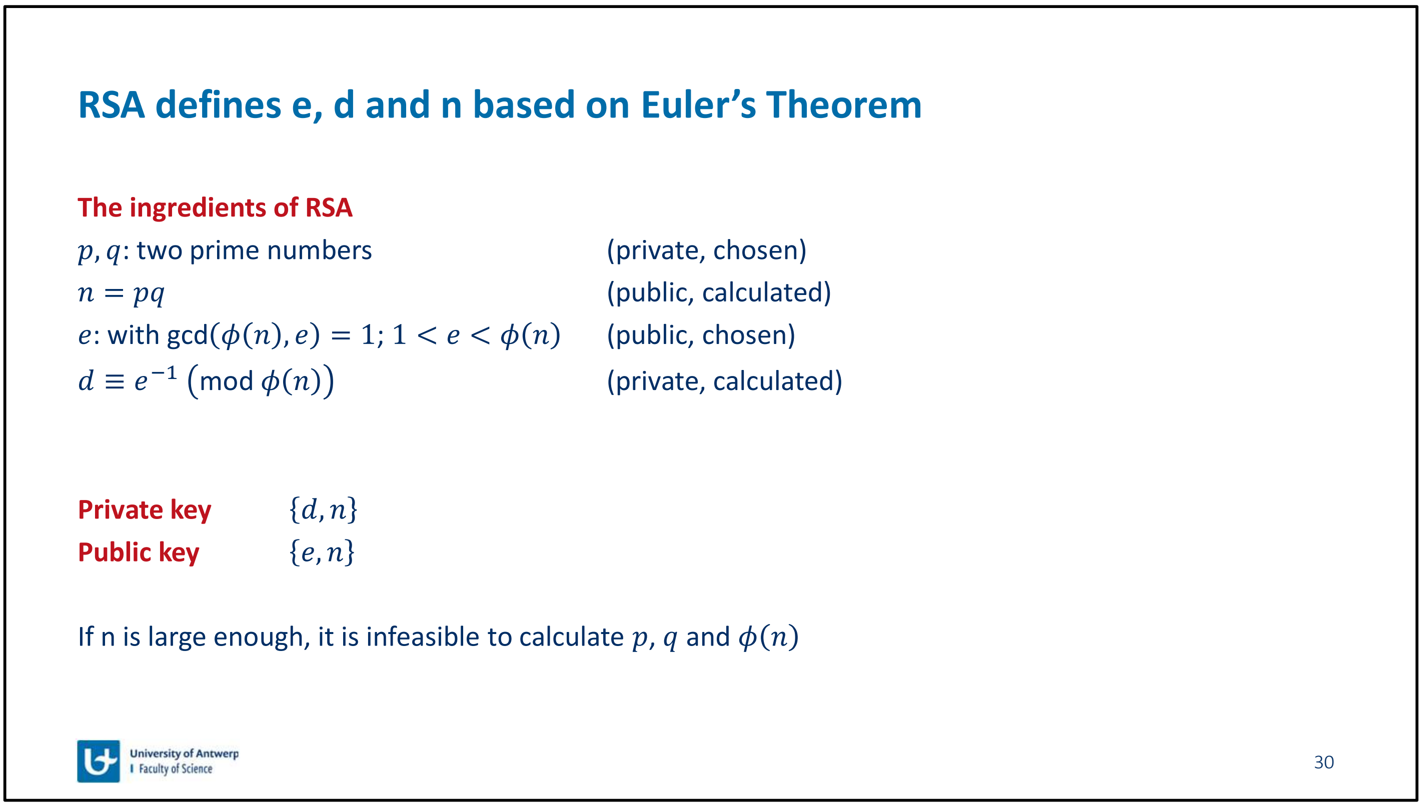
The preceding relationship holds if *e* and *d* are multiplicative inverses modulo $\phi(n)$, where $\phi(n)$ is the Euler totient function (cf., Euler's theorem). It can be shown that for *p, q* prime, $\phi(pq) = (p - 1)(q - 1)$. The relationship between *e* and *d* can be expressed as:

ed mod $\phi(n)$ = 1

This is equivalent to saying

$$ed \equiv 1 \left(\text{mod}\ \phi(n)\right)$$
$$d \equiv e^{-1} \left(\text{mod}\ \phi(n)\right)$$

That is, *e* and *d* are multiplicative inverses mod $\phi(n)$. Note that, according to the rules of modular arithmetic, this is true only if *d* (and therefore *e*) is relatively prime to $\phi(n)$. Equivalently, gcd($\phi(n)$, *d*) = 1.



RSA defines e, d and n based on Euler's Theorem

The ingredients of RSA

p, q: two prime numbers	(private, chosen)
$n = pq$	(public, calculated)
e: with $\gcd(\phi(n), e) = 1; 1 < e < \phi(n)$	(public, chosen)
$d \equiv e^{-1} \left(\text{mod } \phi(n)\right)$	(private, calculated)

Private key	$\{d, n\}$
Public key	$\{e, n\}$

If n is large enough, it is infeasible to calculate p, q and $\phi(n)$

University of Antwerp | Faculty of Science

30

The private key consists of {d, n} and the public key consists of {e, n}. Suppose that user A has published its public key and that user B wishes to send the message M to A. Then B calculates $C = M^e \bmod n$ and transmits C. On receipt of this ciphertext, user A decrypts by calculating $M = C^d \bmod n$.

Exercise: Calculating RSA private key

Exercise: Given two prime numbers p = 17 and q = 11, and e = 7, what is the private {d, n} RSA key?

$n = pq$

$d \equiv e^{-1} \pmod{\phi(n)}$

Initialization

$r_{-1} = \phi(n)$ $\quad$ $r_0 = e$

$w_{-1} = 0$ $\quad$ $w_0 = 1$

Recursive algorithm

$r_i = r_{i-2} \bmod r_{i-1}$

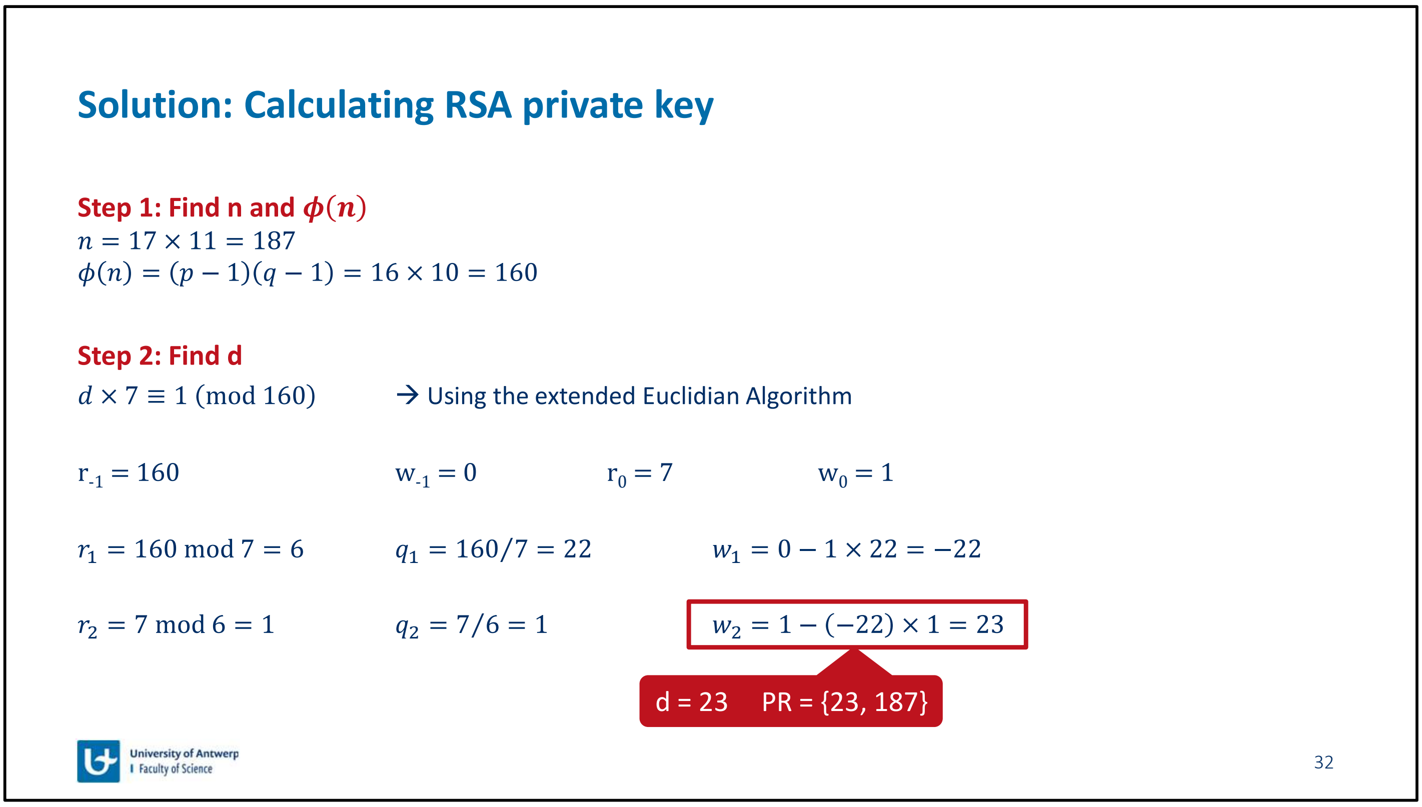
$q_i = r_{i-2} / r_{i-1}$

$w_i = w_{i-2} - q_i\, w_{i-1}$

Stop condition

$if\ r_i = 1$ $\quad$ $then\ w_i = e^{-1}$

University of Antwerp | Faculty of Science



Solution: Calculating RSA private key

Step 1: Find n and $\phi(n)$

$n = 17 \times 11 = 187$

$\phi(n) = (p-1)(q-1) = 16 \times 10 = 160$

Step 2: Find d

$d \times 7 \equiv 1 \pmod{160}$ → Using the extended Euclidian Algorithm

$r_{-1} = 160$ $w_{-1} = 0$ $r_0 = 7$ $w_0 = 1$

$r_1 = 160 \bmod 7 = 6$ $q_1 = 160/7 = 22$ $w_1 = 0 - 1 \times 22 = -22$

$r_2 = 7 \bmod 6 = 1$ $q_2 = 7/6 = 1$ $\boxed{w_2 = 1 - (-22) \times 1 = 23}$

d = 23 PR = {23, 187}

University of Antwerp
Faculty of Science

32

Exercise: Encryption with RSA

Exercise: Given p = 17, q = 11, e = 7, and d = 23, perform RSA encryption for M = 88.

$C = M^e \bmod n$

$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$

Solution: Encryption with RSA

$C = M^e \bmod n$ with M = 88, e = 7, and n = 187

Solution

$88^7\ mod\ 187$

$= \left[\left(88^2 \bmod 187\right) \times \left(88^2 \bmod 187\right) \times \left(88^2 \bmod 187\right) \times (88 \bmod 187)\right] \bmod 187$
$= [(77 \times 77) \bmod 187\ \times (77 \times 88) \bmod 187] \bmod 187$
$= [132 \times 44] \bmod 187$
$= 11 = C$

University of Antwerp
Faculty of Science

Exercise: Decryption with RSA

Exercise: Given p = 17, q = 11, e = 7, and d = 23, perform RSA decryption for C = 11.

$C = M^e \bmod n$

$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$

Solution: Decryption with RSA

$M = C^d \bmod n$ with C = 11, d = 23, and n = 187

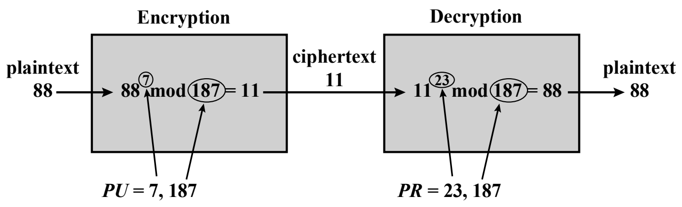
Solution

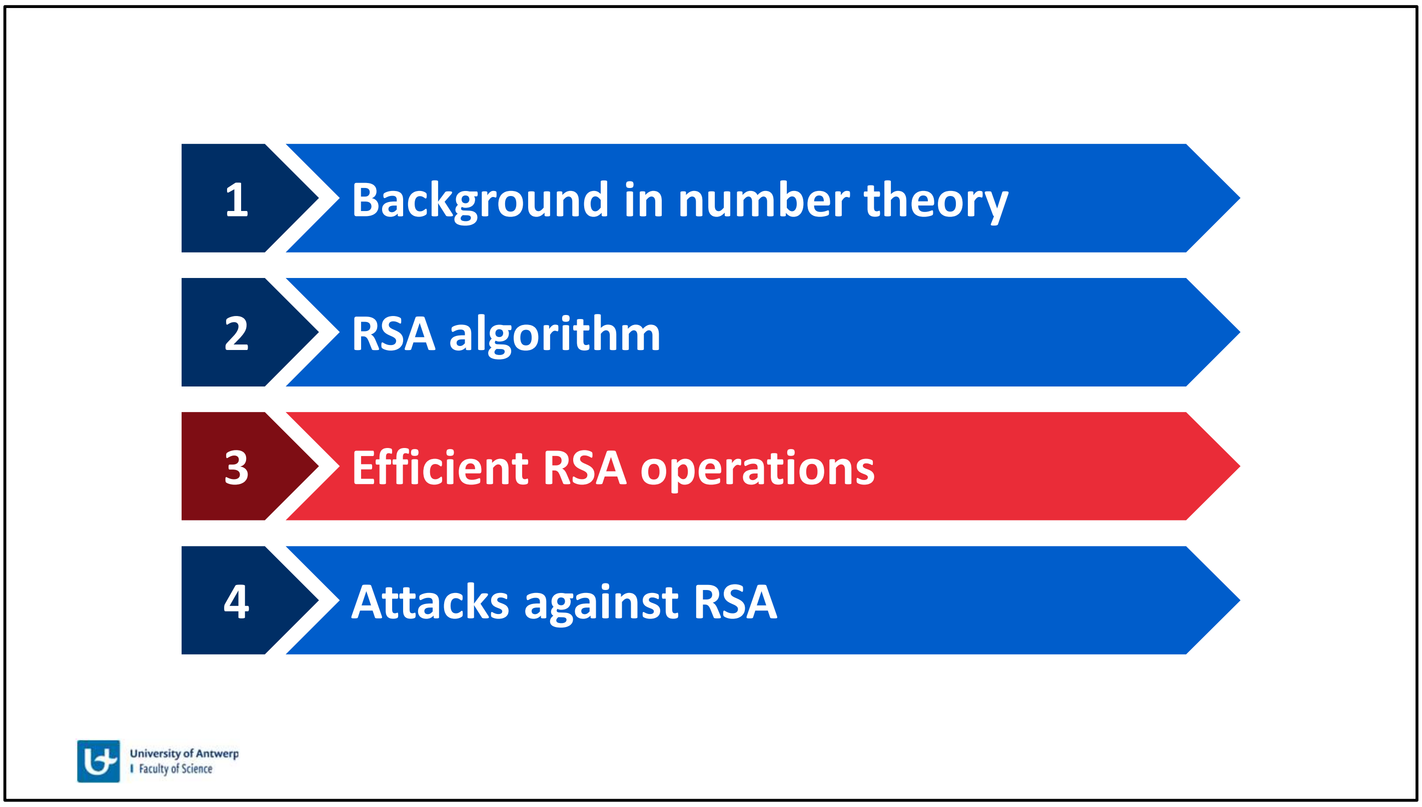
$11^{23}\ mod\ 187$

$= \left[(11^7 \bmod 187) \times (11^8 \bmod 187) \times (11^8 \bmod 187)\right] \bmod 187$

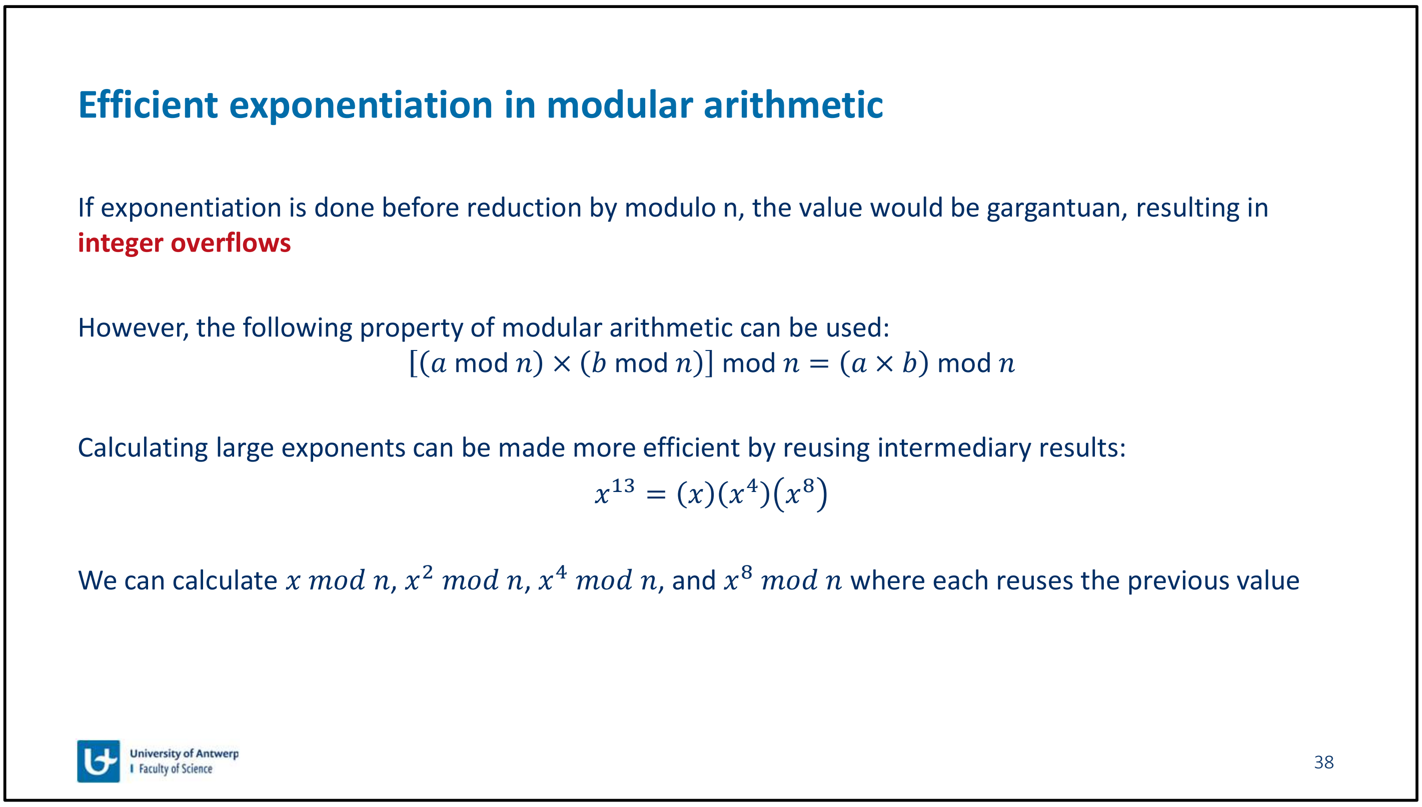
$= [88 \times 33 \times 33] \bmod 187$

$= 88 = M$





University of Antwerp
Faculty of Science



Efficient exponentiation in modular arithmetic

If exponentiation is done before reduction by modulo n, the value would be gargantuan, resulting in **integer overflows**

However, the following property of modular arithmetic can be used:

$$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$$

Calculating large exponents can be made more efficient by reusing intermediary results:

$$x^{13} = (x)(x^4)(x^8)$$

We can calculate $x\ mod\ n$, $x^2\ mod\ n$, $x^4\ mod\ n$, and $x^8\ mod\ n$ where each reuses the previous value

University of Antwerp
I Faculty of Science

38

We now turn to the issue of the complexity of the computation required to use RSA. There are actually two issues to consider: encryption/decryption and key generation. Let us look first at the process of encryption and decryption and then consider key generation.

Both encryption and decryption in RSA involve raising an integer to an integer power, mod *n*. If the exponentiation is done over the integers and then reduced modulo *n*, the intermediate values would be gargantuan. Fortunately, we can make use of a property of modular arithmetic:

$[(a \bmod n) * (b \bmod n)] \bmod n = (a * b) \bmod n$

Thus, we can reduce intermediate results modulo *n*. This makes the calculation practical.

Another consideration is the efficiency of exponentiation, because with RSA, we are dealing with potentially large exponents. To see how efficiency might be increased, consider that we wish to compute x^{16}. A straightforward approach requires 15 multiplications:

$x^{16} = x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x$

However, we can achieve the same final result with only four multiplications if we repeatedly take the square of each partial result, successively forming (x^2, x^4, x^8, x^{16}). As another example, suppose we wish to calculate $x^{11} \bmod n$ for some integers *x* and *n*. Observe that $x^{11} = x^{1+2+8} = (x)(x^2)(x^8)$. In this case, we compute $x \bmod n$, $x^2 \bmod n$, $x^4 \bmod n$, and $x^8 \bmod n$ and then calculate $[(x \bmod n) * (x^2 \bmod n) * (x^8 \bmod n)] \bmod n$.

Algorithm for efficient modular exponentiation

Goal Find $a^b \bmod n$

b can be expressed as a binary number $b_k b_{k-1} \dots b_1 b_0$, or

$$b = \sum_{b_i \neq 0} 2^i$$

Therefore,

$$a^b = a^{\left(\sum_{b_i \neq 0} 2^i\right)} = \prod_{b_i \neq 0} a^{(2^i)}$$

$$a^b \bmod n = \left[\prod_{b_i \neq 0} a^{(2^i)}\right] \bmod n = \left[\prod_{b_i \neq 0} \left(a^{(2^i)} \bmod n\right)\right] \bmod n$$

This equation can be turned into an algorithm

$$a^b \bmod n = \left[\prod_{b_i \neq 0} a^{(2^i)}\right] \bmod n = \left[\prod_{b_i \neq 0} \left(a^{(2^i)} \bmod n\right)\right] \bmod n$$

```
$f \leftarrow 1$
for $i \leftarrow k$ downto $0$ do
        $f \leftarrow (f \cdot f) \bmod n$
        if $b_i = 1$ then
                $f \leftarrow (f \cdot a) \bmod n$
return $f$
```

$$f = a^b \bmod n$$

Exercise: Efficient exponentiation (1)

Exercise: Calculate 7^{21} mod 127 using the efficient modular exponentiation algorithm.

```
$f \leftarrow 1$
for $i \leftarrow k$ downto $0$ do
        $f \leftarrow (f \cdot f) \bmod n$
        if $b_i = 1$ then
                $f \leftarrow (f \cdot a) \bmod n$
return $f$
```

Solution: Efficient exponentiation (1)

```
$f \leftarrow 1$
for $i \leftarrow k$ downto $0$ do
        $f \leftarrow (f \cdot f) \bmod n$
        if $b_i = 1$ then
                $f \leftarrow (f \cdot a) \bmod n$
return $f$
```

Solution

i	4	3	2	1	0
b_i	1	0	1	0	1
f	7	49	43	71	$\boxed{108}$

$7^{21} \bmod 127 = 108$

University of Antwerp
Faculty of Science

Exercise: Efficient exponentiation (2)

Exercise: Calculate 7^{560} mod 561 using the efficient modular exponentiation algorithm.

```
$f \leftarrow 1$
for $i \leftarrow k$ downto $0$ do
    $f \leftarrow (f \cdot f) \bmod n$
    if $b_i = 1$ then
        $f \leftarrow (f \cdot a) \bmod n$
return $f$
```

University of Antwerp
I Faculty of Science

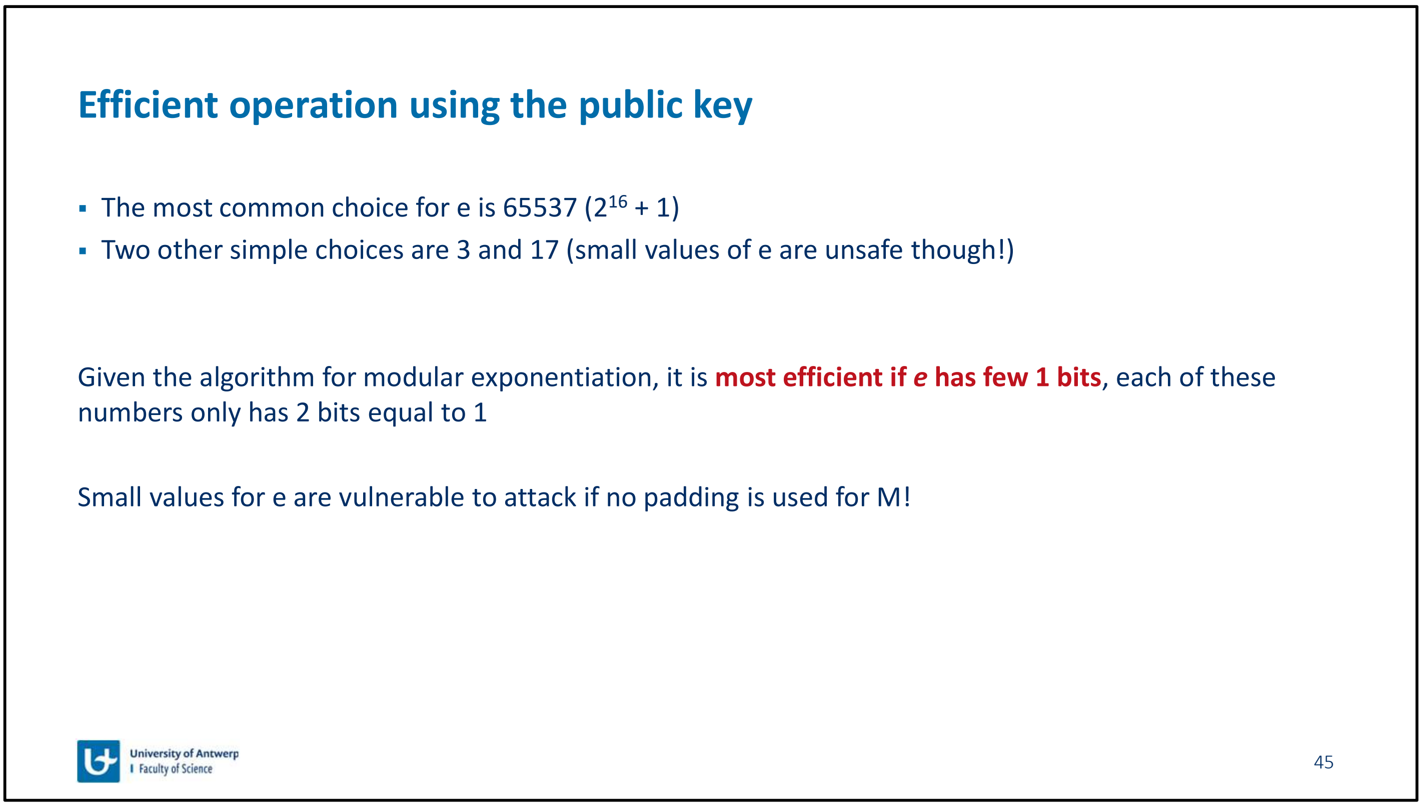
Solution: Efficient exponentiation (2)

```
$f \leftarrow 1$
for $i \leftarrow k$ downto $0$ do
    $f \leftarrow (f \cdot f) \bmod n$
    if $b_i = 1$ then
        $f \leftarrow (f \cdot a) \bmod n$
return $f$
```

Solution

$7^{560} \bmod 561 = 1$

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
f	7	49	157	526	160	241	298	166	67	1



Efficient operation using the public key

- The most common choice for e is 65537 (2^{16} + 1)
- Two other simple choices are 3 and 17 (small values of e are unsafe though!)

Given the algorithm for modular exponentiation, it is **most efficient if *e* has few 1 bits**, each of these numbers only has 2 bits equal to 1

Small values for e are vulnerable to attack if no padding is used for M!

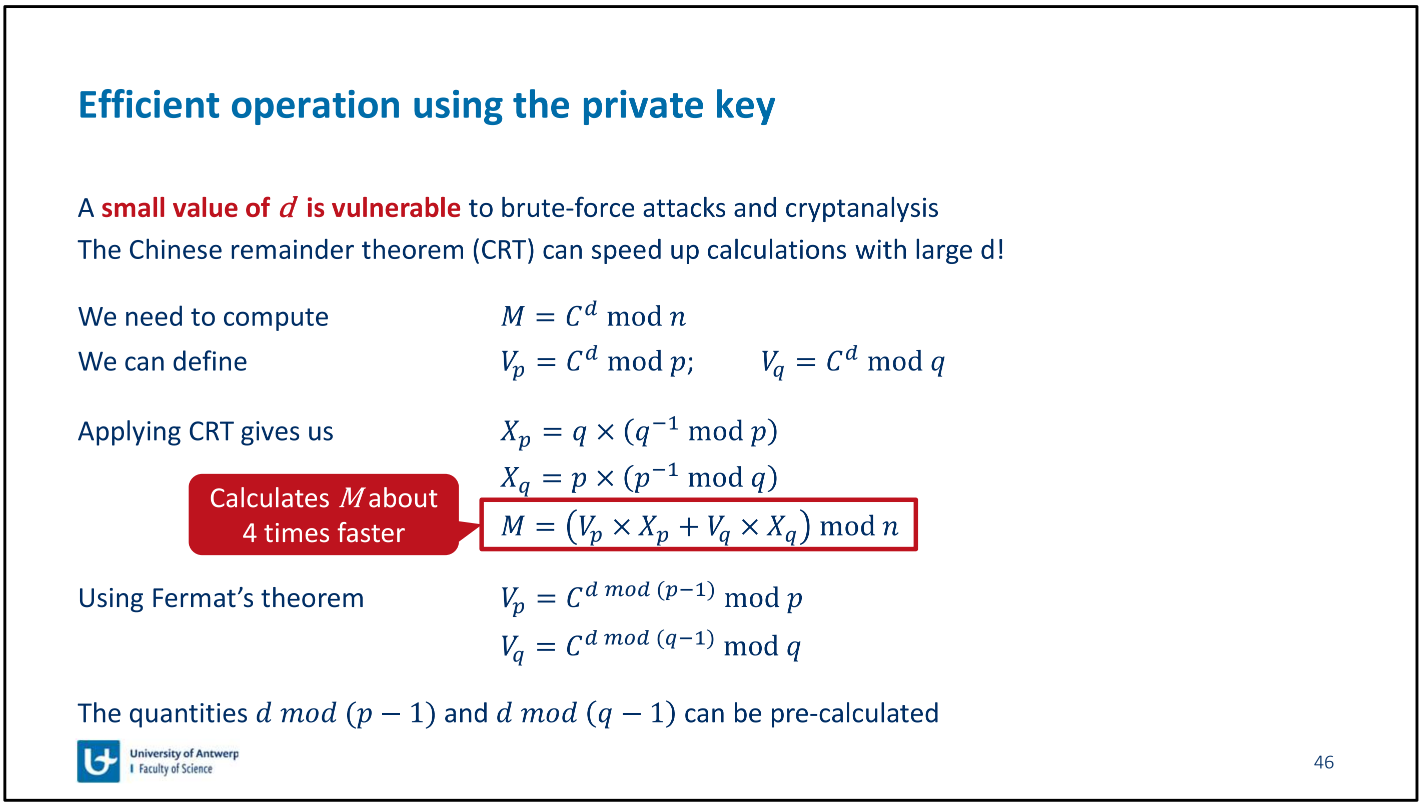
University of Antwerp
Faculty of Science

45

To speed up the operation of the RSA algorithm using the public key, a specific choice of e is usually made. The most common choice is 65537 (2^{16} + 1); two other popular choices are 3 and 17. Each of these choices has only two 1 bits, so the number of multiplications required to perform exponentiation is minimized.

However, with a very small public key, such as $e = 3$, RSA becomes vulnerable to a simple attack. Suppose we have three different RSA users who all use the value $e = 3$ but have unique values of n, namely (n_1, n_2, n_3). If user A sends the same encrypted message M to all three users, then the three ciphertexts are $C_1 = M^3 \bmod n_1$, $C_2 = M^3 \bmod n_2$, and $C_3 = M^3 \bmod n_3$. It is likely that n_1,n_2, and n_3 are pairwise relatively prime. Therefore, one can use the Chinese remainder theorem (CRT) to compute $M^3 \bmod (n_1\ n_2\ n_3)$. By the rules of the RSA algorithm, M is less than each of the n_i; therefore $M^3 < n_1 n_2 n_3$. Accordingly, the attacker need only compute the cube root of M^3. This attack can be countered by adding a unique pseudorandom bit string as padding to each instance of M to be encrypted.

The reader may have noted that the definition of the RSA algorithm requires that during key generation the user selects a value of e that is relatively prime to $\phi(n)$. Thus, if a value of e is selected first and the primes p and q are generated, it may turn out that gcd($\phi(n)$, e) $\neq$ 1. In that case, the user must reject the p, q values and generate a new p, q pair.



Efficient operation using the private key

A **small value of d is vulnerable** to brute-force attacks and cryptanalysis
The Chinese remainder theorem (CRT) can speed up calculations with large d!

We need to compute $M = C^d \bmod n$

We can define $V_p = C^d \bmod p; \quad V_q = C^d \bmod q$

Applying CRT gives us
$X_p = q \times (q^{-1} \bmod p)$
$X_q = p \times (p^{-1} \bmod q)$
$\boxed{M = (V_p \times X_p + V_q \times X_q) \bmod n}$

Calculates M about 4 times faster

Using Fermat's theorem
$V_p = C^{d \bmod (p-1)} \bmod p$
$V_q = C^{d \bmod (q-1)} \bmod q$

The quantities $d \bmod (p-1)$ and $d \bmod (q-1)$ can be pre-calculated

University of Antwerp | Faculty of Science

46

CRT says that given n = p x q, every M <= n can be represented as a tuple (m_1, m_2), where:

m_1 = M mod p
m_2 = M mod q

The definition of the bijection (CRT) states that:

$$X_p = \frac{n}{p} \times \left(\left(\frac{p}{n}\right) mod\ p\right) = \frac{p \times q}{p} \times \left(\left(\frac{p}{p \times q}\right) mod\ p\right) = q \times (q^{-1} mod\ p)$$

$$X_q = \frac{n}{q} \times \left(\left(\frac{q}{n}\right) mod\ q\right) = p \times (p^{-1} mod\ q)$$

The definition of the bijection further states that:
M = X_p x m_1 + X_q x m_2

Where m_1 and m_2, equal V_p and V_q.

$$m_1 = M\ mod\ p = (C^d\ mod\ n)\ mod\ p = C^d\ mod\ p = V_p$$

$$m_2 = M\ mod\ q = (C^d\ mod\ n)\ mod\ q = C^d\ mod\ q = V_q$$

Simplifying V_p, based on Fermat's theorem:

$$V_p = C^d\ mod\ p = C^{(p-1)x + (d\ mod\ (p-1))}\ mod\ p = C^{(p-1)x} \times C^{d\ mod\ (p-1)}\ mod\ p$$

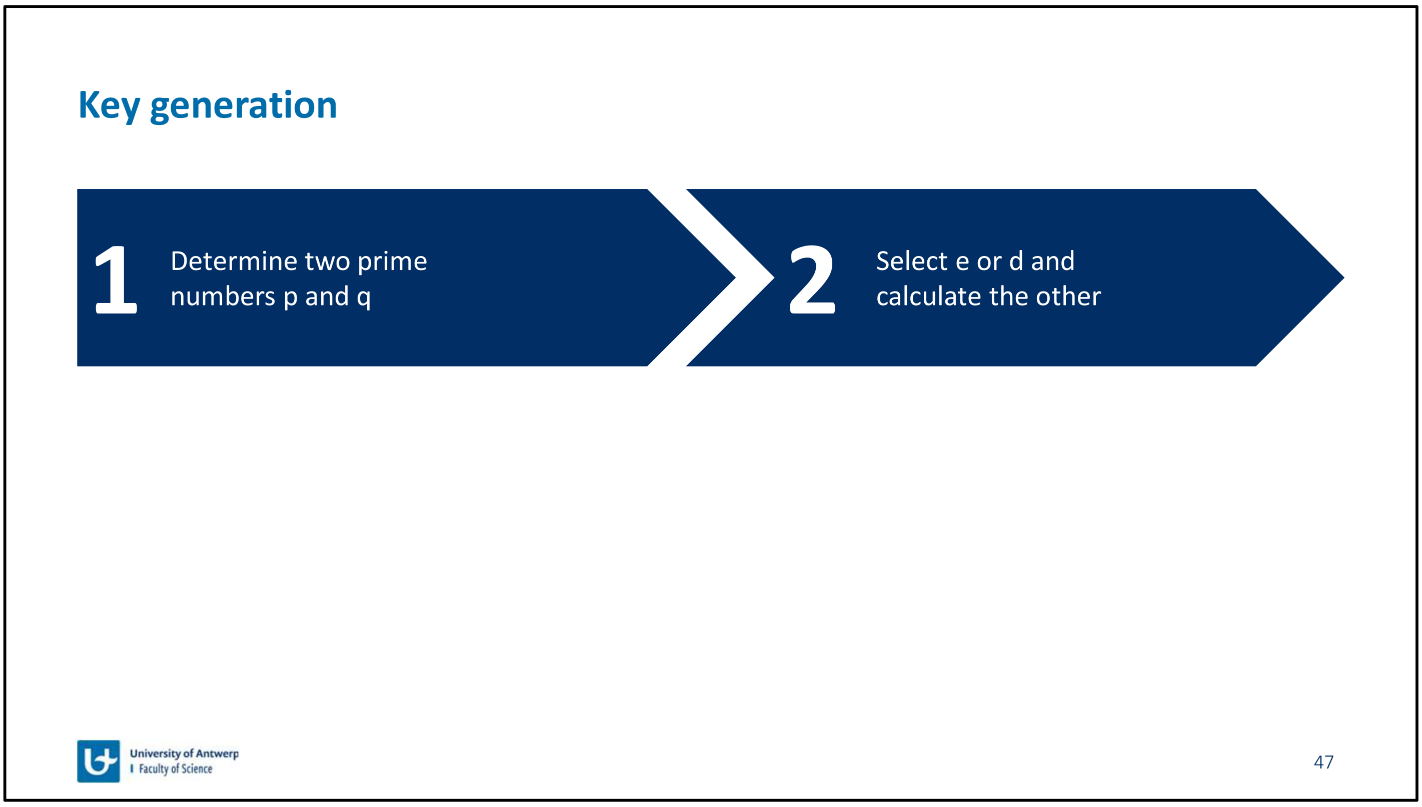
Fermat's theorem states that if C is a positive integer not divisible by p, then:

$C^{p-1} \equiv 1 (mod\ p)$ and by extension $C^{(p-1)x} \equiv 1^x \equiv 1 (mod\ p)$

Using this equality:

$$V_p = C^{(p-1)x} \times C^{d \bmod (p-1)} \equiv C^{d \bmod (p-1)} \pmod{p}$$

A similar proof can be devised for V_q



Key generation

1 Determine two prime numbers p and q

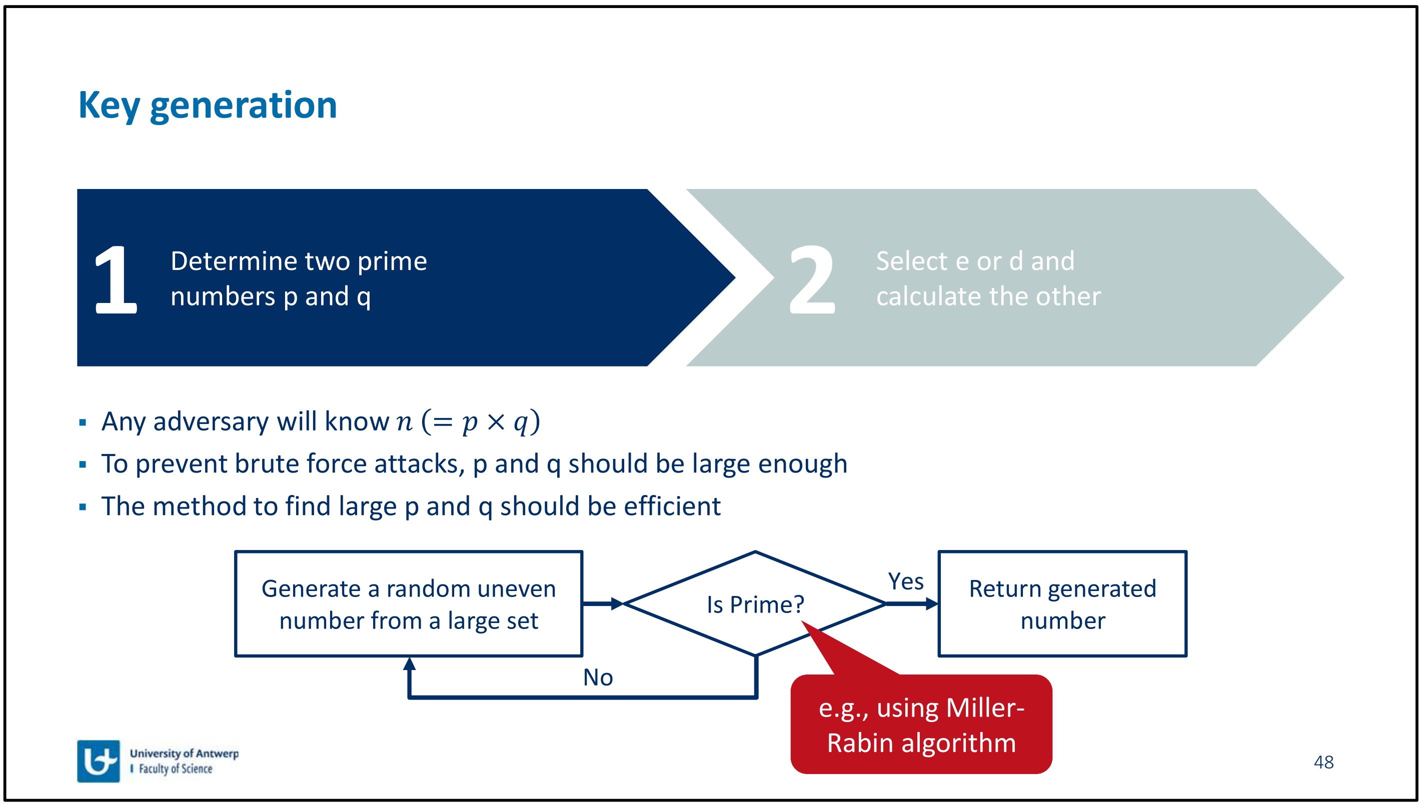
2 Select e or d and calculate the other

University of Antwerp | Faculty of Science

47

Before the application of the public-key cryptosystem, each participant must generate a pair of keys. This involves the following tasks:

- Determining two prime numbers, *p* and *q.*
- Selecting either *e* or *d* and calculating the other.



Key generation

1 Determine two prime numbers p and q

2 Select e or d and calculate the other

- Any adversary will know $n\ (= p \times q)$
- To prevent brute force attacks, p and q should be large enough
- The method to find large p and q should be efficient

Generate a random uneven number from a large set
Is Prime?
Yes
Return generated number
No
e.g., using Miller-Rabin algorithm

University of Antwerp | Faculty of Science

48

First, consider the selection of *p* and *q*. Because the value of *n* = *p x q* will be known to any potential adversary, in order to prevent the discovery of *p* and *q* by exhaustive methods, these primes must be chosen from a sufficiently large set (i.e., *p* and *q* must be large numbers). On the other hand, the method used for finding large primes must be reasonably efficient.

At present, there are no useful techniques that yield arbitrarily large primes, so some other means of tackling the problem is needed. The procedure that is generally used is to pick at random an odd number of the desired order of magnitude and test whether that number is prime. If not, pick successive random numbers until one is found that tests prime.

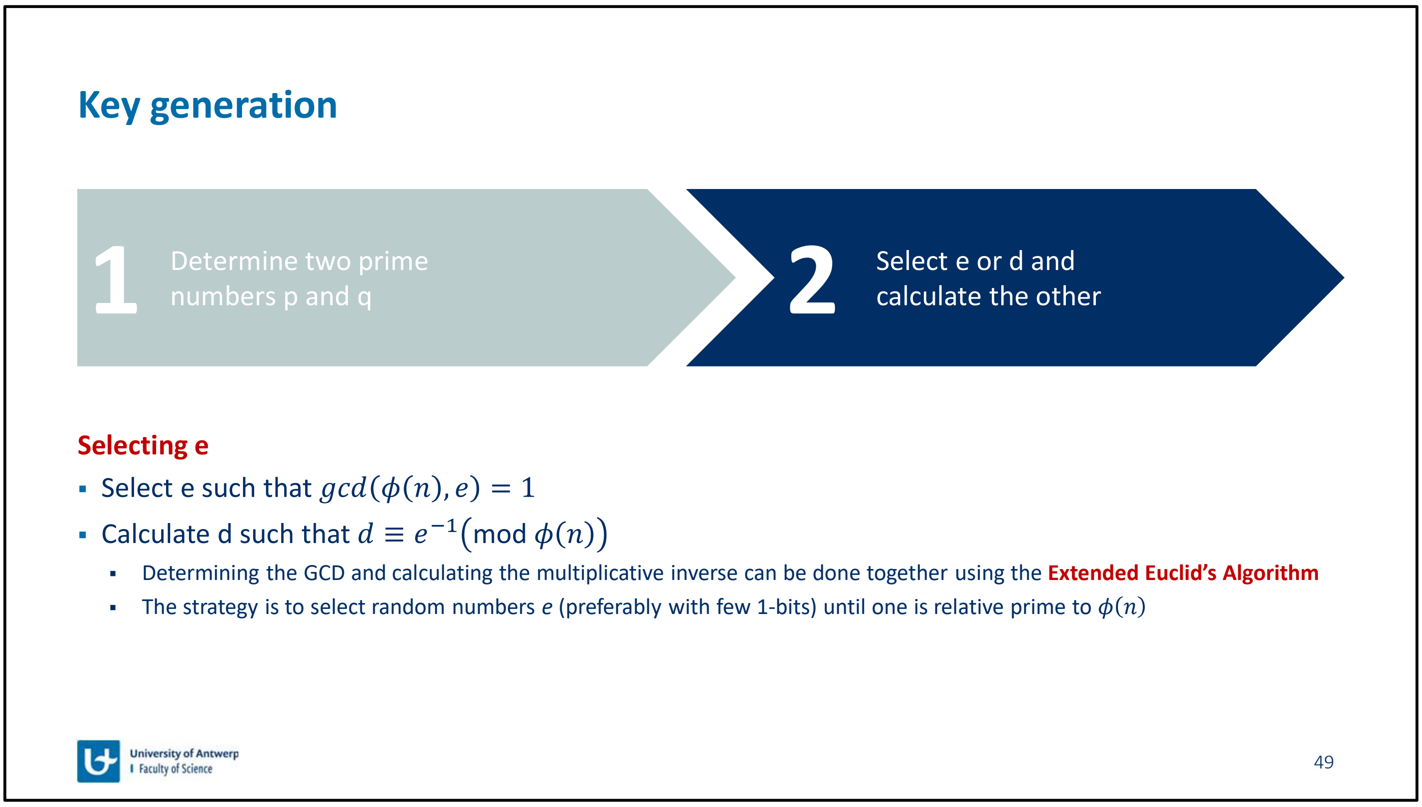
A variety of tests for primality have been developed. Almost invariably, the tests are probabilistic. That is, the test will merely determine that a given integer is *probably* prime. Despite this lack of certainty, these tests can be run in such a way as to make the probability as close to 1.0 as desired. As an example, one of the more efficient and popular algorithms is the Miller–Rabin algorithm. With this algorithm and most such algorithms, the procedure for testing whether a given integer *n* is prime is to perform some calculation that involves *n* and a randomly chosen integer *a*. If *n* "fails" the test, then *n* is not prime. If *n* "passes" the test, then *n* may be prime or nonprime. If *n* passes many such tests with many different randomly chosen values for *a*, then we can have high confidence that *n* is, in fact, prime.

In summary, the procedure for picking a prime number is as follows.
1. Pick an odd integer *n* at random (e.g., using a pseudorandom number generator).
2. Pick an integer *a* 6 *n* at random.

3. Perform the probabilistic primality test, such as Miller–Rabin, with *a* as a parameter. If *n* fails the test, reject the value *n* and go to step 1.
4. If *n* has passed a sufficient number of tests, accept *n*; otherwise, go to step 2.

This is a somewhat tedious procedure. However, remember that this process is performed relatively infrequently: only when a new pair (*PU*, *PR*) is needed.

It is worth noting how many numbers are likely to be rejected before a prime number is found. A result from number theory, known as the prime number theorem, states that the primes near *N* are spaced on the average one every ln(*N*) integers. Thus, on average, one would have to test on the order of ln(*N*) integers before a prime is found. Actually, because all even integers can be immediately rejected, the correct figure is ln(*N*)/2. For example, if a prime on the order of magnitude of 2^{200} were sought, then about $\ln(2^{200})/2 = 70$ trials would be needed to find a prime.



Key generation

1 Determine two prime numbers p and q

2 Select e or d and calculate the other

Selecting e

- Select e such that $gcd(\phi(n), e) = 1$
- Calculate d such that $d \equiv e^{-1}(\text{mod } \phi(n))$
  - Determining the GCD and calculating the multiplicative inverse can be done together using the **Extended Euclid's Algorithm**
  - The strategy is to select random numbers *e* (preferably with few 1-bits) until one is relative prime to $\phi(n)$

University of Antwerp
Faculty of Science

49

Having determined prime numbers *p* and *q*, the process of key generation is completed by selecting a value of *e* and calculating *d* or, alternatively, selecting a value of *d* and calculating *e*. Assuming the former, then we need to select an *e* such that gcd($\phi(n)$, *e*) = 1 and then calculate *d* = e^{-1} (mod $\phi(n)$). Fortunately, there is a single algorithm that will, at the same time, calculate the greatest common divisor of two integers and, if the gcd is 1, determine the inverse of one of the integers modulo the other. The algorithm is referred to as the extended Euclid's algorithm. Thus, the procedure is to generate a series of random numbers, testing each against $\phi(n)$ until a number relatively prime to $\phi(n)$ is found. Again, we can ask the question: How many random numbers must we test to find a usable number, that is, a number relatively prime to $\phi(n)$? It can be shown easily that the probability that two random numbers are relatively prime is about 0.6; thus, very few tests would be needed to find a suitable integer.

Exercise: Selecting *e* and *d*

Exercise: Given p = 7 and q = 11, determine the highest suitable value for e and calculate d.

Relevant formulas

$n = p \times q$

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

$gcd(\phi(n), e) = 1$

$d \equiv e^{-1}(\text{mod } \phi(n))$

Initialization

$r_{-1} = \phi(n)$ $\quad r_0 = e$

$w_{-1} = 0$ $\quad w_0 = 1$

Recursive algorithm

$r_i = r_{i-2} \text{ mod } r_{i-1}$

$q_i = r_{i-2} / r_{i-1}$

$w_i = w_{i-2} - q_i w_{i-1}$

Stop condition

$\text{if } r_i = 1$ $\quad \text{then } w_i = e^{-1} = d$

Solution: Selecting *e* and *d*

Step 1: Finding n and $\phi(n)$

$n = p \times q = 77$

$\phi(n) = (p-1)(q-1) = 60$

Step 2: Finding suitable values for *e*

$gcd(\phi(n), e) = 1$ for e = 59, 53, 49, 47, 43, …, 11, 7, 1

Step 3: Finding d

$d \equiv e^{-1} (\text{mod } \phi(n))$

For e = 59, the extended Euclidian algorithm shows that d = 59

e = d = 59 is **insecure**!

University of Antwerp
Faculty of Science

Solution: Selecting *e* and *d*

Step 4: Finding a better e and d

$d \equiv e^{-1} \pmod{\phi(n)}$

Let us find a solution for e = 53 using the extended Euclidian algorithm

iteration	-1	0
r	60	53
q	--	--
w	0	1

University of Antwerp
Faculty of Science

Solution: Selecting *e* and *d*

Step 4: Finding a better e and d

$d \equiv e^{-1}(\text{mod } \phi(n))$

Let us find a solution for e = 53 using the extended Euclidian algorithm

iteration	-1	0	1
r	60	53	7
q	--	--	1
w	0	1	59

University of Antwerp
Faculty of Science

Solution: Selecting *e* and *d*

Step 4: Finding a better e and d

$d \equiv e^{-1} \pmod{\phi(n)}$

Let us find a solution for e = 53 using the extended Euclidian algorithm

iteration	**-1**	**0**	**1**	**2**
r	60	53	7	4
q	--	--	1	7
w	0	1	59	8

Solution: Selecting *e* and *d*

Step 4: Finding a better e and d

$d \equiv e^{-1}\left(\bmod \phi(n)\right)$

Let us find a solution for e = 53 using the extended Euclidian algorithm

iteration	-1	0	1	2	3
r	60	53	7	4	3
q	--	--	1	7	1
w	0	1	59	8	51

University of Antwerp
Faculty of Science

Solution: Selecting *e* and *d*

Step 4: Finding a better e and d

$d \equiv e^{-1} \pmod{\phi(n)}$

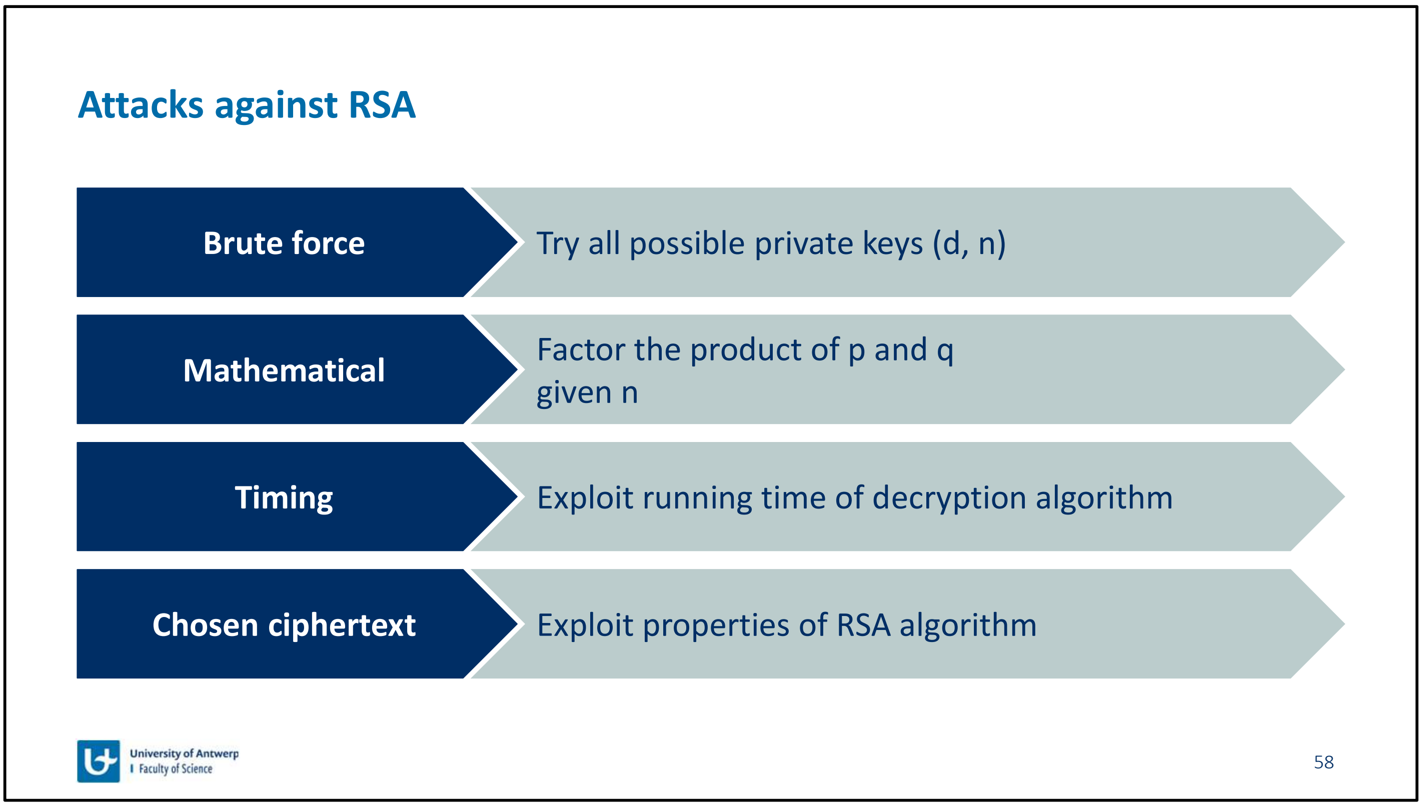
Let us find a solution for e = 53 using the extended Euclidian algorithm

iteration	-1	0	1	2	3	4
r	60	53	7	4	3	1
q	--	--	1	7	1	1
w	0	1	59	8	51	$\boxed{17}$

$e = 53, d = 17$

University of Antwerp
I Faculty of Science

University of Antwerp
Faculty of Science



Attacks against RSA

Brute force	Try all possible private keys (d, n)
Mathematical	Factor the product of p and q given n
Timing	Exploit running time of decryption algorithm
Chosen ciphertext	Exploit properties of RSA algorithm

University of Antwerp | Faculty of Science

58

Four possible approaches to attacking the RSA algorithm are:

- **Brute force:** This involves trying all possible private keys.
- **Mathematical attacks:** There are several approaches, all equivalent in effort to factoring the product of two primes.
- **Timing attacks:** These depend on the running time of the decryption algorithm.
- **Chosen ciphertext attacks:** This type of attack exploits properties of the RSA algorithm.

The defense against the brute-force approach is the same for RSA as for other cryptosystems, namely, to use a large key space. Thus, the larger the number of bits in *d*, the better. However, because the calculations involved, both in key generation and in encryption/decryption, are complex, the larger the size of the key, the slower the system will run.

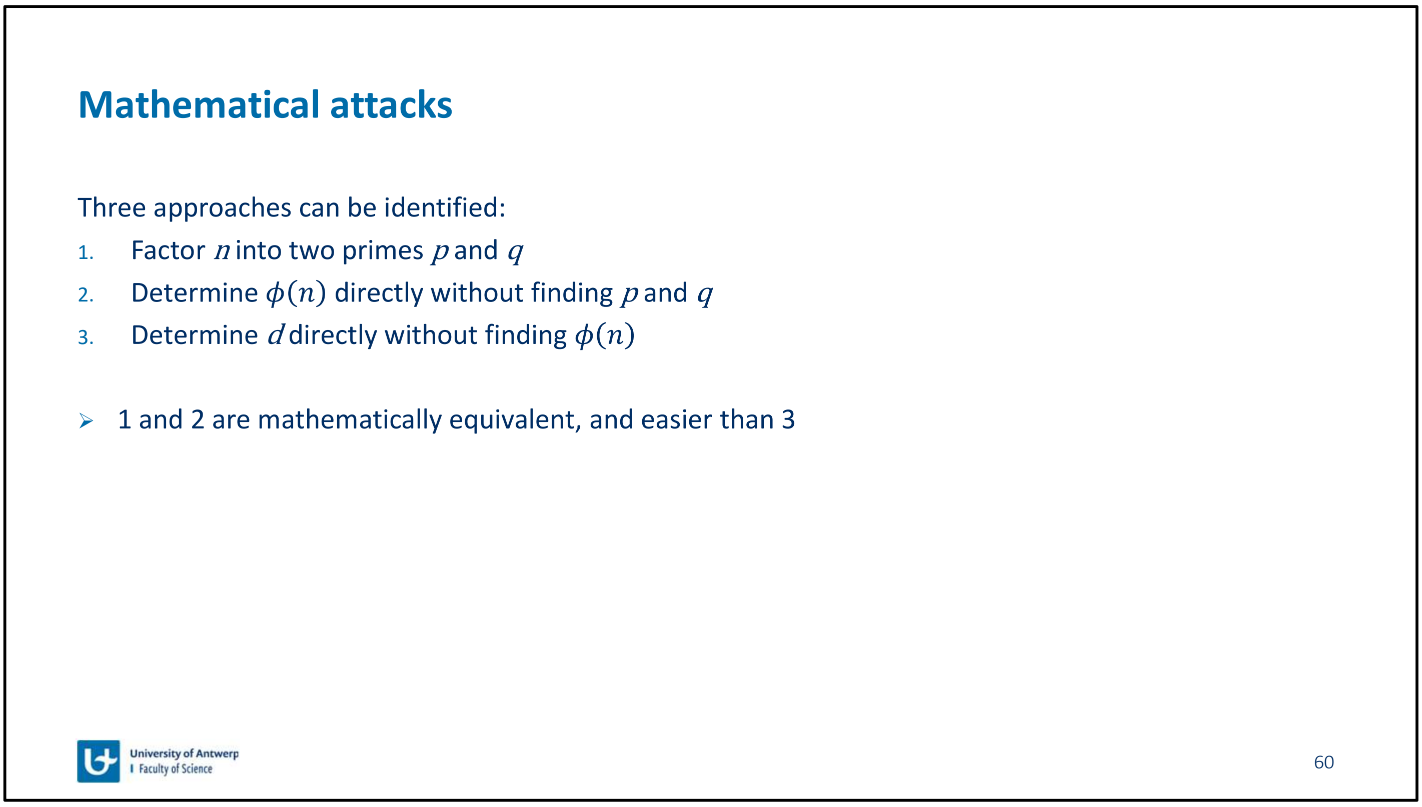
Comparison of cipher security for brute force

Equivalence of key sizes in terms of computational effort for crypto analysis for various encryption algorithms

Symmetric Key Algorithms	**Diffie-Hellman, Digital Signature Algorithm**	**RSA (size of *n* in bits)**	**ECC (modulus size in bits)**
80	$L = 1024$ $N = 160$	1024	160–223
112	$L = 2048$ $N = 224$	2048	224–255
128	$L = 3072$ $N = 256$	3072	256–383
192	$L = 7680$ $N = 384$	7680	384–511
256	$L = 15{,}360$ $N = 512$	15,360	512+

Note: L = size of public key, N = size of private key

Minimum key sizes considered safe up to 2030 (November 2014 report)



Mathematical attacks

Three approaches can be identified:

1. Factor n into two primes p and q
2. Determine $\phi(n)$ directly without finding p and q
3. Determine d directly without finding $\phi(n)$

➢ 1 and 2 are mathematically equivalent, and easier than 3

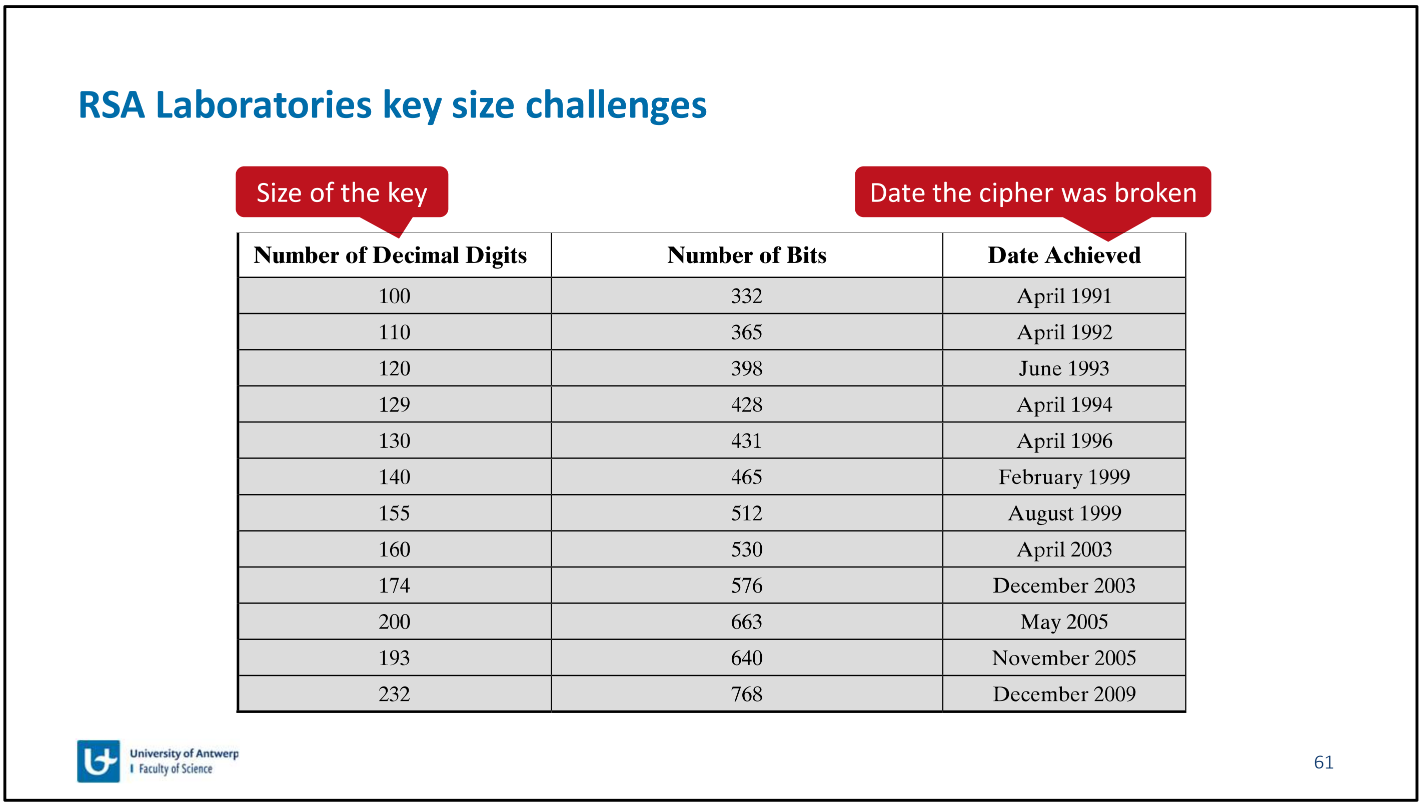
University of Antwerp | Faculty of Science

60

We can identify three approaches to attacking RSA mathematically.

1. Factor *n* into its two prime factors. This enables calculation of $\phi(n)$ = (*p* - 1) * (*q* - 1), which in turn enables determination of *d* = e^{-1} (mod $\phi(n)$).
2. Determine $\phi(n)$ directly, without first determining *p* and *q*. Again, this enables determination of *d* = e^{-1} (mod $\phi(n)$).
3. Determine *d* directly, without first determining $\phi(n)$.

Most discussions of the cryptanalysis of RSA have focused on the task of factoring *n* into its two prime factors. Determining $\phi(n)$ given *n* is equivalent to factoring *n*. With presently known algorithms, determining *d* given *e* and *n* appears to be at least as time-consuming as the factoring problem. Hence, we can use factoring performance as a benchmark against which to evaluate the security of RSA.



RSA Laboratories key size challenges

Size of the key

Date the cipher was broken

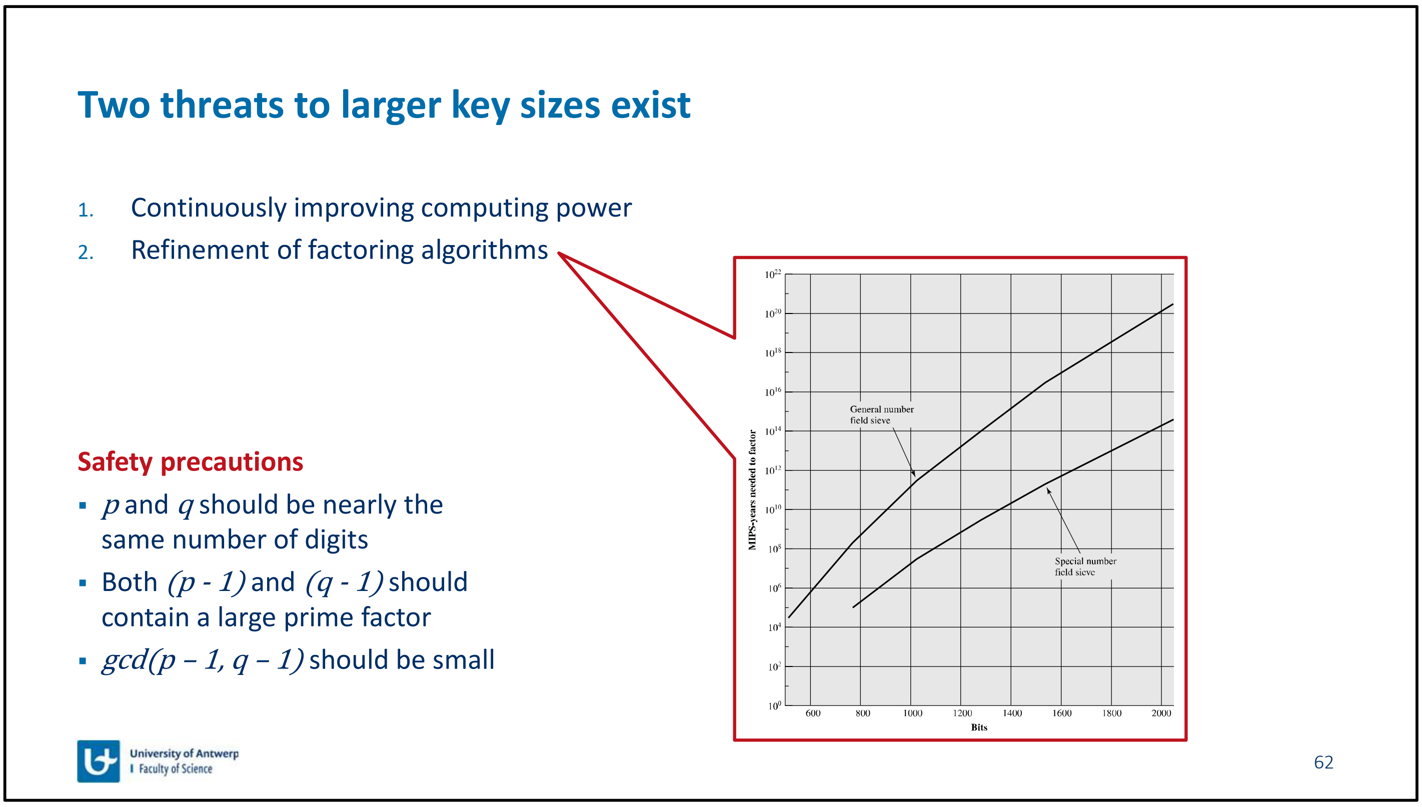
Number of Decimal Digits	Number of Bits	Date Achieved
100	332	April 1991
110	365	April 1992
120	398	June 1993
129	428	April 1994
130	431	April 1996
140	465	February 1999
155	512	August 1999
160	530	April 2003
174	576	December 2003
200	663	May 2005
193	640	November 2005
232	768	December 2009

University of Antwerp
Faculty of Science

61

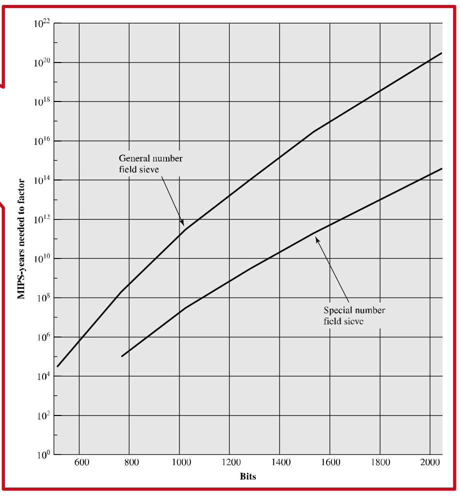
For a large *n* with large prime factors, factoring is a hard problem, but it is not as hard as it used to be. A striking illustration of this is the following. In 1977, the three inventors of RSA dared *Scientific American* readers to decode a cipher they printed in Martin Gardner's "Mathematical Games" column. They offered a $100 reward for the return of a plaintext sentence, an event they predicted might not occur for some 40 quadrillion years. In April of 1994, a group working over the Internet claimed the prize after only eight months of work. This challenge used a public key size (length of *n*) of 129 decimal digits, or around 428 bits. In the meantime, just as they had done for DES, RSA Laboratories had issued challenges for the RSA cipher with key sizes of 100, 110, 120, and so on, digits. The latest challenge to be met is the RSA-768 challenge with a key length of 232 decimal digits, or 768 bits.

A striking fact about the progress reflected in the table concerns the method used. Until the mid-1990s, factoring attacks were made using an approach known as the quadratic sieve. The attack on RSA-130 used a newer algorithm, the generalized number field sieve (GNFS), and was able to factor a larger number than RSA-129 at only 20% of the computing effort.



Two threats to larger key sizes exist

1. Continuously improving computing power
2. Refinement of factoring algorithms



Safety precautions

- *p* and *q* should be nearly the same number of digits
- Both *(p - 1)* and *(q - 1)* should contain a large prime factor
- *gcd(p – 1, q – 1)* should be small

University of Antwerp
Faculty of Science

62

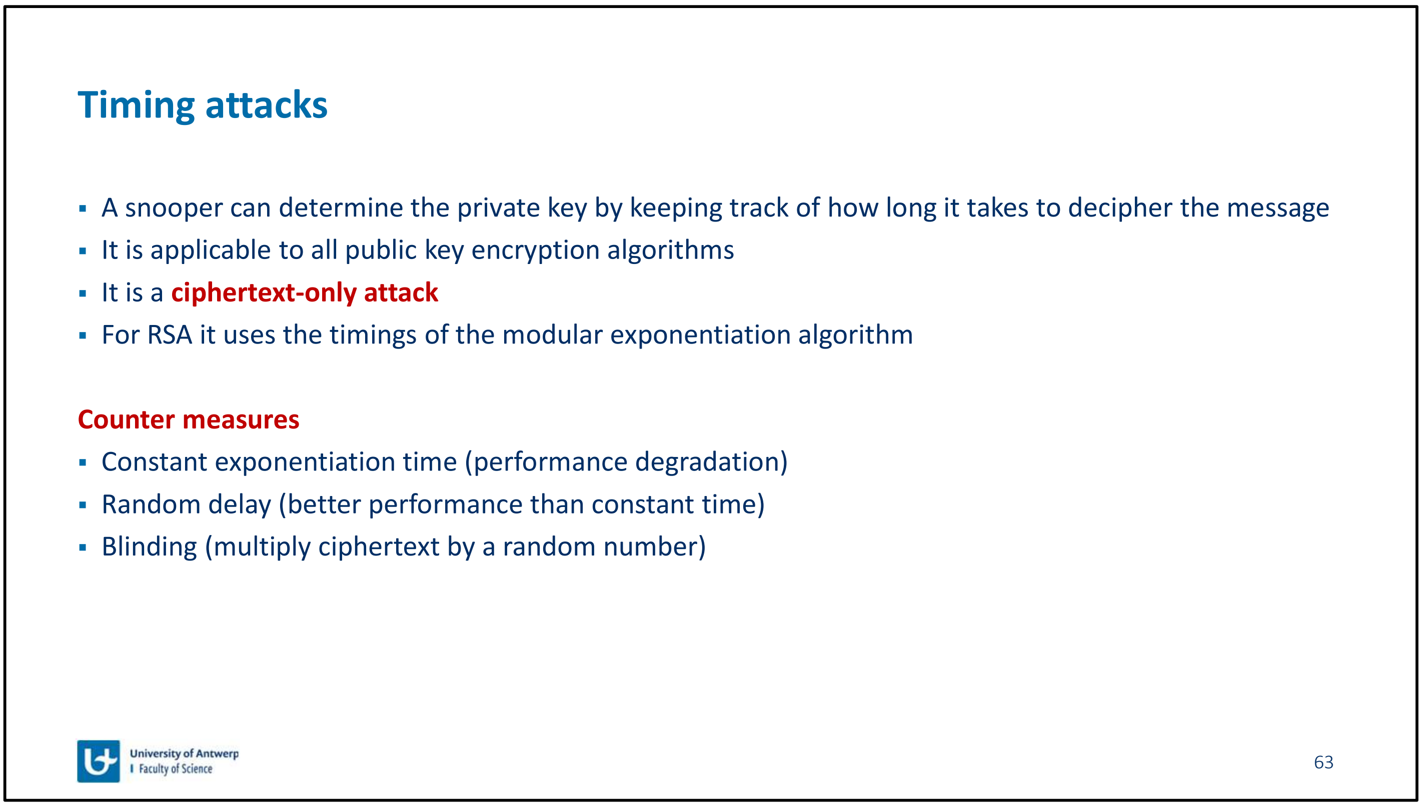
The threat to larger key sizes is twofold: the continuing increase in computing power and the continuing refinement of factoring algorithms. We have seen that the move to a different algorithm resulted in a tremendous speedup. We can expect further refinements in the GNFS, and the use of an even better algorithm is also a possibility. In fact, a related algorithm, the special number field sieve (SNFS), can factor numbers with a specialized form considerably faster than the generalized number field sieve.

It is reasonable to expect a breakthrough that would enable a general factoring performance in about the same time as SNFS, or even better. Thus, we need to be careful in choosing a key size for RSA. The team that produced the 768-bit factorization observed that factoring a 1024-bit RSA modulus would be about a thousand times harder than factoring a 768-bit modulus, and a 768-bit RSA modulus is several thousands times harder to factor than a 512-bit one. Based on the amount of time between the 512-bit and 768-bit factorization successes, the team felt it to be reasonable to expect that the 1024-bit RSA moduli could be factored well within the next decade by a similar academic effort. Thus, they recommended phasing out usage of 1024-bit RSA within the next few years (from 2010).

In addition to specifying the size of *n*, a number of other constraints have been suggested by researchers. To avoid values of *n* that may be factored more easily, the algorithm's inventors suggest the following constraints on *p* and *q*:

1. *p* and *q* should differ in length by only a few digits. Thus, for a 1024-bit key (309 decimal digits), both *p* and *q* should be on the order of magnitude of 10^{75} to 10^{100}.
2. Both (*p* - 1) and (*q* - 1) should contain a large prime factor.
3. gcd(*p* - 1, *q* - 1) should be small.

In addition, it has been demonstrated that if $e < n$ and $d < n^{1/4}$, then d can be easily determined.



Timing attacks

- A snooper can determine the private key by keeping track of how long it takes to decipher the message
- It is applicable to all public key encryption algorithms
- It is a **ciphertext-only attack**
- For RSA it uses the timings of the modular exponentiation algorithm

Counter measures

- Constant exponentiation time (performance degradation)
- Random delay (better performance than constant time)
- Blinding (multiply ciphertext by a random number)

University of Antwerp | Faculty of Science

63

If one needed yet another lesson about how difficult it is to assess the security of a cryptographic algorithm, the appearance of timing attacks provides a stunning one. Paul Kocher, a cryptographic consultant, demonstrated that a snooper can determine a private key by keeping track of how long a computer takes to decipher messages. Timing attacks are applicable not just to RSA, but to other public-key cryptography systems. This attack is alarming for two reasons: It comes from a completely unexpected direction, and it is a ciphertext-only attack.

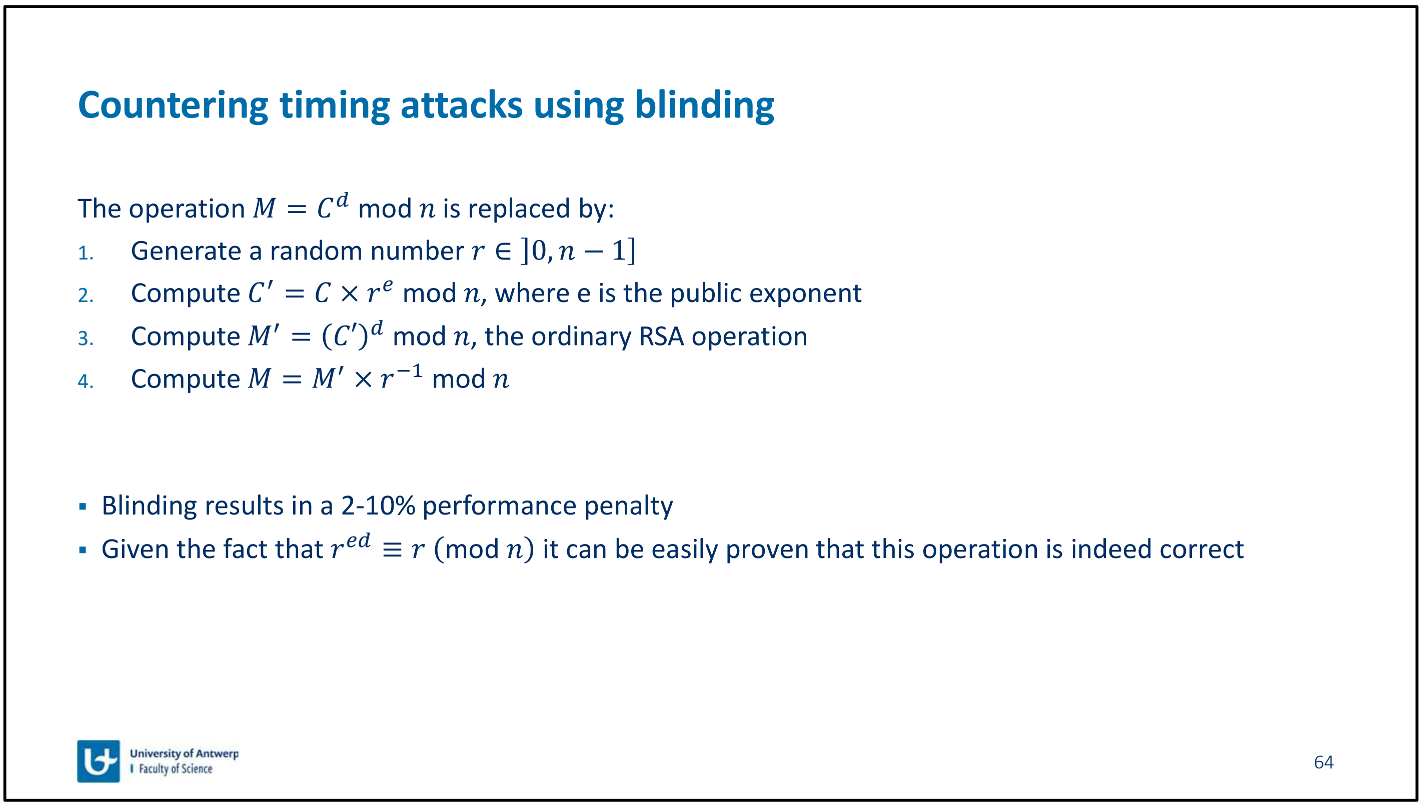
A **timing attack** is somewhat analogous to a burglar guessing the combination of a safe by observing how long it takes for someone to turn the dial from number to number. We can explain the attack using the modular exponentiation algorithm, but the attack can be adapted to work with any implementation that does not run in fixed time. In this algorithm, modular exponentiation is accomplished bit by bit, with one modular multiplication performed at each iteration and an additional modular multiplication performed for each 1 bit.

As Kocher points out in his paper, the attack is simplest to understand in an extreme case. Suppose the target system uses a modular multiplication function that is very fast in almost all cases but in a few cases takes much more time than an entire average modular exponentiation. The attack proceeds bit-by-bit starting with the leftmost bit, b_k. Suppose that the first j bits are known (to obtain the entire exponent, start with $j = 0$ and repeat the attack until the entire exponent is known). For a given ciphertext, the attacker can complete the first j iterations of the **for** loop. The operation of the subsequent step depends on the unknown exponent bit. If the bit is set, $d \leftarrow (d \times a) \bmod n$ will be executed. For a few values of a and d, the modular multiplication will be extremely slow, and the attacker knows which these are. Therefore, if the observed time to execute the decryption algorithm is always slow when this particular iteration is slow with a 1 bit, then this bit is assumed to be 1. If a number of observed execution times for the entire algorithm are fast, then this bit is assumed to be 0.

In practice, modular exponentiation implementations do not have such extreme timing variations, in which the execution time of a single iteration can exceed the mean execution time of the entire algorithm. Nevertheless, there is enough variation to make this attack practical.

Although the timing attack is a serious threat, there are simple countermeasures that can be used, including the following:

- **Constant exponentiation time:** Ensure that all exponentiations take the same amount of time before returning a result. This is a simple fix but does degrade performance.
- **Random delay:** Better performance could be achieved by adding a random delay to the exponentiation algorithm to confuse the timing attack. Kocher points out that if defenders don't add enough noise, attackers could still succeed by collecting additional measurements to compensate for the random delays.
- **Blinding:** Multiply the ciphertext by a random number before performing exponentiation. This process prevents the attacker from knowing what cipher-text bits are being processed inside the computer and therefore prevents the bit-by-bit analysis essential to the timing attack.



Countering timing attacks using blinding

The operation $M = C^d \bmod n$ is replaced by:

1. Generate a random number $r \in]0, n-1]$
2. Compute $C' = C \times r^e \bmod n$, where e is the public exponent
3. Compute $M' = (C')^d \bmod n$, the ordinary RSA operation
4. Compute $M = M' \times r^{-1} \bmod n$

- Blinding results in a 2-10% performance penalty
- Given the fact that $r^{ed} \equiv r \pmod{n}$ it can be easily proven that this operation is indeed correct

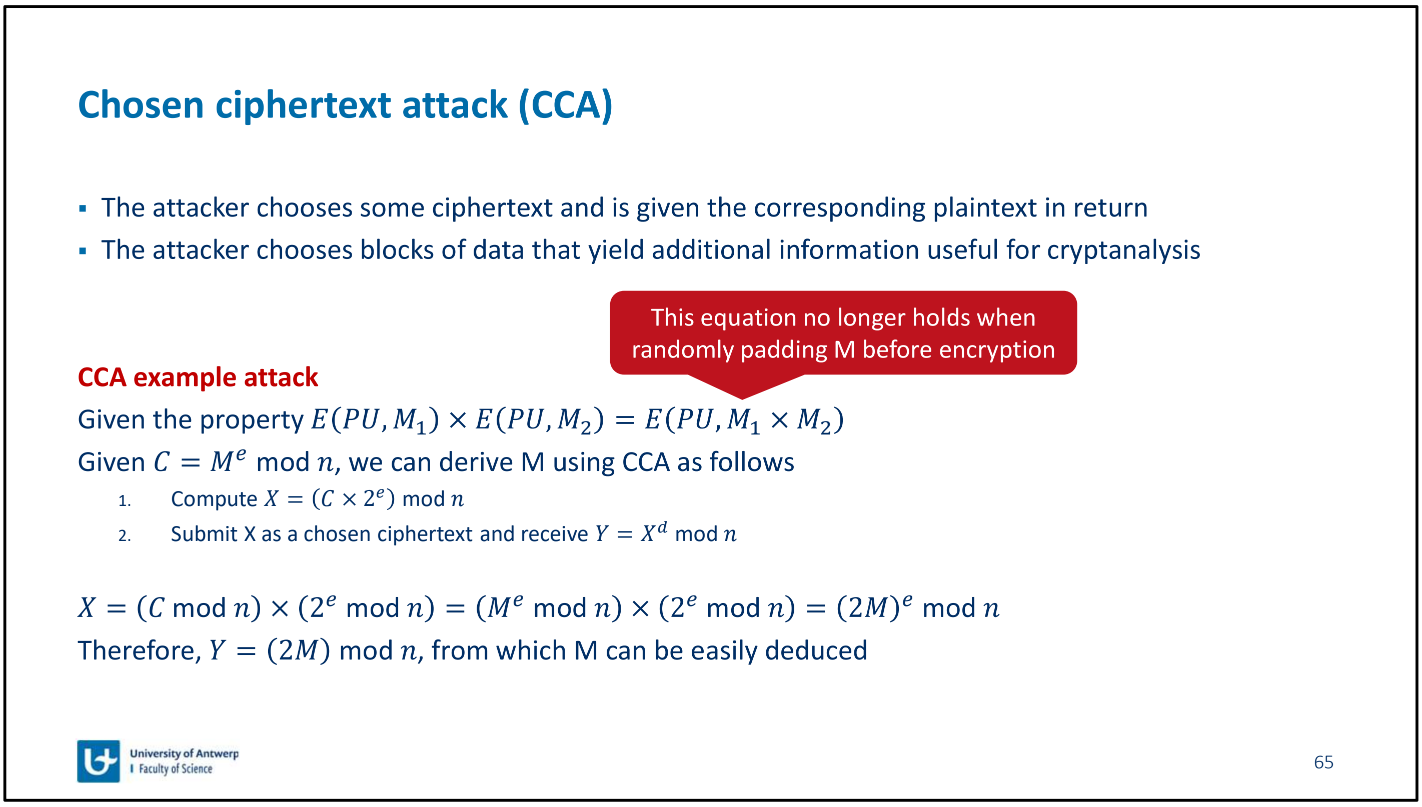
University of Antwerp
Faculty of Science

64

RSA Data Security incorporates a blinding feature into some of its products. The private-key operation $M = C^d \bmod n$ is implemented as follows.

1. Generate a secret random number r between 0 and n - 1.
2. Compute $C' = C (r^e) \bmod n$, where e is the public exponent.
3. Compute $M' = (C')^d \bmod n$ with the ordinary RSA implementation.
4. Compute $M = M' r^{-1} \bmod n$. In this equation, r^{-1} is the multiplicative inverse of r mod n.

It can be demonstrated that this is the correct result by observing that $r^{ed} \bmod n = r \bmod n$. RSA Data Security reports a 2 to 10% performance penalty for blinding.



Chosen ciphertext attack (CCA)

- The attacker chooses some ciphertext and is given the corresponding plaintext in return
- The attacker chooses blocks of data that yield additional information useful for cryptanalysis

This equation no longer holds when randomly padding M before encryption

CCA example attack

Given the property $E(PU, M_1) \times E(PU, M_2) = E(PU, M_1 \times M_2)$

Given $C = M^e \bmod n$, we can derive M using CCA as follows

1. Compute $X = (C \times 2^e) \bmod n$
2. Submit X as a chosen ciphertext and receive $Y = X^d \bmod n$

$X = (C \bmod n) \times (2^e \bmod n) = (M^e \bmod n) \times (2^e \bmod n) = (2M)^e \bmod n$

Therefore, $Y = (2M) \bmod n$, from which M can be easily deduced

University of Antwerp
Faculty of Science

65

The basic RSA algorithm is vulnerable to a **chosen ciphertext attack (CCA)**. CCA is defined as an attack in which the adversary chooses a number of ciphertexts and is then given the corresponding plaintexts, decrypted with the target's private key. Thus, the adversary could select a plaintext, encrypt it with the target's public key, and then be able to get the plaintext back by having it decrypted with the private key. Clearly, this provides the adversary with no new information. Instead, the adversary exploits properties of RSA and selects blocks of data that, when processed using the target's private key, yield information needed for cryptanalysis.

A simple example of a CCA against RSA takes advantage of the following property of RSA:

$$E(PU, M1) * E(PU, M2) = E(PU, [M1 * M2])$$

We can decrypt $C = M^e \bmod n$ using a CCA as follows:

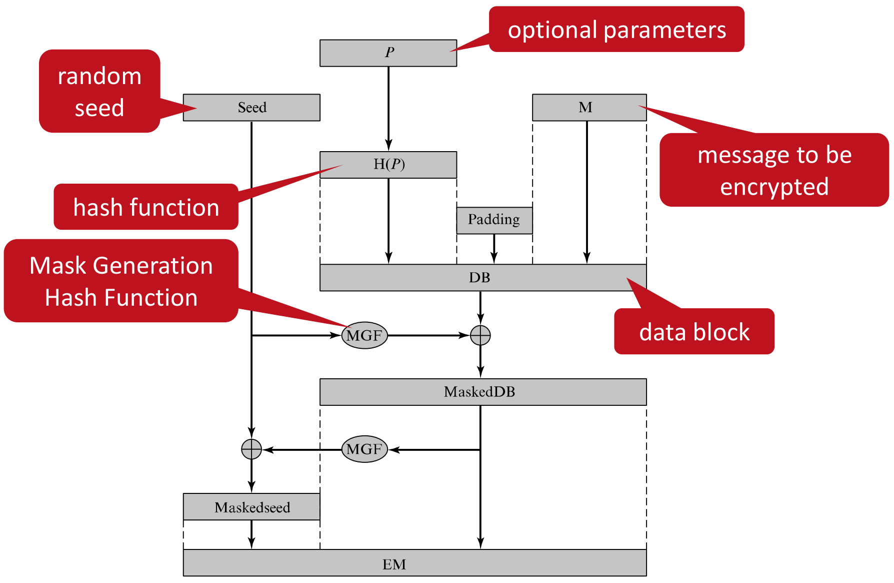
1. Compute $X = (C * 2^e) \bmod n$.
2. Submit X as a chosen ciphertext and receive back $Y = X^d \bmod n$.

But now note that:

$$X = (C \bmod n) * (2^e \bmod n) = (M^e \bmod n) * (2^e \bmod n) = (2M)^e \bmod n$$

Therefore, $Y = (2M) \bmod n$. From this, we can deduce M.

Optimal asymmetric encryption padding (OAEP)



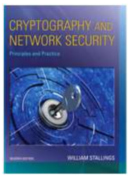
To overcome the CCA attack, practical RSA-based cryptosystems randomly pad the plaintext prior to encryption. This randomizes the ciphertext so that E(PU, M_1) * E(PU, M_2) = E(PU, [M_1 * M_2]) no longer holds. However, more sophisticated CCAs are possible, and a simple padding with a random value has been shown to be insufficient to provide the desired security. To counter such attacks, RSA Security Inc., a leading RSA vendor and former holder of the RSA patent, recommends modifying the plaintext using a procedure known as **optimal asymmetric encryption padding** (OAEP).

As a first step, the message M to be encrypted is padded. A set of optional parameters, P, is passed through a hash function, H. The output is then padded with zeros to get the desired length in the overall data block (DB). Next, a random seed is generated and passed through another hash function, called the mask generating function (MGF). The resulting hash value is bit-by-bit XORed with DB to produce a maskedDB. The maskedDB is in turn passed through the MGF to form a hash that is XORed with the seed to produce the maskedseed. The concatenation of the maskedseed and the maskedDB forms the encoded message EM. Note that the EM includes the padded message, masked by the seed, and the seed, masked by the maskedDB. The EM is then encrypted using RSA.

Summary on RSA

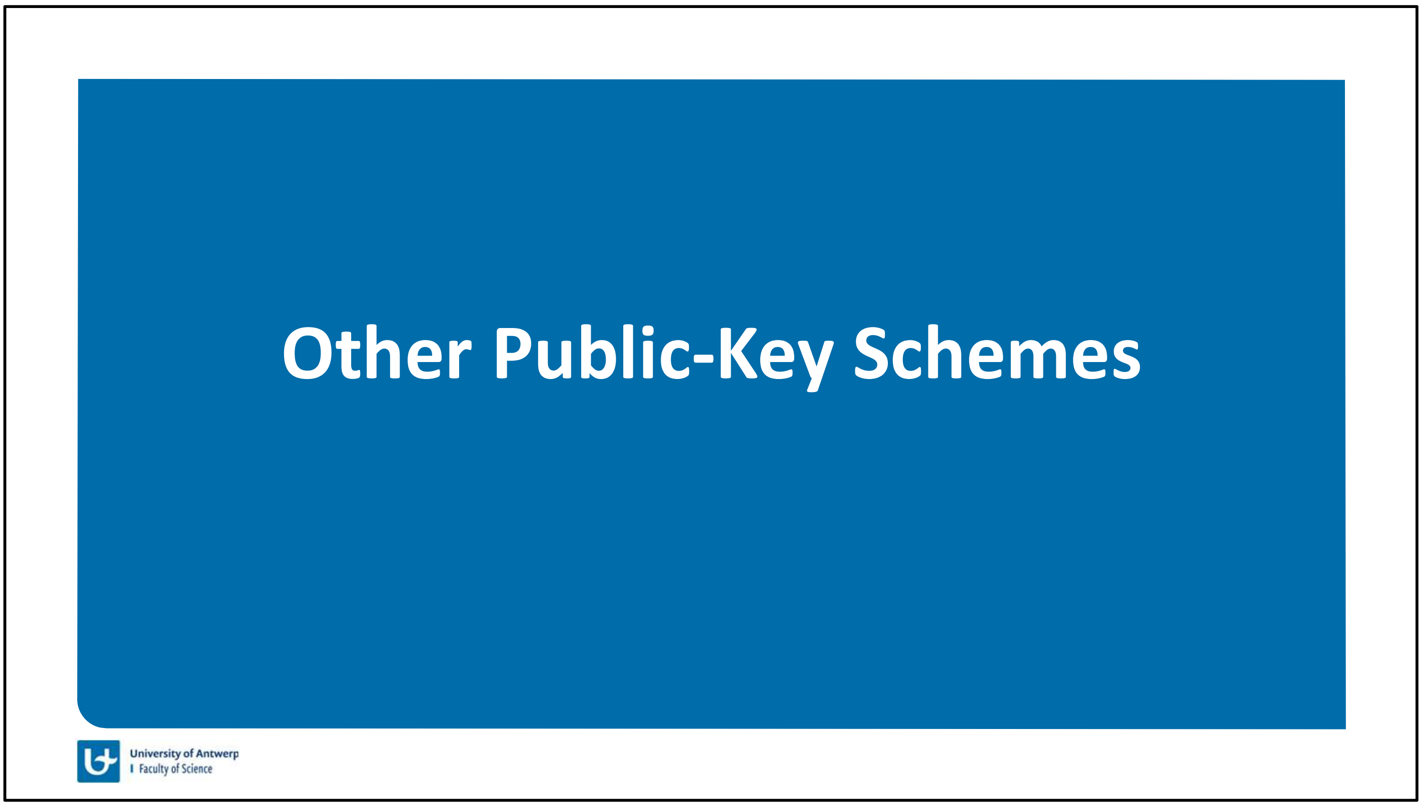
p, q:	two prime numbers; $n = p \times q$; $\phi(n) = (p-1)(q-1)$
public key $\{n, e\}$:	$\gcd(\phi(n), e) = 1$; $1 < e < \phi(n)$
private key $\{n, d\}$:	$d \equiv e^{-1} \pmod{\phi(n)}$
encryption:	$C = M^e \bmod n$
decryption:	$M = C^d \bmod n$

- **Efficient calculation** needed for modular exponentiation and key operations
- 4 main **attacks**: brute force, mathematical, timing and chosen ciphertext



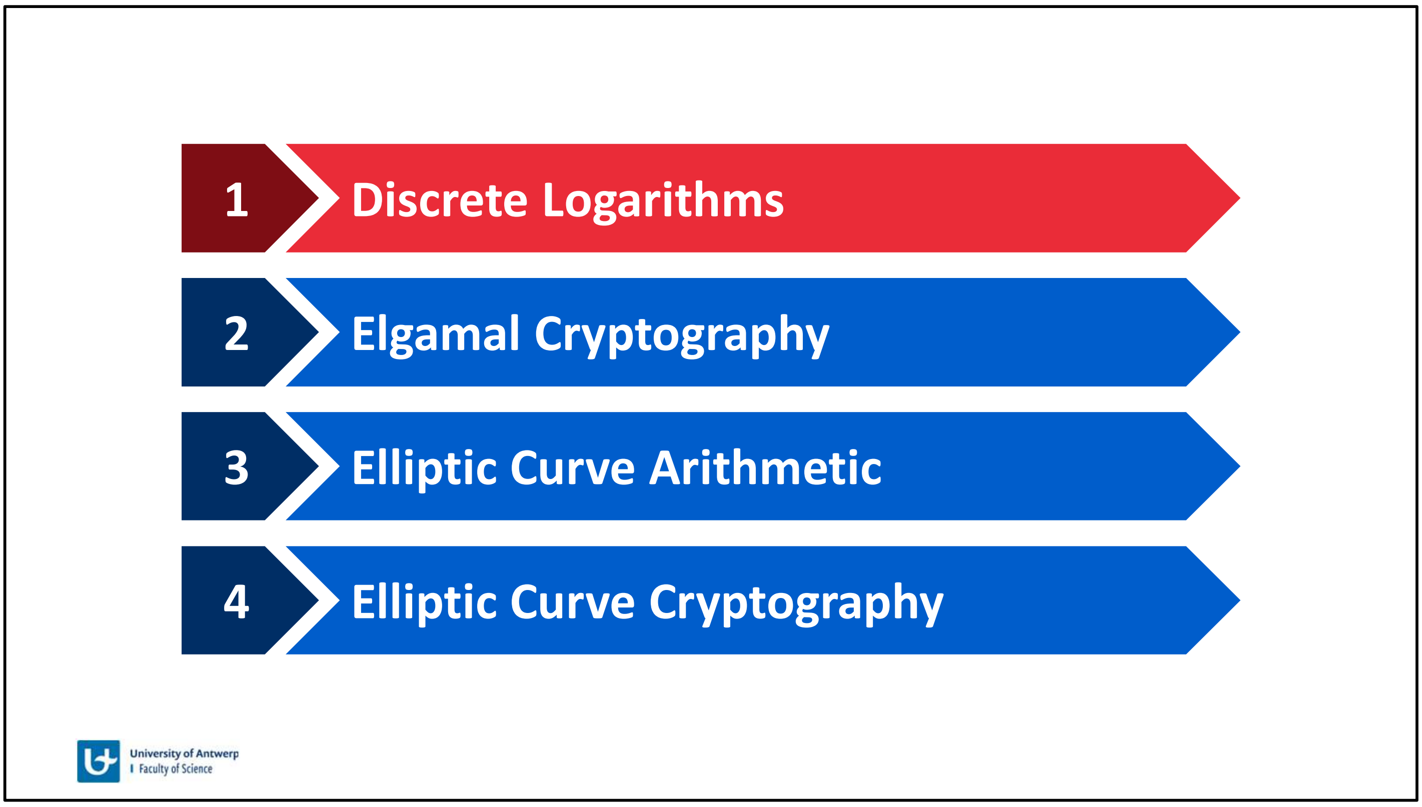
Link with the book
- [optional] Chapter 2 (Sections 2.1 – 2.7)
- Chapter 9 (Section 9.2)

Link with the videos
- Week 6 (Encryption from trapdoor permutations: RSA)
- Week 6 (Encryption from trapdoor permutations: attacks)



Other Public-Key Schemes

University of Antwerp
I Faculty of Science



University of Antwerp
Faculty of Science

Primitive root

Consider the general expression: $a^m \equiv 1 \pmod{n}$

- According to Euler's theorem, if a and n are relatively prime, then there is at least 1 integer m that satisfies this, namely $m = \phi(n)$

- The lowest positive exponent m for which this holds is to **order** of a (mod n)

- A number *a* is a **primitive root** of *n* if the order of a (mod n) is $\phi(n)$
  - Then $a, a^2, \ldots, a^{\phi(n)}$ are distinct (mod n) and all relatively prime to n
  - For a prime number p, if a is a primitive root of p, then $a, a^2, \ldots, a^{p-1}$ are all distinct (mod p).
  - Only $2, 4, p^\alpha, 2p^\alpha$(with p a prime, and α a positive integer) have primitive roots

University of Antwerp
Faculty of Science

Discrete logarithms for modular arithmetic

- For a primitive root *a* of a prime *p*, the powers of a from 1 to (p - 1) produce each integer from 1 through (p – 1) exactly once
- We also know that for any integer *b* an *r* exists, such that: $b \equiv r \pmod{p}$, with $0 \leq r \leq p-1$

The two points above, prove the existence of a unique exponent i:

$$b \equiv a^i \pmod{p} \quad \text{where } 0 \leq i \leq (p-1)$$

The exponent i is referred to as the **discrete logarithm** of the number *b* for the base *a (mod p)*, denoted as $\text{dlog}_{a,p}(b) = i$

- $\text{dlog}_{a,p}(1) = 0$
- $\text{dlog}_{a,p}(a) = 1$

- Used in **Diffie-Hellman** and **Elgamal** encryption

Example of discrete logarithms (1)

Example: p = 7

Step 1: Find primitive root(s) of 7

$a \in [1,6]$ is a primitive root of 7 if the smallest m for which $a^m \equiv 1 \pmod 7$ equals $\phi(7) = 6$:

- a = 1: smallest m = 1
- a = 2: smallest m = 3
- a = 3: smallest m = 6
- a = 4: smallest m = 3
- a = 5: smallest m = 6
- a = 6: smallest m = 2

Both 3 and 5 are primitive roots of 7

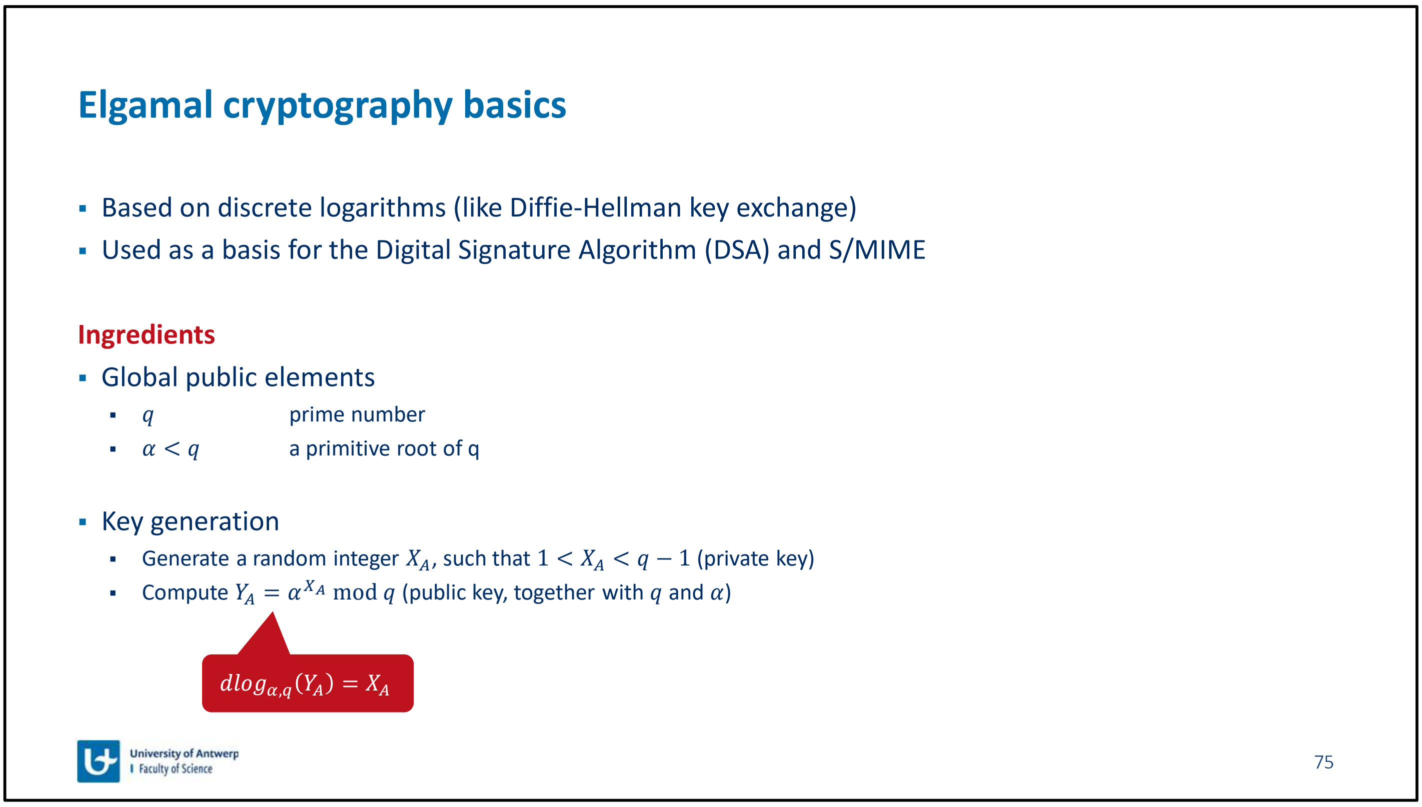
Example of discrete logarithms (2)

Example: p = 7

Step 2: Find discrete logarithms for a = 3 and p = 7

- $3^1 \; mod \; 7 = 3 \qquad \Rightarrow dlog_{3,7}(3) = 1$
- $3^2 \; mod \; 7 = 2 \qquad \Rightarrow dlog_{3,7}(2) = 2$
- $3^3 \; mod \; 7 = 6 \qquad \Rightarrow dlog_{3,7}(6) = 3$
- $3^4 \; mod \; 7 = 4 \qquad \Rightarrow dlog_{3,7}(4) = 4$
- $3^5 \; mod \; 7 = 5 \qquad \Rightarrow dlog_{3,7}(5) = 5$
- $3^6 \; mod \; 7 = 1 \qquad \Rightarrow dlog_{3,7}(1) = 6$

University of Antwerp
Faculty of Science



Elgamal cryptography basics

- Based on discrete logarithms (like Diffie-Hellman key exchange)
- Used as a basis for the Digital Signature Algorithm (DSA) and S/MIME

Ingredients

- Global public elements

q	prime number
$\alpha < q$	a primitive root of q

- Key generation
  - Generate a random integer X_A, such that $1 < X_A < q - 1$ (private key)
  - Compute $Y_A = \alpha^{X_A} \bmod q$ (public key, together with q and α)

$dlog_{\alpha,q}(Y_A) = X_A$

University of Antwerp | Faculty of Science

75

In 1984, T. Elgamal announced a public-key scheme based on discrete logarithms, closely related to the Diffie–Hellman technique. The Elgamal cryptosystem is used in some form in a number of standards including the digital signature standard (DSS) and the S/MIME email standard.

the global elements of Elgamal are a prime number q and α, which is a primitive root of q.
User A generates a private/public key pair as follows:
1. Generate a random integer X_A, such that $1 < X_A < q - 1$.
2. Compute $Y_A = \alpha^{XA} \bmod q$.
3. A's private key is X_A and A's public key is $\{q, \alpha, Y_A\}$.

The private key X_A is the discrete logarithm of Y_A base α modulo q.

Elgamal encryption and decryption

- **Encryption** (using the receiver's public key)
  - Message M is represented as an integer in the range $1 \leq M \leq q-1$
  - Choose a random integer k such that $1 \leq k \leq q-1$
  - Compute a one-time key $K = (Y_A)^k \bmod q$
  - Encrypt M as the pair of integers (C_1, C_2) with
$$C_1 = \alpha^k \bmod q, C_2 = (\mathrm{K} \times M) \bmod q$$

- **Decryption** (using the receiver's private key)
  - Recover the key by computing $K = (C_1)^{X_A} \bmod q$
  - Compute $M = (C_2 \times K^{-1}) \bmod q$

University of Antwerp
Faculty of Science

Proof that Elgamal encryption works

Step 1: Recovery of K

$K = (Y_A)^k \bmod q$	(definition of K during encryption)
$K = \left(\alpha^{X_A} \bmod q\right)^k \bmod q$	(substitution of Y_A according to definition)
$K = \alpha^{kX_A} \bmod q$	(rules of modular arithmetic)
$K = (C_1)^{X_A} \bmod q$	(substitution of C_1 according to definition)

Step 2: Recovery of the plaintext

$$\begin{aligned} C_2 &= (\mathrm{K} \times M) \bmod q \\ (C_2 K^{-1}) \bmod q &= (\mathrm{K} \times M \times K^{-1}) \bmod q \\ &= M \bmod q \\ &= M \end{aligned}$$

University of Antwerp
Faculty of Science

Exercise: Elgamal encryption

Exercise: Given q = 19, and α = 10, and private key X_A = 5. What is the encrypted message (C_1, C_2) of plaintext M = 17, and k = 6?

- Key generation
  - $Y_A = \alpha^{X_A} \bmod q$
- Encryption
  - Message M is represented as an integer in the range $1 \leq M \leq q-1$
  - Choose a random integer k such that $1 \leq k \leq q-1$
  - Compute a one-time key $K = (Y_A)^k \bmod q$
  - Encrypt M as the pair of integers (C_1, C_2) with

$$C_1 = \alpha^k \bmod q, C_2 = (\mathrm{K} \times M) \bmod q$$

University of Antwerp
I Faculty of Science

Solution: Elgamal encryption

Step 1: Calculate the public key component Y_A

$$Y_A = \alpha^{X_A} \bmod q = 10^5 \bmod 19 = 3$$

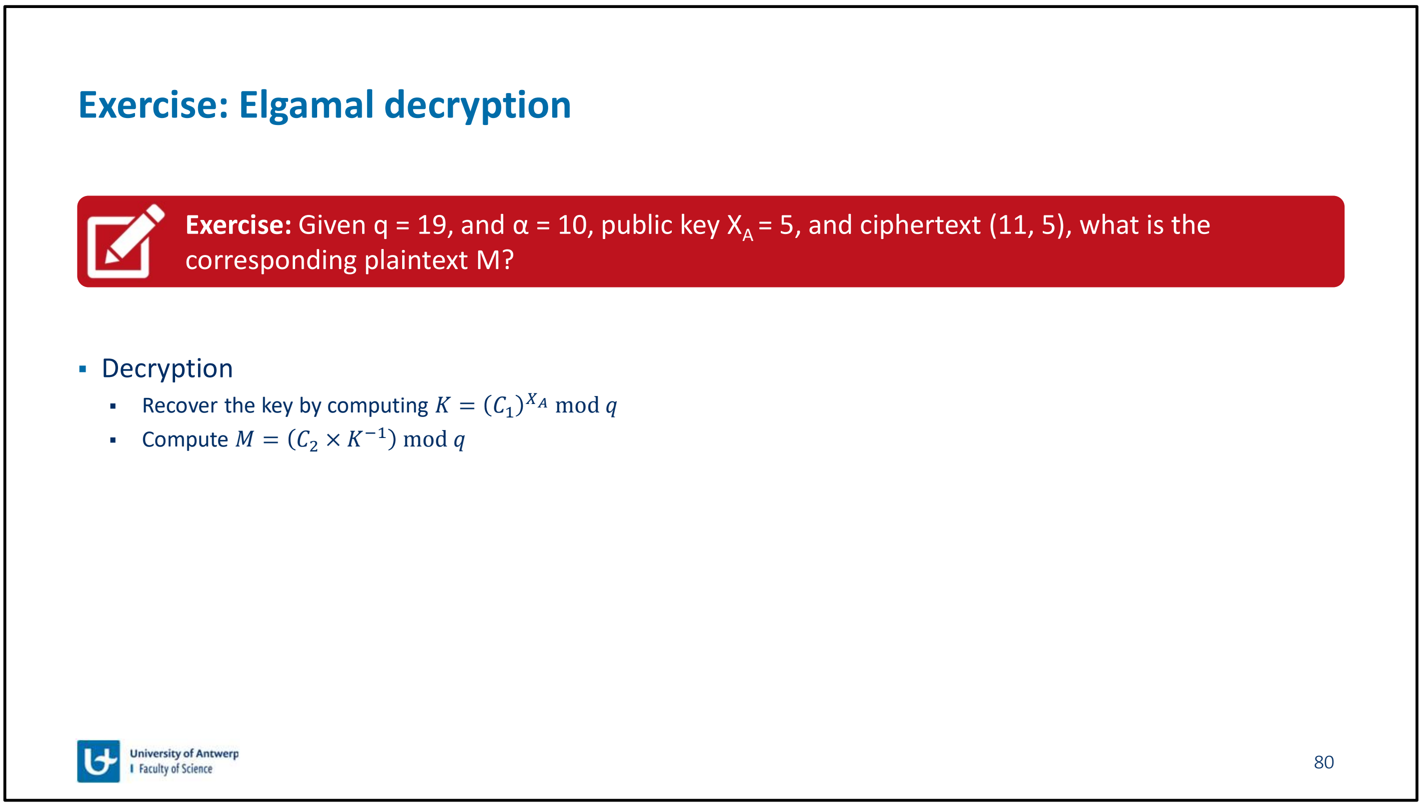
Step 2: Calculate one-time key K using public key [19, 10, 3] and k = 6

$$K = (Y_A)^k \bmod q = 3^6 \bmod 19 = 7$$

Step 3: Calculate the ciphertext (C_1, C_2) using public key [19, 10, 3] and K

$$C_1 = \alpha^k \bmod q = 10^6 \bmod 19 = 11$$
$$C_2 = (\mathrm{K} \times M) \bmod q = (7 \times 17) \bmod 19 = 5$$

Ciphertext (C_1, C_2) = (11, 5)



Exercise: Elgamal decryption

Exercise: Given q = 19, and α = 10, public key X_A = 5, and ciphertext (11, 5), what is the corresponding plaintext M?

- Decryption
  - Recover the key by computing $K = (C_1)^{X_A} \bmod q$
  - Compute $M = (C_2 \times K^{-1}) \bmod q$

University of Antwerp
I Faculty of Science

80

Solution: Elgamal decryption

Step 1: Calculate one-time key K using C_1

$$K = (C_1)^{X_A} \bmod q = 11^5 \bmod 19 = 7$$

Step 2: Calculate inverse of K = 7 in GF(19), i.e., Extended Euclidian Algorithm

$$r_{-1} = q = 19, r_0 = K = 7, w_{-1} = 0, w_0 = 1$$
$$r_1 = 5, q_1 = 2, w_1 = 17$$
$$r_2 = 2, q_2 = 1, w_2 = 3$$
$$r_3 = 1, q_3 = 2, w_3 = 11 = K^{-1}$$

Step 3: Calculate plaintext M using one-time key K and C_2

$$M = (C_2 \times K^{-1}) \bmod q = (5 \times 11) \bmod 19 = 17$$

Security of Elgamal encryption: key reuse

If a message consists of multiple blocks, a **new unique k should be used** for each block, as otherwise knowledge of block M_1 can be exploited.

Given

$$C_{1,1} = \alpha^k \bmod q, C_{2,1} = (\mathrm{K} \times M_1) \bmod q$$
$$C_{1,2} = \alpha^k \bmod q, C_{2,2} = (\mathrm{K} \times M_2) \bmod q$$

Then

$$\frac{C_{2,1}}{C_{2,2}} = \frac{(\mathrm{K}\times M_1) \bmod q}{(\mathrm{K}\times M_2) \bmod q} = \frac{M_1 \bmod q}{M_2 \bmod q}$$

If M_1 is known, then M_2 can be easily calculated as follows:

$$M_2 = \left(C_{2,1}\right)^{-1} C_{2,2} M_1 \bmod q$$

University of Antwerp
Faculty of Science

Security of Elgamal encryption: cryptoanalysis

The security of Elgamal is based on the difficulty of computing discrete logarithms:

- To recover the **private key** X_A the adversary needs to calculate
$$X_A = dlog_{\alpha,q}(Y_A)$$

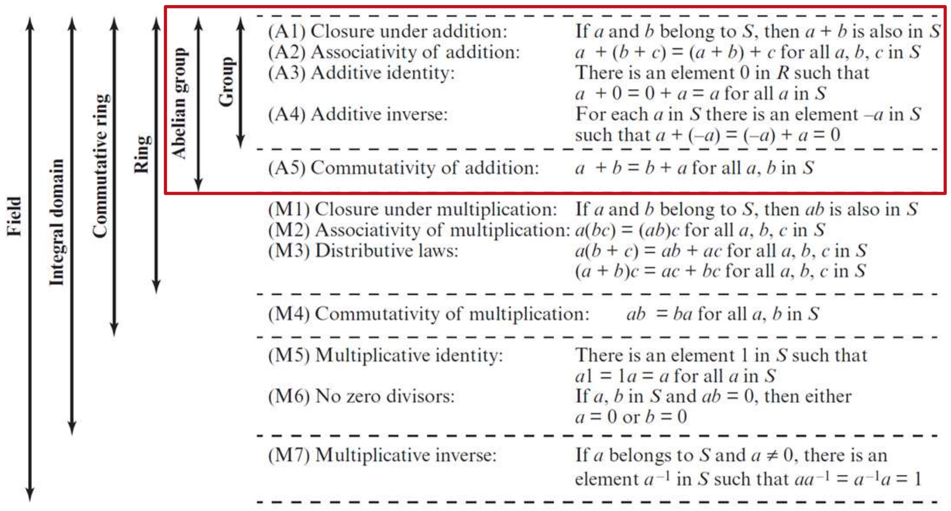
- To recover the **one-time key** K, the adversary needs to calculate
$$k = dlog_{\alpha,q}(C_1)$$

- Calculating discrete logarithms is computationally infeasible if:
  - q is at least 300 decimal digits long
  - q - 1 has at least one large prime factor

University of Antwerp
Faculty of Science

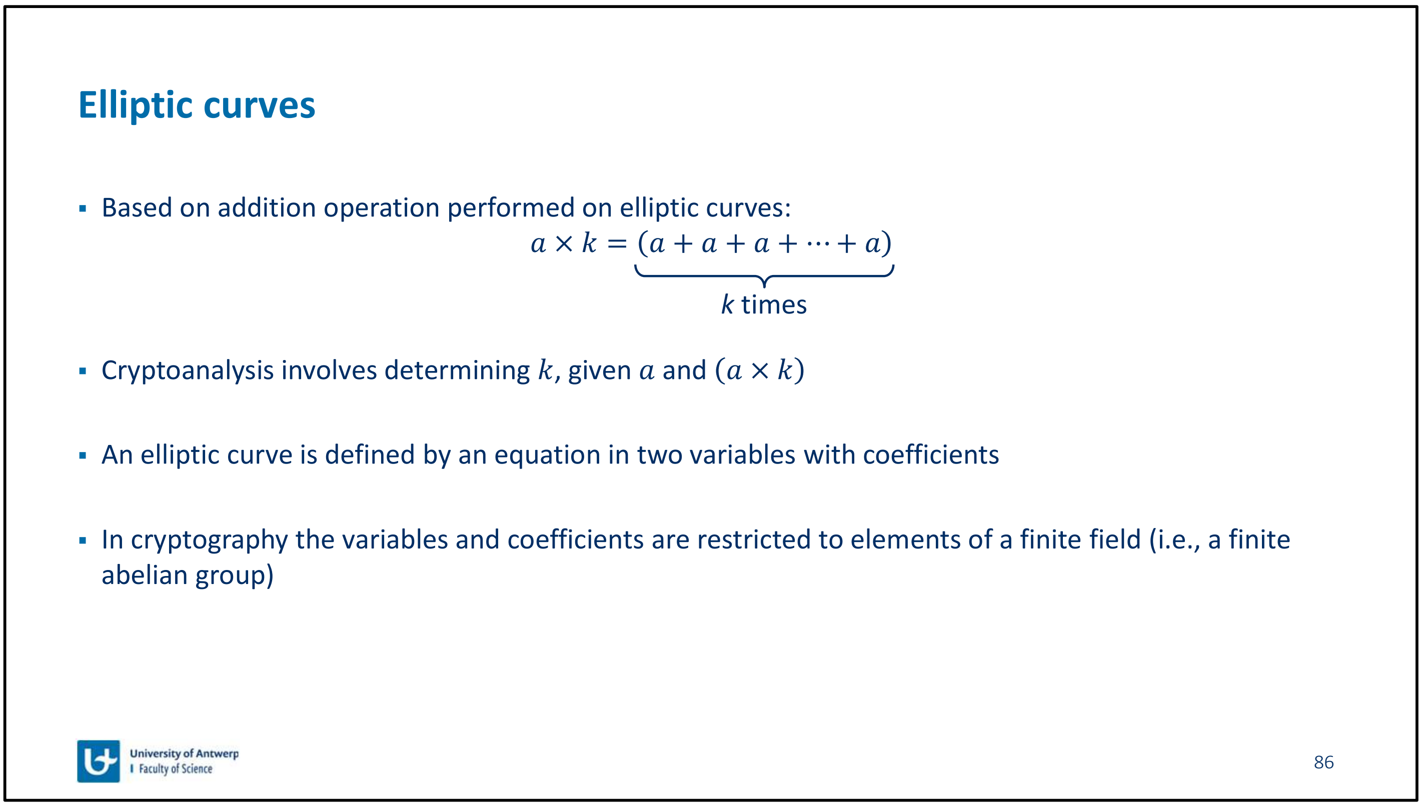
University of Antwerp
Faculty of Science

Albian groups are used as a basis for elliptic curves



(A1) Closure under addition:	If a and b belong to S, then $a + b$ is also in S
(A2) Associativity of addition:	$a + (b + c) = (a + b) + c$ for all a, b, c in S
(A3) Additive identity:	There is an element 0 in R such that $a + 0 = 0 + a = a$ for all a in S
(A4) Additive inverse:	For each a in S there is an element $-a$ in S such that $a + (-a) = (-a) + a = 0$
(A5) Commutativity of addition:	$a + b = b + a$ for all a, b in S
(M1) Closure under multiplication:	If a and b belong to S, then ab is also in S
(M2) Associativity of multiplication:	$a(bc) = (ab)c$ for all a, b, c in S
(M3) Distributive laws:	$a(b + c) = ab + ac$ for all a, b, c in S $(a + b)c = ac + bc$ for all a, b, c in S
(M4) Commutativity of multiplication:	$ab = ba$ for all a, b in S
(M5) Multiplicative identity:	There is an element 1 in S such that $a1 = 1a = a$ for all a in S
(M6) No zero divisors:	If a, b in S and $ab = 0$, then either $a = 0$ or $b = 0$
(M7) Multiplicative inverse:	If a belongs to S and $a \neq 0$, there is an element a^{-1} in S such that $aa^{-1} = a^{-1}a = 1$

University of Antwerp
Faculty of Science



Elliptic curves

- Based on addition operation performed on elliptic curves:

$$a \times k = \underbrace{(a + a + a + \cdots + a)}_{k \text{ times}}$$

- Cryptoanalysis involves determining k, given a and $(a \times k)$
- An elliptic curve is defined by an equation in two variables with coefficients
- In cryptography the variables and coefficients are restricted to elements of a finite field (i.e., a finite abelian group)

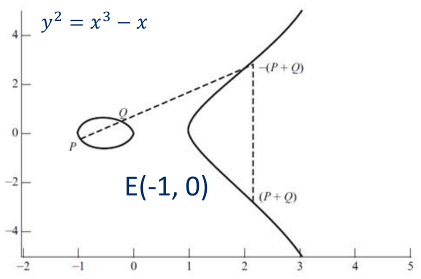
University of Antwerp
I Faculty of Science

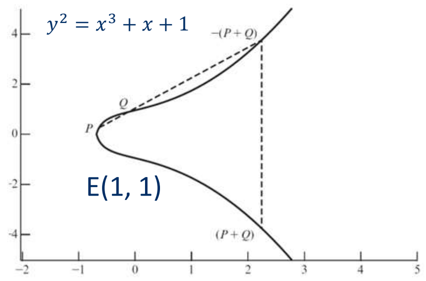
86

For elliptic curve cryptography, an operation over elliptic curves, called addition, is used. Multiplication is defined by repeated addition. An elliptic curve is defined by an equation in two variables with coefficients. For cryptography, the variables and coefficients are restricted to elements in a finite field, which results in the definition of a finite abelian group. Before looking at this, we first look at elliptic curves in which the variables and coefficients are real numbers. This case is perhaps easier to visualize.

Elliptic curves over real numbers

- Elliptic curves are described using cubic Weierstrass equations:
$$y^2 + axy + by = x^3 + cx^2 + dx + e$$
- For cryptographic purposes, a simplified equation is used
$$y^2 = x^3 + ax + b$$
$$y = \sqrt{x^3 + ax + b}$$
- All the points (x, y) that satisfy the equation for a specific (a, b), are E(a, b)





Elliptic curves are not ellipses. They are so named because they are described by cubic equations, similar to those used for calculating the circumference of an ellipse. In general, cubic equations for elliptic curves take the following form, known as a Weierstrass equation:

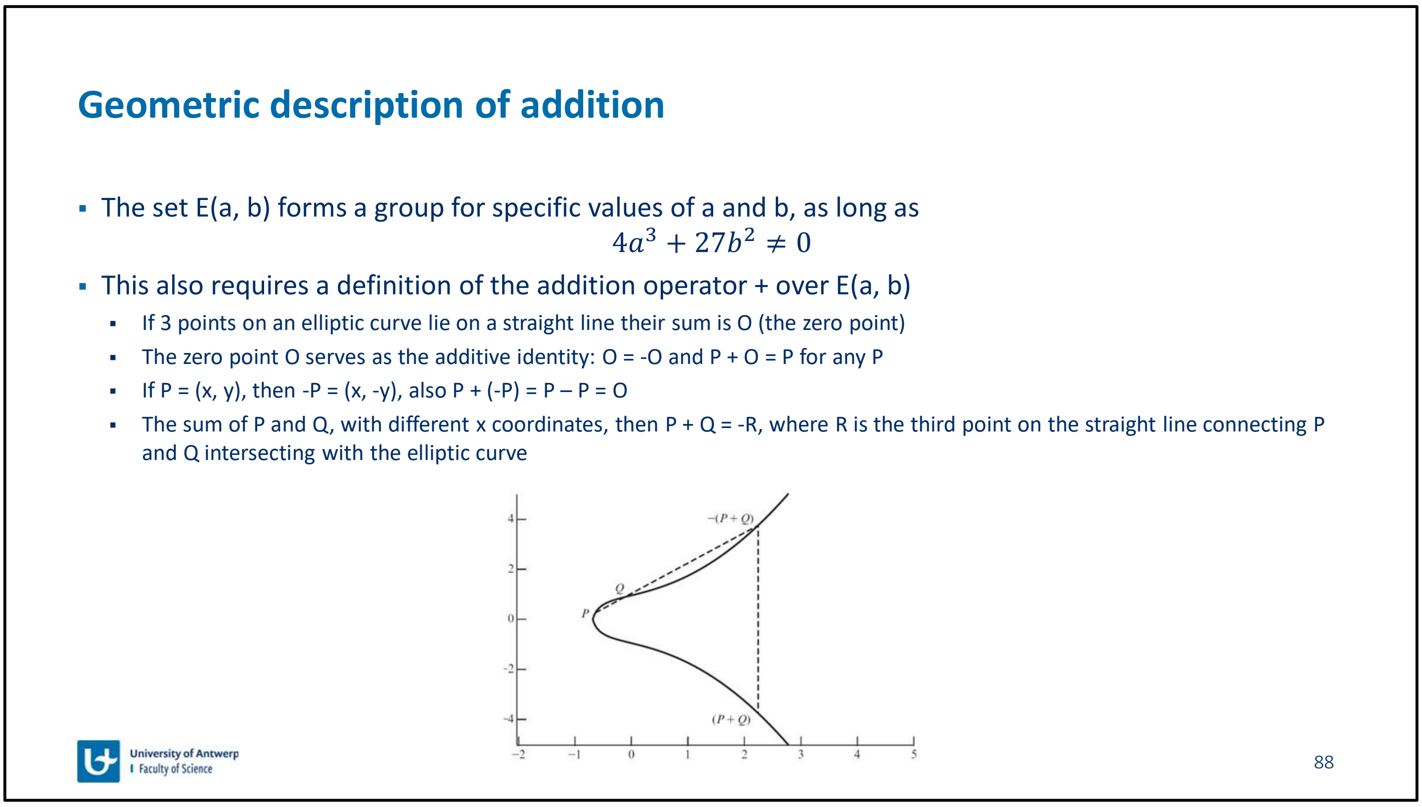
$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

where *a, b, c, d, e* are real numbers and *x* and *y* take on values in the real numbers. For our purpose, it is sufficient to limit ourselves to equations of the form

$$y^2 = x^3 + ax + b$$

Also included in the definition of an elliptic curve is a single element denoted *O* and called the *point at infinity* or the *zero point.* For given values of *a* and *b*, the plot consists of positive and negative values of *y* for each value of *x*. Thus, each curve is symmetric about *y* = 0. The figures show two examples of elliptic curves. As you can see, the formula sometimes produces weird-looking curves.

Now, consider the set of points E(*a, b*) consisting of all of the points (*x, y*) that satisfy the equation together with the element *O*. Using a different value of the pair (*a, b*) results in a different set E(*a, b*). Using this terminology, the two curves depict the sets E(-1, 0) and E(1, 1), respectively.

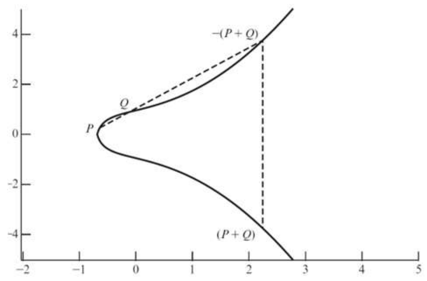


Geometric description of addition

- The set E(a, b) forms a group for specific values of a and b, as long as

$$4a^3 + 27b^2 \neq 0$$

- This also requires a definition of the addition operator + over E(a, b)
  - If 3 points on an elliptic curve lie on a straight line their sum is O (the zero point)
  - The zero point O serves as the additive identity: O = -O and P + O = P for any P
  - If P = (x, y), then -P = (x, -y), also P + (-P) = P – P = O
  - The sum of P and Q, with different x coordinates, then P + Q = -R, where R is the third point on the straight line connecting P and Q intersecting with the elliptic curve



University of Antwerp
Faculty of Science

88

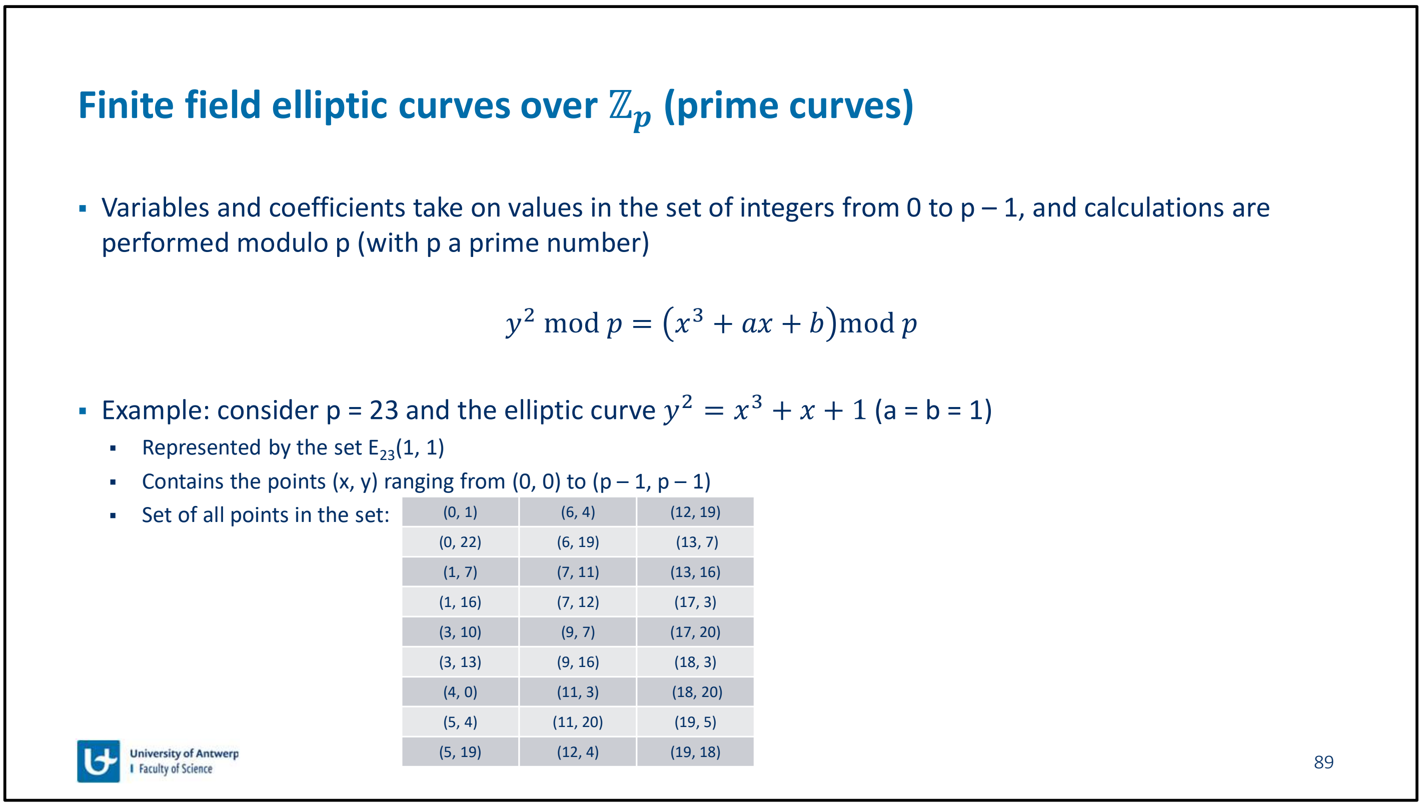
It can be shown that a group can be defined based on the set E(*a, b*) for specific values of *a* and *b*, provided
the following condition is met:

$$4a^3 + 27b^2 \neq 0$$

To define the group, we must define an operation, called addition and denoted by +, for the set E(*a, b*), where *a* and *b* satisfy this equation. In geometric terms, the rules for addition can be stated as follows: If three points on an elliptic curve lie on a straight line, their sum is *O*. From this definition, we can define the rules of addition over an elliptic curve:

1. *O* serves as the additive identity. Thus *O* = -*O*; for any point *P* on the elliptic curve, *P* + *O* = *P*. In what follows, we assume *P* ≠ *O* and *Q* ≠ *O*.
2. The negative of a point *P* is the point with the same *x* coordinate but the negative of the *y* coordinate; that is, if *P* = (*x*, *y*), then -*P* = (*x*, -*y*). Note that these two points can be joined by a vertical line. Note that *P* + (-*P*) = *P* - *P* = *O*.
3. To add two points *P* and *Q* with different *x* coordinates, draw a straight line between them and find the third point of intersection *R*. It is easily seen that there is a unique point *R* that is the point of intersection (unless the line is tangent to the curve at either *P* or *Q*, in which case we take *R* = *P* or *R* = *Q*, respectively). To form a group structure, we need to define addition on these three points: *P* + *Q* = -*R*. That is, we define *P* + *Q* to be the mirror image (with respect to the *x* axis) of the third point of intersection. The figure illustrates this construction.
4. The geometric interpretation of the preceding item also applies to two points, *P* and -*P*, with the same *x* coordinate. The points are joined by a vertical line, which can be viewed as also intersecting the curve at the infinity point. We therefore have *P* + (-*P*) = *O*, which is consistent with item (2).

5. To double a point Q, draw the tangent line and find the other point of intersection S. Then $Q + Q = 2Q = -S$.



Finite field elliptic curves over $\mathbb{Z}_p$ (prime curves)

- Variables and coefficients take on values in the set of integers from 0 to p – 1, and calculations are performed modulo p (with p a prime number)

$$y^2 \bmod p = \left(x^3 + ax + b\right) \bmod p$$

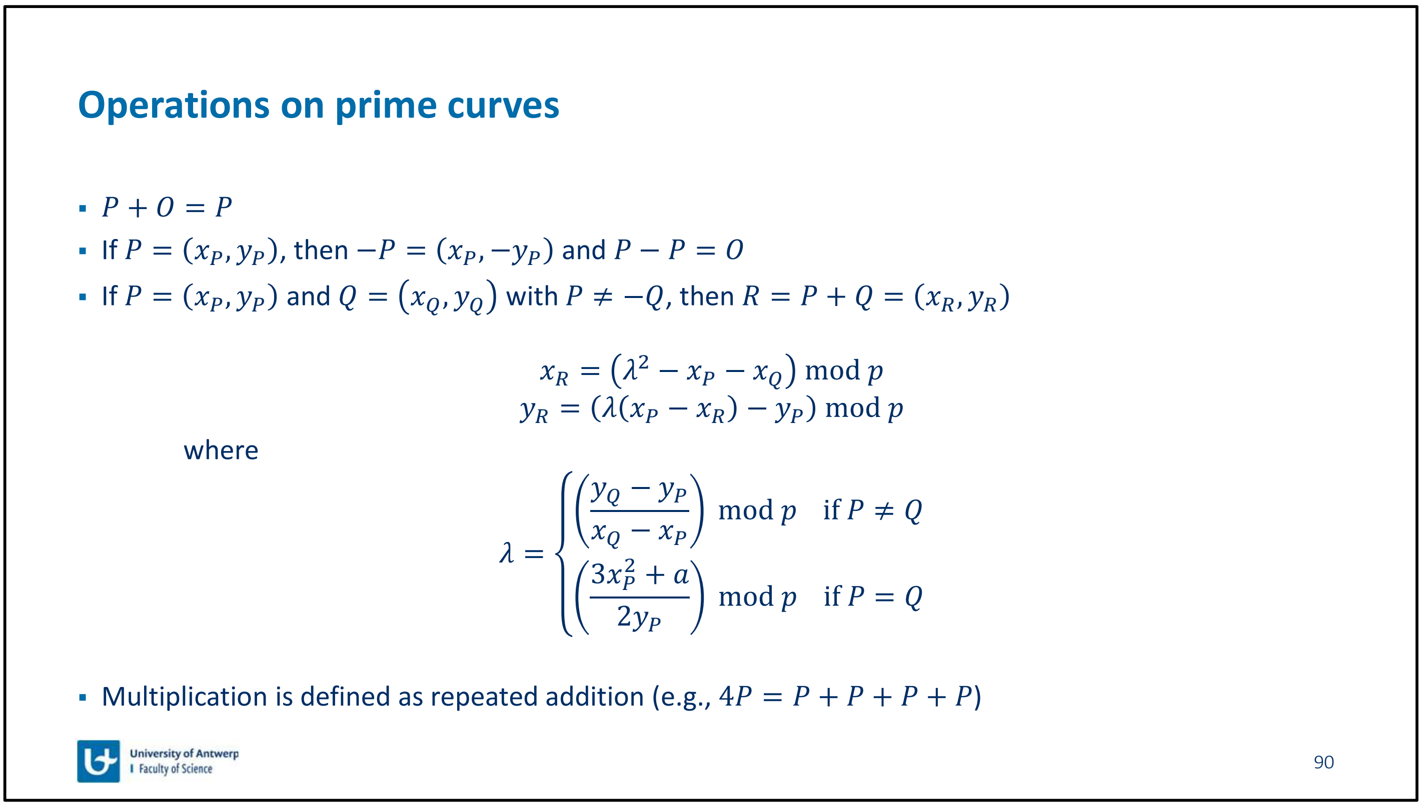
- Example: consider p = 23 and the elliptic curve $y^2 = x^3 + x + 1$ (a = b = 1)
  - Represented by the set $E_{23}(1, 1)$
  - Contains the points (x, y) ranging from (0, 0) to (p – 1, p – 1)
  - Set of all points in the set:

(0, 1)	(6, 4)	(12, 19)
(0, 22)	(6, 19)	(13, 7)
(1, 7)	(7, 11)	(13, 16)
(1, 16)	(7, 12)	(17, 3)
(3, 10)	(9, 7)	(17, 20)
(3, 13)	(9, 16)	(18, 3)
(4, 0)	(11, 3)	(18, 20)
(5, 4)	(11, 20)	(19, 5)
(5, 19)	(12, 4)	(19, 18)

University of Antwerp
Faculty of Science

89

Elliptic curve cryptography makes use of elliptic curves in which the variables and coefficients are all restricted to elements of a finite field. Two families of elliptic curves are used in cryptographic applications: prime curves over Z_p and binary curves over GF(2^m). For a prime curve over Z_p, we use a cubic equation in which the variables and coefficients all take on values in the set of integers from 0 through *p* - 1 and in which calculations are performed modulo *p*. For a binary curve defined over GF(2^m), the variables and coefficients all take on values in GF(2^m) and in calculations are performed over GF(2^m). Prime curves are best for software applications, because the extended bit-fiddling operations needed by binary curves are not required; and that binary curves are best for hardware applications, where it takes remarkably few logic gates to create a powerful, fast cryptosystem. As an example, we consider prime curves.



Operations on prime curves

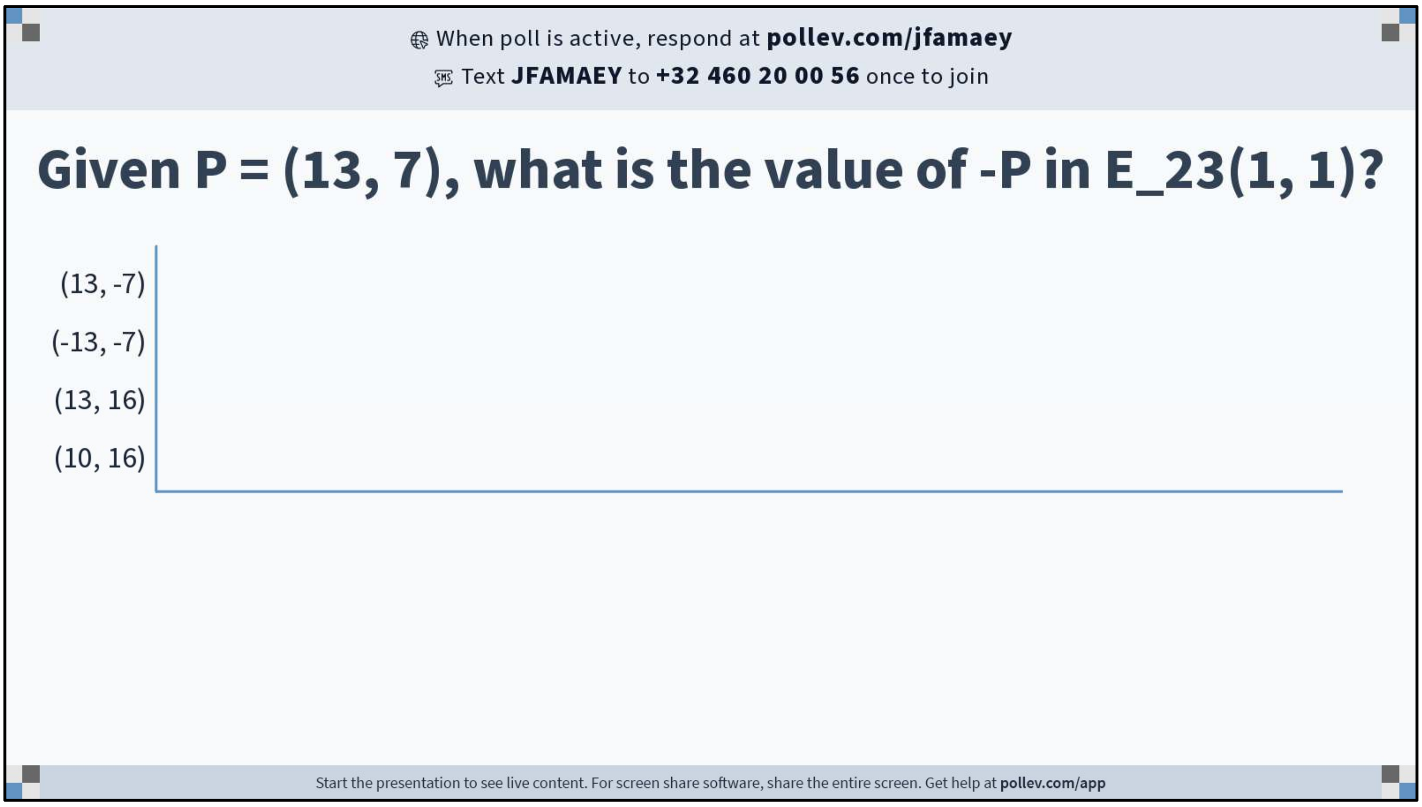
- $P + O = P$
- If $P = (x_P, y_P)$, then $-P = (x_P, -y_P)$ and $P - P = O$
- If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $P \neq -Q$, then $R = P + Q = (x_R, y_R)$

$$x_R = (\lambda^2 - x_P - x_Q) \bmod p$$
$$y_R = (\lambda(x_P - x_R) - y_P) \bmod p$$

where

$$\lambda = \begin{cases} \left(\dfrac{y_Q - y_P}{x_Q - x_P}\right) \bmod p & \text{if } P \neq Q \\ \left(\dfrac{3x_P^2 + a}{2y_P}\right) \bmod p & \text{if } P = Q \end{cases}$$

- Multiplication is defined as repeated addition (e.g., $4P = P + P + P + P$)



When poll is active, respond at **pollev.com/jfamaey**

Text **JFAMAEY** to **+32 460 20 00 56** once to join

Given P = (13, 7), what is the value of -P in E_23(1, 1)?

(13, -7)

(-13, -7)

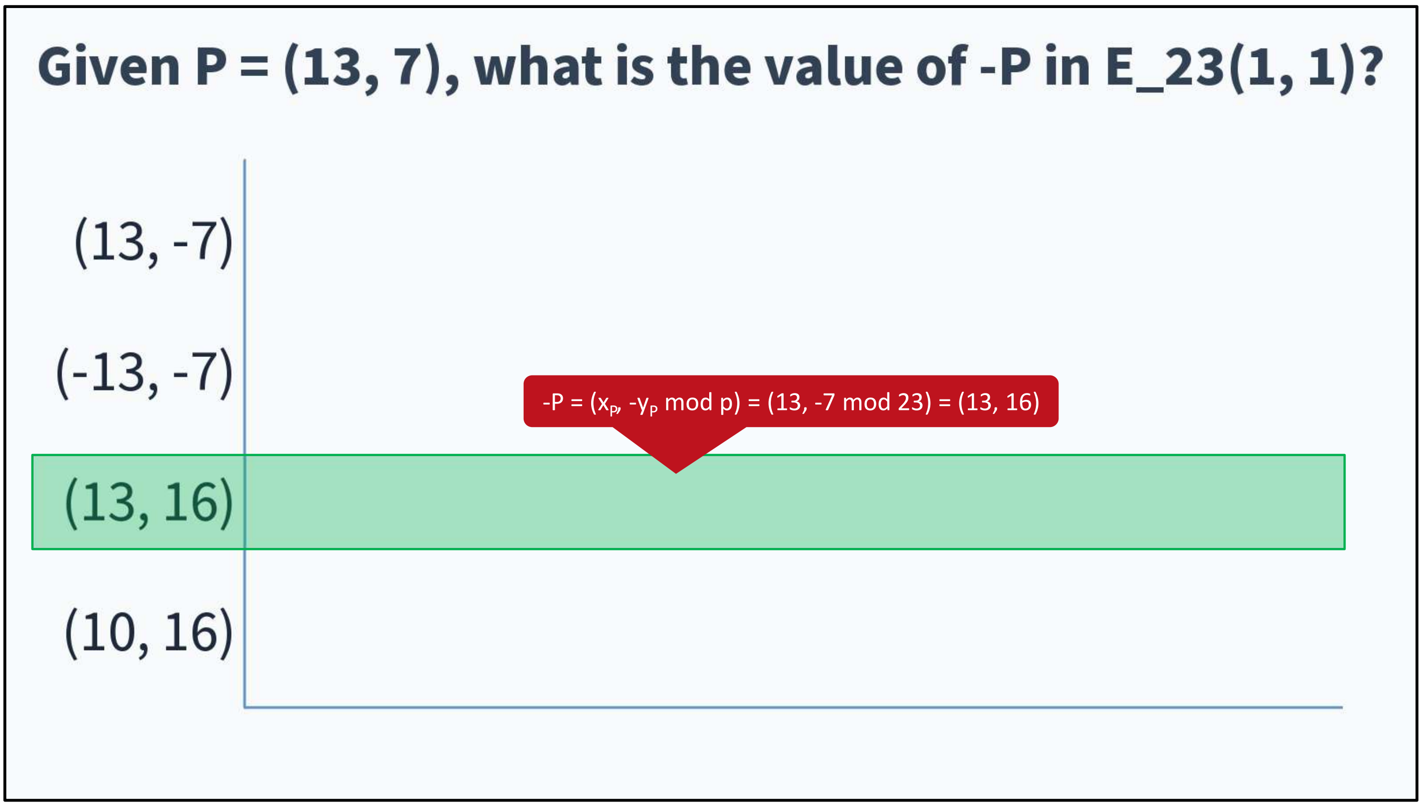
(13, 16)

(10, 16)

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

Given P = (13, 7), what is the value of -P in E_23(1, 1)?
https://www.polleverywhere.com/multiple_choice_polls/esruv72ktwsDDU4zrgk8Y?state=opened&flow=Default&onscreen=persist



Given P = (13, 7), what is the value of -P in E_23(1, 1)?

(13, -7)

(-13, -7)

(13, 16)

(10, 16)

-P = (x_P, -y_P mod p) = (13, -7 mod 23) = (13, 16)

Poll Title: Given P = (13, 7), what is the value of -P in E_23(1, 1)?

Exercise: Calculating the sum in elliptic curves over $\mathbb{Z}_p$

Exercise: What is the sum of $P = (3, 10)$ and $Q = (9, 7)$ over the elliptic curve $E_{23}(1,1)$?

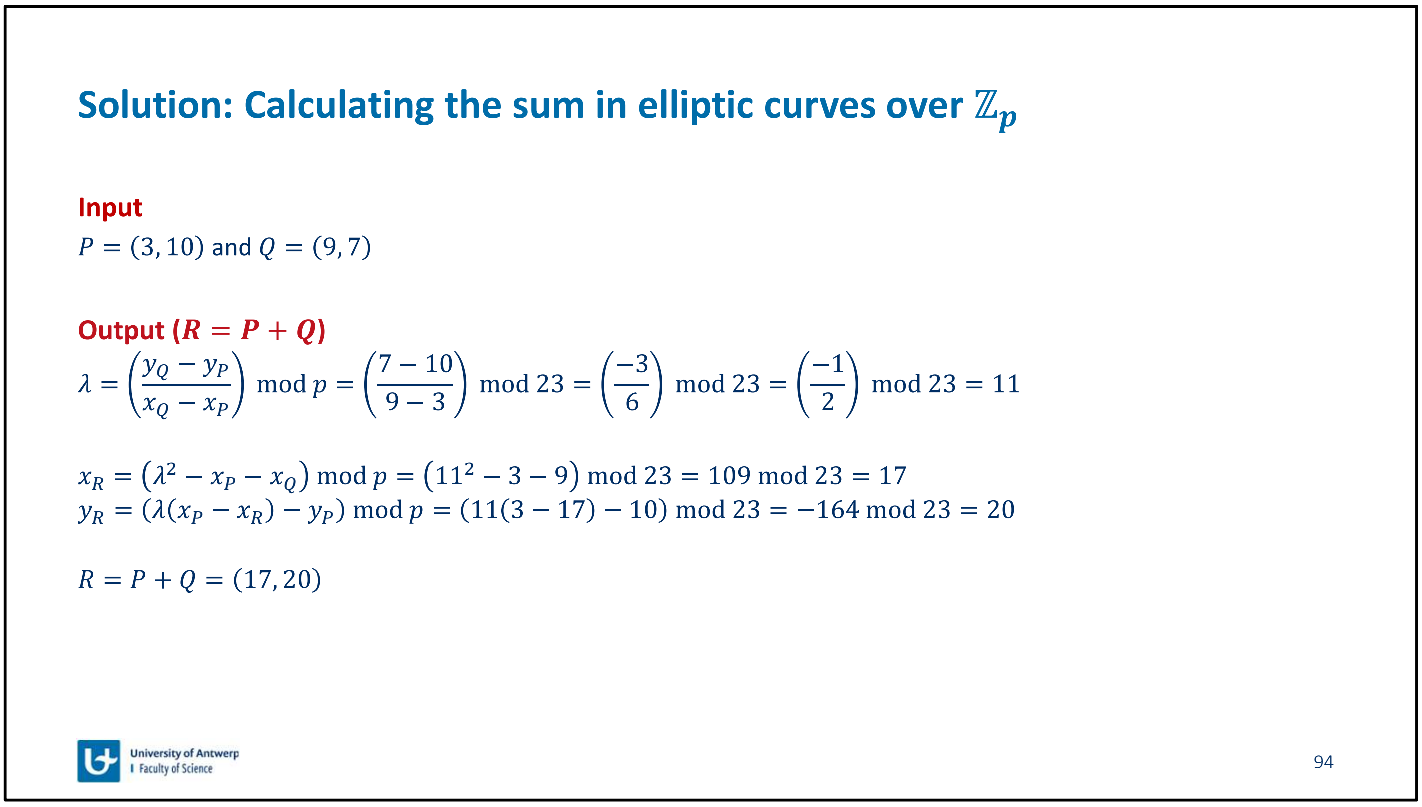
If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $P \neq -Q$, then $R = P + Q = (x_R, y_R)$

$$x_R = (\lambda^2 - x_P - x_Q) \bmod p$$
$$y_R = (\lambda(x_P - x_R) - y_P) \bmod p$$

where

$$\lambda = \begin{cases} \left(\dfrac{y_Q - y_P}{x_Q - x_P}\right) \bmod p & \text{if } P \neq Q \\ \left(\dfrac{3x_P^2 + a}{2y_P}\right) \bmod p & \text{if } P = Q \end{cases}$$

University of Antwerp
Faculty of Science



Solution: Calculating the sum in elliptic curves over $\mathbb{Z}_p$

Input

$P = (3, 10)$ and $Q = (9, 7)$

Output ($R = P + Q$)

$$\lambda = \left(\frac{y_Q - y_P}{x_Q - x_P}\right) \bmod p = \left(\frac{7 - 10}{9 - 3}\right) \bmod 23 = \left(\frac{-3}{6}\right) \bmod 23 = \left(\frac{-1}{2}\right) \bmod 23 = 11$$

$$x_R = \left(\lambda^2 - x_P - x_Q\right) \bmod p = \left(11^2 - 3 - 9\right) \bmod 23 = 109 \bmod 23 = 17$$
$$y_R = \left(\lambda(x_P - x_R) - y_P\right) \bmod p = \left(11(3 - 17) - 10\right) \bmod 23 = -164 \bmod 23 = 20$$

$R = P + Q = (17, 20)$

University of Antwerp
Faculty of Science

94

Calculating the modulo of a division requires calculating the multiplicative inverse of the divisor: $\frac{a}{b} \bmod p = (a \bmod p) \times (b^{-1} \bmod p) \bmod p$

Exercise: Multiplication in elliptic curves over $\mathbb{Z}_p$

Exercise: Given $P = (3, 10)$, what is $2P$ over the elliptic curve $E_{23}(1,1)$?

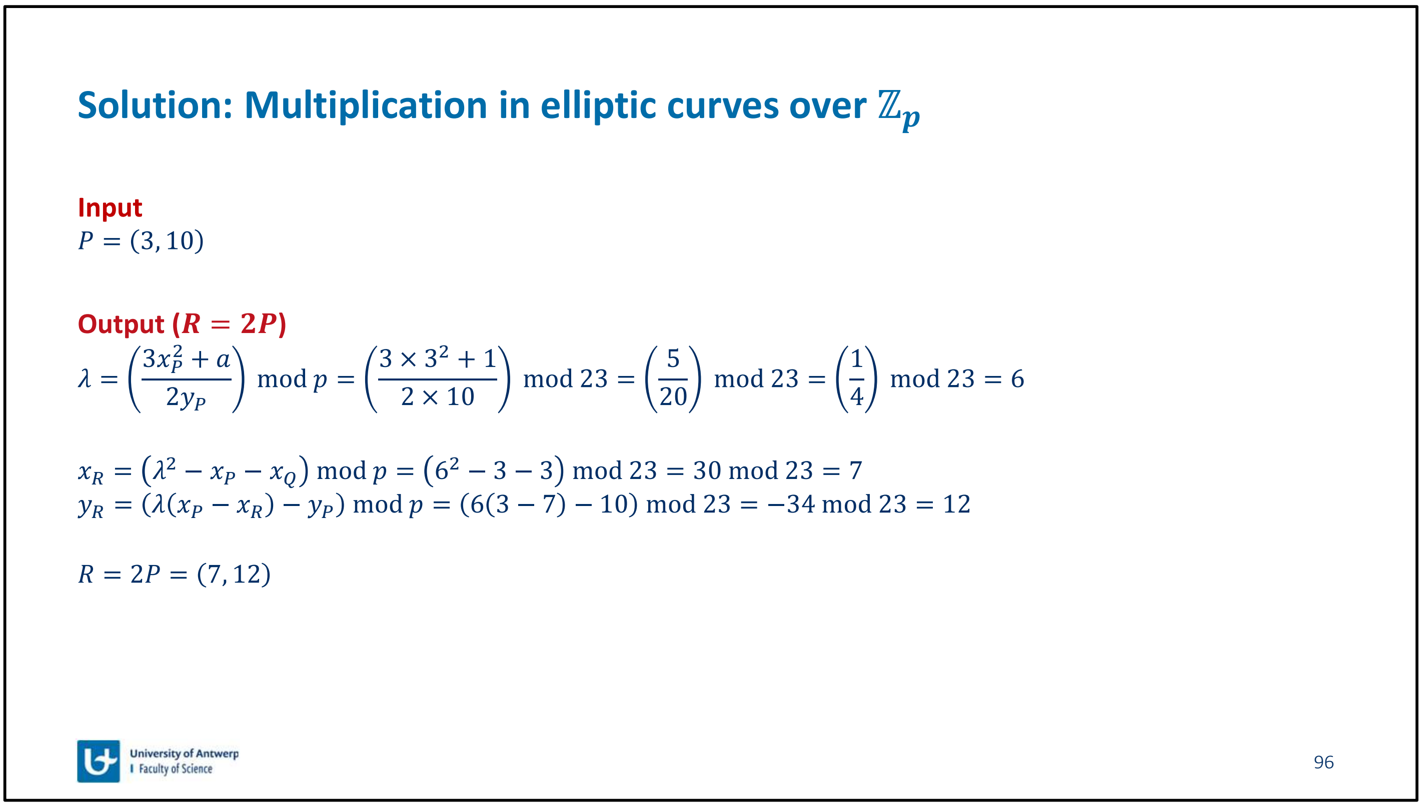
If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $P \neq -Q$, then $R = P + Q = (x_R, y_R)$

$$x_R = (\lambda^2 - x_P - x_Q) \bmod p$$
$$y_R = (\lambda(x_P - x_R) - y_P) \bmod p$$

where

$$\lambda = \begin{cases} \left(\dfrac{y_Q - y_P}{x_Q - x_P}\right) \bmod p & \text{if } P \neq Q \\ \left(\dfrac{3x_P^2 + a}{2y_P}\right) \bmod p & \text{if } P = Q \end{cases}$$

University of Antwerp | Faculty of Science



Solution: Multiplication in elliptic curves over $\mathbb{Z}_p$

Input
$P = (3, 10)$

Output ($R = 2P$)
$\lambda = \left(\frac{3x_P^2 + a}{2y_P}\right) \bmod p = \left(\frac{3 \times 3^2 + 1}{2 \times 10}\right) \bmod 23 = \left(\frac{5}{20}\right) \bmod 23 = \left(\frac{1}{4}\right) \bmod 23 = 6$

$x_R = \left(\lambda^2 - x_P - x_Q\right) \bmod p = \left(6^2 - 3 - 3\right) \bmod 23 = 30 \bmod 23 = 7$
$y_R = (\lambda(x_P - x_R) - y_P) \bmod p = (6(3 - 7) - 10) \bmod 23 = -34 \bmod 23 = 12$

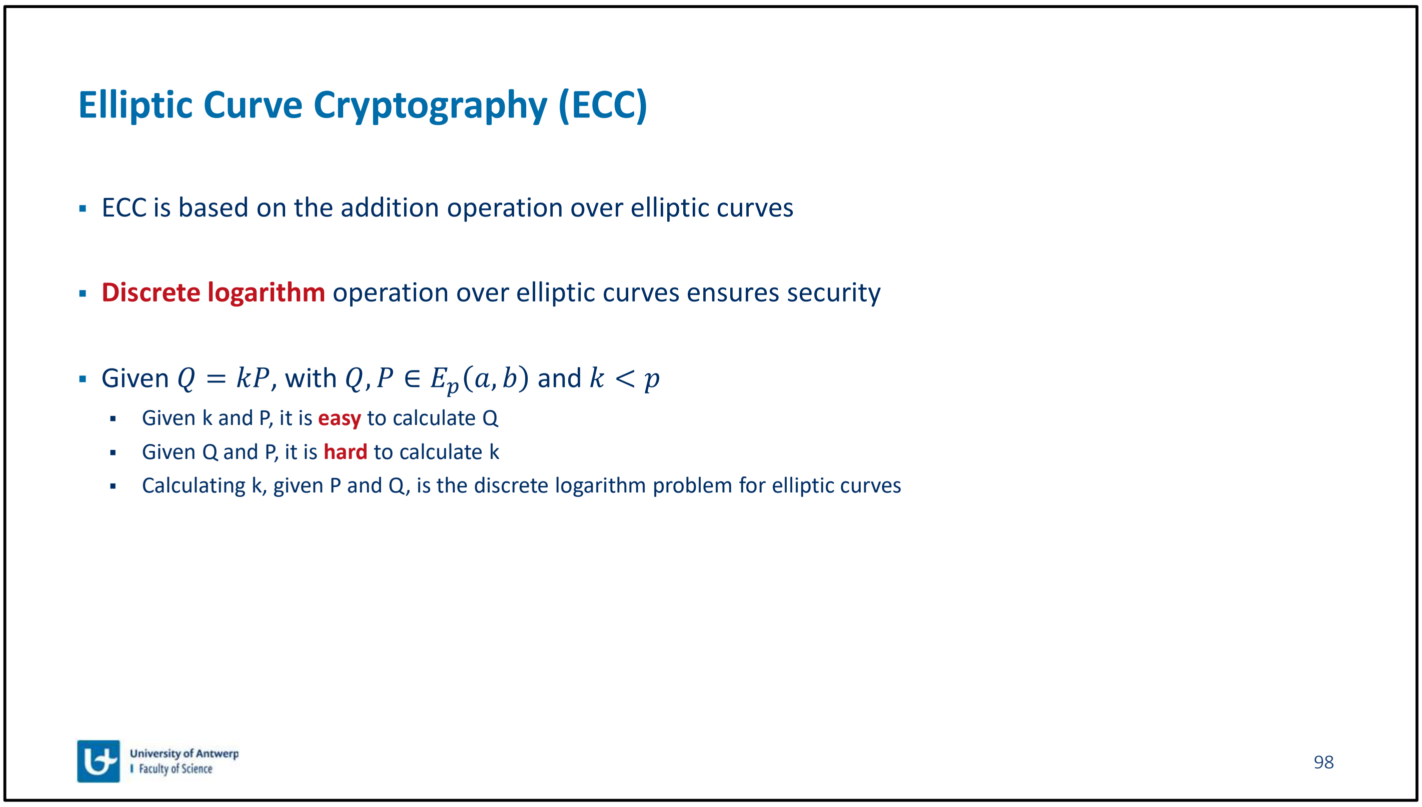
$R = 2P = (7, 12)$

University of Antwerp | Faculty of Science

96

Calculating the modulo of a division requires calculating the multiplicative inverse of the divisor: $\frac{a}{b} \bmod p = (a \bmod p) \times (b^{-1} \bmod p) \bmod p$

University of Antwerp
Faculty of Science


Elliptic Curve Cryptography (ECC)

- ECC is based on the addition operation over elliptic curves
- **Discrete logarithm** operation over elliptic curves ensures security
- Given $Q = kP$, with $Q, P \in E_p(a, b)$ and $k < p$
  - Given k and P, it is **easy** to calculate Q
  - Given Q and P, it is **hard** to calculate k
  - Calculating k, given P and Q, is the discrete logarithm problem for elliptic curves

University of Antwerp
Faculty of Science

98

Most of the products and standards that use public-key cryptography for encryption and digital signatures use RSA. As we have seen, the key length for secure RSA use has increased over recent years, and this has put a heavier processing load on applications using RSA. This burden has ramifications, especially for electronic commerce sites that conduct large numbers of secure transactions. A competing system challenges RSA: elliptic curve cryptography (ECC). ECC is showing up in standardization efforts, including the IEEE P1363 Standard for Public-Key Cryptography.

The addition operation in ECC is the counterpart of modular multiplication in RSA, and multiple addition is the counterpart of modular exponentiation. To form a cryptographic system using elliptic curves, we need to find a “hard problem” corresponding to factoring the product of two primes or taking the discrete logarithm.

Consider the equation $Q = kP$ where $Q, P \in E_p(a, b)$ and $k < p$. It is relatively easy to calculate Q given k and P, but it is hard to determine k given Q and P. This is called the discrete logarithm problem for elliptic curves.

Exercise: Find k using brute-force

Exercise: Given $E_{23}(9, 17)$, what is the discrete logarithm k of $Q = (4, 5)$ to the base $P = (16, 5)$?

$y^2 \bmod p = \left(x^3 + ax + b\right) \bmod p$

If $P = (x_P, y_P)$ and $Q = \left(x_Q, y_Q\right)$ with $P \neq -Q$, then $R = P + Q = (x_R, y_R)$

$$x_R = \left(\lambda^2 - x_P - x_Q\right) \bmod p$$
$$y_R = (\lambda(x_P - x_R) - y_P) \bmod p$$

where

$$\lambda = \begin{cases} \left(\dfrac{y_Q - y_P}{x_Q - x_P}\right) \bmod p & \text{if } P \neq Q \\ \left(\dfrac{3x_P^2 + a}{2y_P}\right) \bmod p & \text{if } P = Q \end{cases}$$

University of Antwerp
Faculty of Science

Solution: Find k using brute-force (1)

Solution in brief: Calculate 2P, 3P, 4P, ... until it equals Q

Step 1: Calculating 2P

$$\lambda = \left(\frac{3x_P^2 + a}{2y_P}\right) \bmod p = \left(\frac{3 \times 16^2 + 9}{2 \times 5}\right) \bmod 23 = \left(\frac{777}{10}\right) \bmod 23$$
$$= 18 \times (10^{-1} \bmod 23) = 18 \times 7 \bmod 23 = 11$$

$$x_{2P} = \left(\lambda^2 - x_P - x_Q\right) \bmod p = \left(11^2 - 16 - 16\right) \bmod 23 = 89 \bmod 23 = 20$$
$$y_{2P} = (\lambda(x_P - x_R) - y_P) \bmod p = (11(16 - 20) - 5) \bmod 23$$
$$= -49 \bmod 23 = 20$$

$2P = (20, 20)$

Solution: Find k using brute-force (2)

Step 2: Calculating $\boldsymbol{3P = P + 2P}$

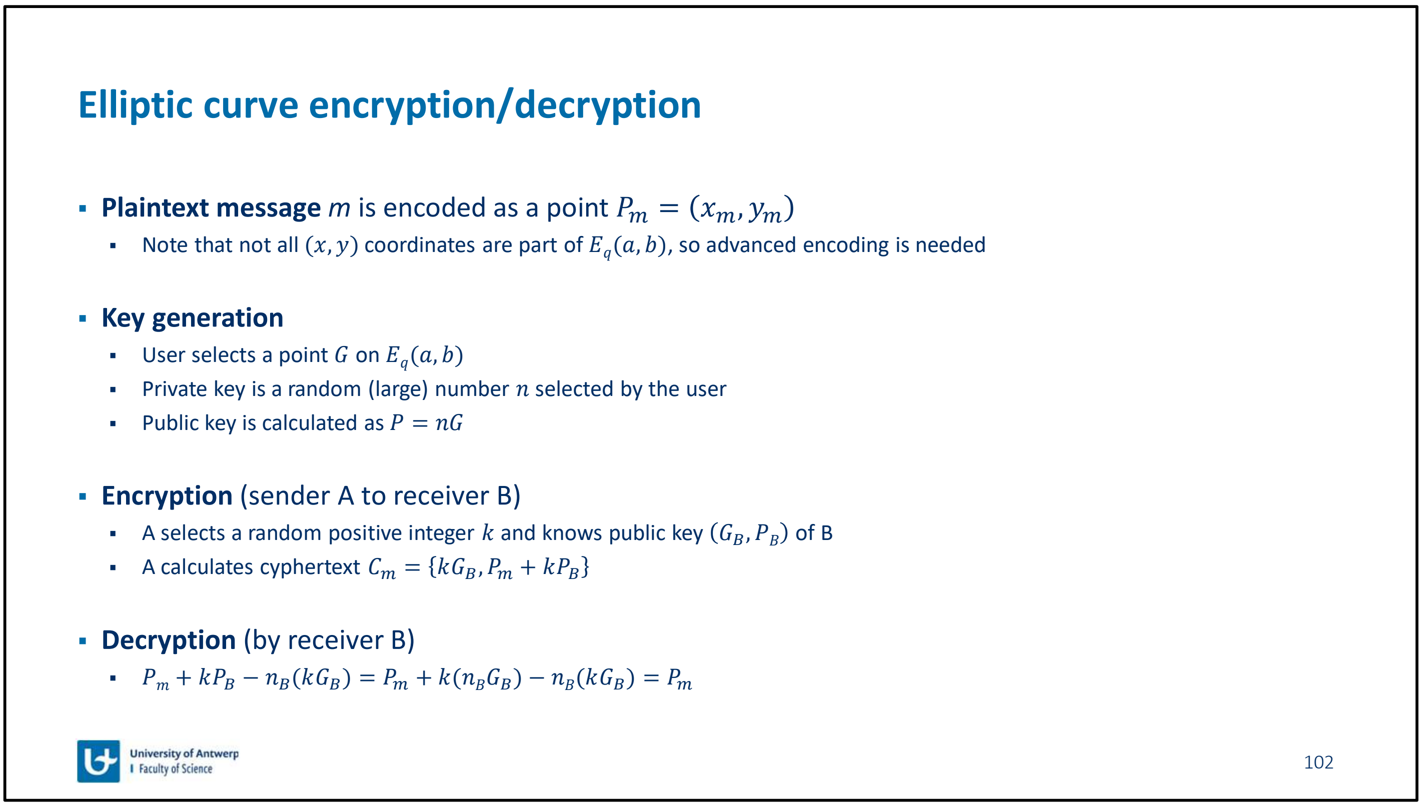
$\lambda = \left(\frac{y_Q - y_P}{x_Q - x_P}\right) \bmod p = \left(\frac{20-5}{20-16}\right) \bmod 23 = \left(\frac{15}{4}\right) \bmod 23 = (15 \times 6) \bmod 23$
$= 21$

$x_{2P} = \left(\lambda^2 - x_P - x_Q\right) \bmod p = \left(21^2 - 16 - 20\right) \bmod 23 = 405 \bmod 23 = 14$
$y_{2P} = (\lambda(x_P - x_R) - y_P) \bmod p = (21(16 - 14) - 5) \bmod 23 = 37 \bmod 23 = 14$

$3P = (14, 14)$

Step 3 (calculation omitted): $4P = P + 3P = (19, 20)$
Step 5 (calculation omitted): $5P = P + 4P = (13, 10)$
…
Step 9 (calculation omitted): $4P + 3P = \boxed{(4, 5)}$

$9P = Q \Leftrightarrow k = 9$



Elliptic curve encryption/decryption

- **Plaintext message** m is encoded as a point $P_m = (x_m, y_m)$
  - Note that not all (x, y) coordinates are part of $E_q(a, b)$, so advanced encoding is needed
- **Key generation**
  - User selects a point G on $E_q(a, b)$
  - Private key is a random (large) number n selected by the user
  - Public key is calculated as $P = nG$
- **Encryption** (sender A to receiver B)
  - A selects a random positive integer k and knows public key (G_B, P_B) of B
  - A calculates cyphertext $C_m = \{kG_B, P_m + kP_B\}$
- **Decryption** (by receiver B)
  - $P_m + kP_B - n_B(kG_B) = P_m + k(n_B G_B) - n_B(kG_B) = P_m$

University of Antwerp
I Faculty of Science

102

Several approaches to encryption/decryption using elliptic curves have been analyzed in the literature. In this subsection, we look at perhaps the simplest. The first task in this system is to encode the plaintext message m to be sent as an (x, y) point P_m.

It is the point P_m that will be encrypted as a ciphertext and subsequently decrypted. Note that we cannot simply encode the message as the x or y coordinate of a point, because not all such coordinates are in $E_q(a, b)$. Again, there are several approaches to this encoding, which we will not address here, but suffice it to say that there are relatively straightforward techniques that can be used.

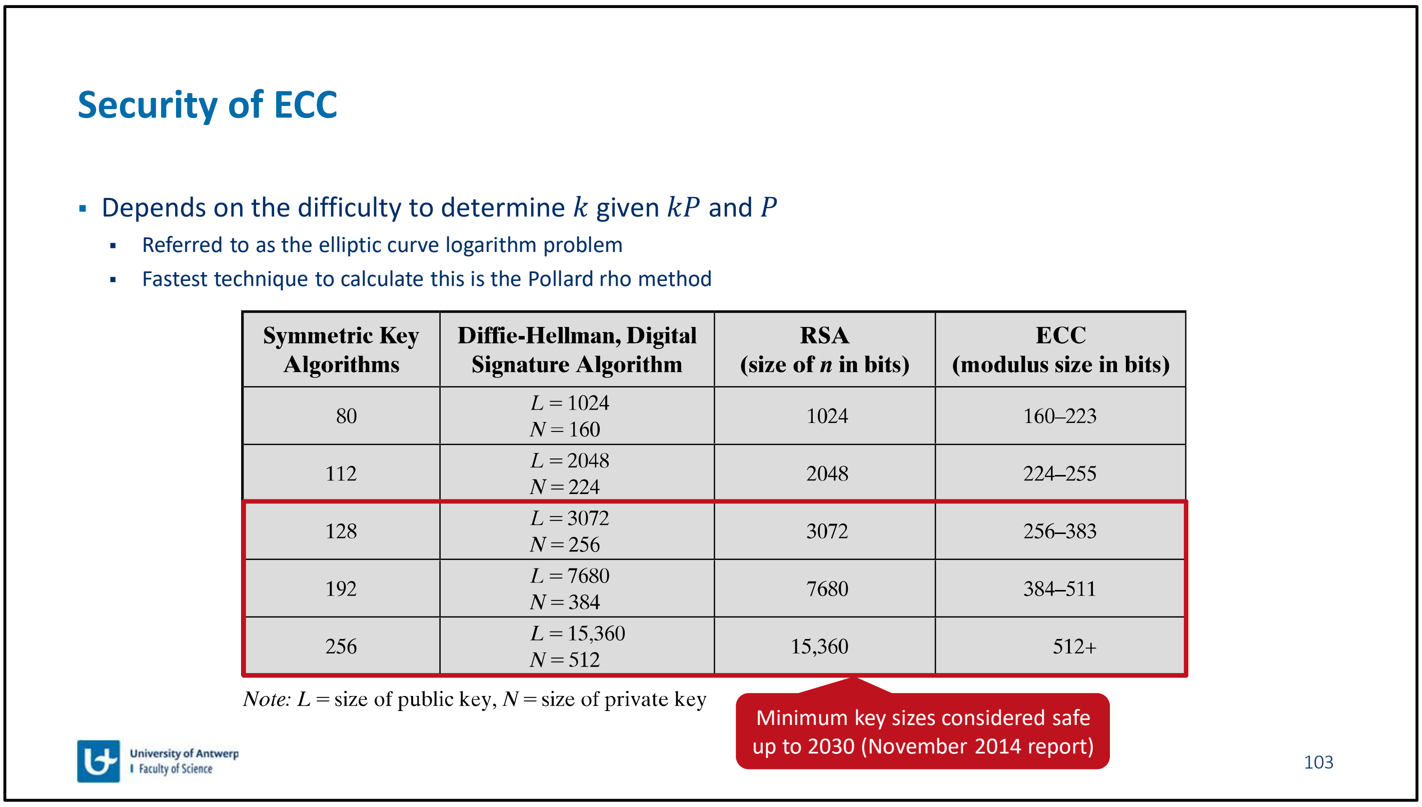
An encryption/decryption system requires a point G and an elliptic group $E_q(a, b)$ as parameters. Each user selects a private key n and generates a public key $P = n * G$. To encrypt and send a message P_m to B, A chooses a random positive integer k and produces the ciphertext C_m consisting of the pair of points:

$$C_m = \{kG, P_m + kP_B\}$$

Note that A has used B's public key P_B. To decrypt the ciphertext, B multiplies the first point in the pair by B's private key and subtracts the result from the second point:

$$P_m + kP_B - n_B(kG) = P_m + k(nG_B) - n_B(kG_B) = P_m$$

A has masked the message P_m by adding kP_B to it. Nobody but A knows the value of k, so even though P_B is a public key, nobody can remove the mask kP_B. However, A also includes a "clue," which is enough to remove the mask if one knows the private key n_B. For an attacker to recover the message, the attacker would have to compute k given G and kG, which is assumed to be hard.



Security of ECC

- Depends on the difficulty to determine k given kP and P
  - Referred to as the elliptic curve logarithm problem
  - Fastest technique to calculate this is the Pollard rho method

Symmetric Key Algorithms	Diffie-Hellman, Digital Signature Algorithm	RSA (size of *n* in bits)	ECC (modulus size in bits)
80	$L = 1024$ $N = 160$	1024	160–223
112	$L = 2048$ $N = 224$	2048	224–255
128	$L = 3072$ $N = 256$	3072	256–383
192	$L = 7680$ $N = 384$	7680	384–511
256	$L = 15{,}360$ $N = 512$	15,360	512+

Note: L = size of public key, N = size of private key

Minimum key sizes considered safe up to 2030 (November 2014 report)

University of Antwerp
Faculty of Science

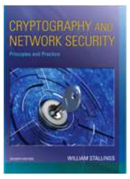
103

The security of ECC depends on how difficult it is to determine *k* given *kP* and *P*. This is referred to as the elliptic curve logarithm problem. The fastest known technique for taking the elliptic curve logarithm is known as the Pollard rho method. The table, from NIST SP 800-57 (*Recommendation for Key Management—Part 1: General*, September 2015), compares various algorithms by showing comparable key sizes in terms of computational effort for cryptanalysis. As can be seen, a considerably smaller key size can be used for ECC compared to RSA. Based on this analysis, SP 800-57 recommends that at least through 2030, acceptable key lengths are from 3072 to 14,360 bits for RSA and 256 to 512 bits for ECC. Similarly, the European Union Agency for Network and Information Security (ENISA) recommends in their 2014 report (*Algorithms, Key Size and Parameters report—2014*, November 2014) minimum key lengths for future system of 3072 bits and 256 bits for RSA and ECC, respectively.

Analysis indicates that for equal key lengths, the computational effort required for ECC and RSA is comparable. Thus, there is a computational advantage to using ECC with a shorter key length than a comparably secure RSA.

Summary on other public-key schemes

- **Elgamal Cryptography**
  - Private key is discrete logarithm of the public key
  - Encryption and decryption use modular multiplication and exponentiation operations
  - Sender calculates **unique one-time key** for each message (key-reuse is unsafe)
  - Security is based on difficulty to calculate discrete logarithms
- **Elliptic Curve Cryptography (ECC)**
  - Uses elliptic curve addition and multiplication over prime ($\mathbb{Z}_p$) or Galois ($GF(2^m)$) fields
  - Security is based on difficulty to calculate discrete logarithms over elliptic curves ($Q = kP$)
  - **Computationally efficient** as it has the same security as RSA with a much smaller key size

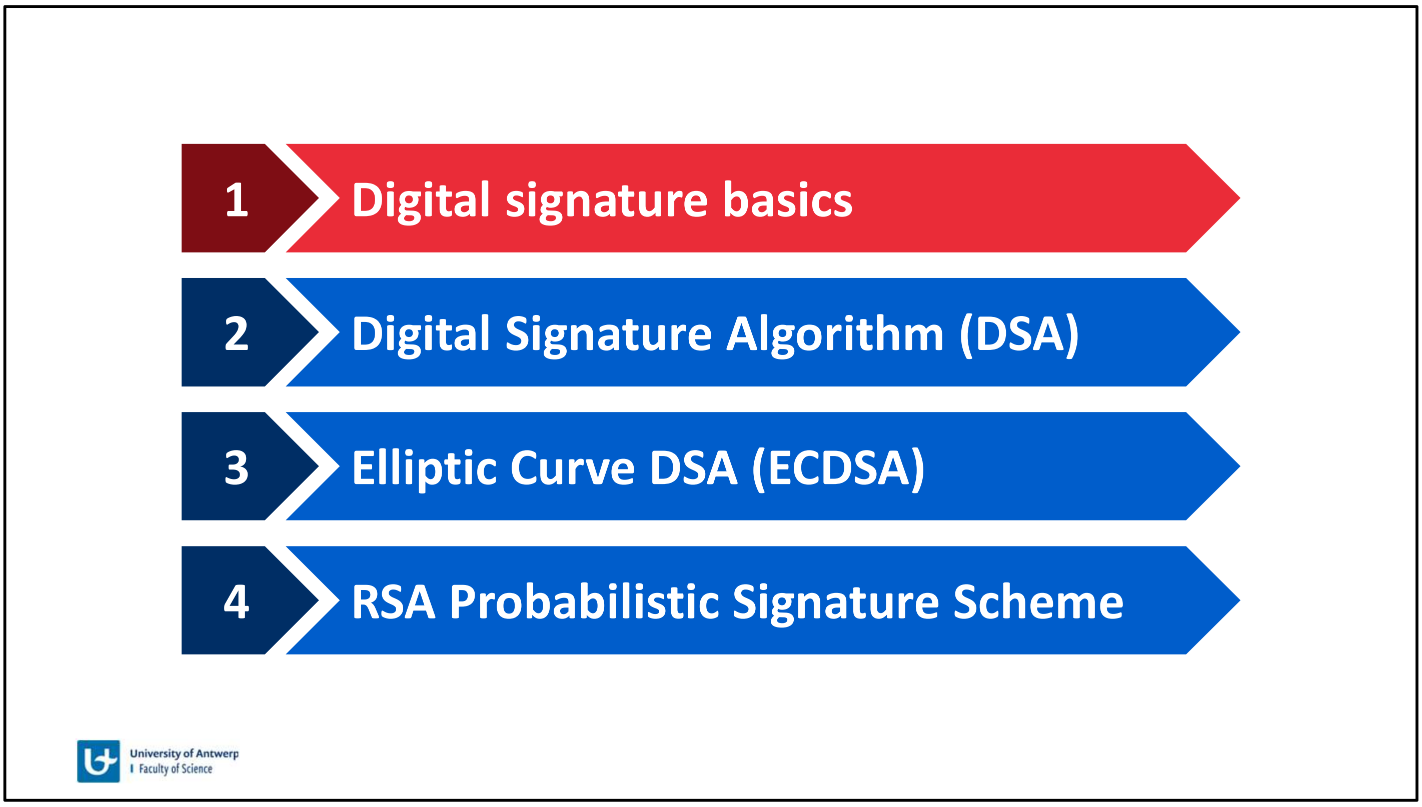


Link with the book
- [optional] Chapter 2 (Section 2.8)
- Chapter 10 (Sections 10.2, 10.3, 10.4)

Link with the videos
- Week 6 (Public Key Encryption From Diffie-Hellman: ElGamal)

Digital Signatures

University of Antwerp
I Faculty of Science



1 Digital signature basics

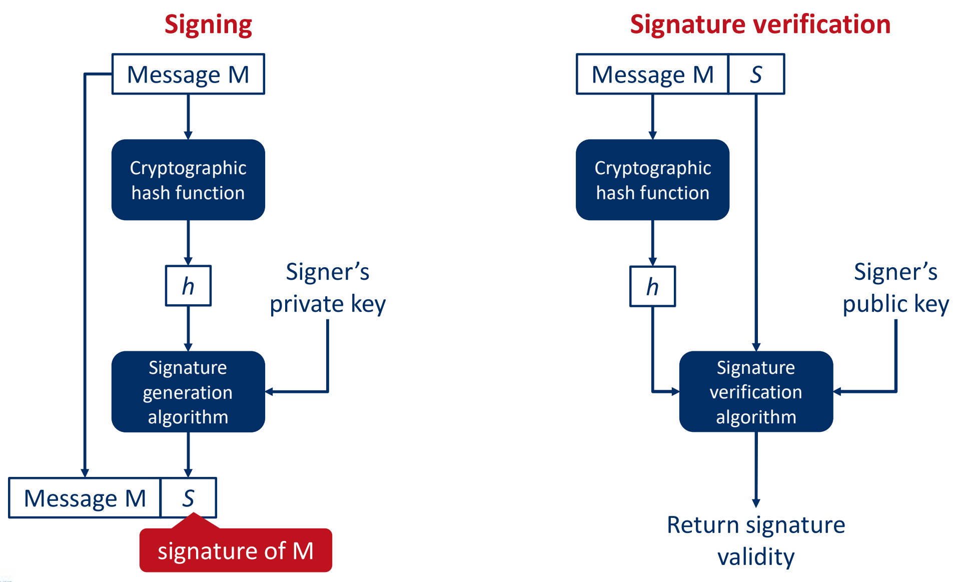
2 Digital Signature Algorithm (DSA)

3 Elliptic Curve DSA (ECDSA)

4 RSA Probabilistic Signature Scheme

University of Antwerp
Faculty of Science

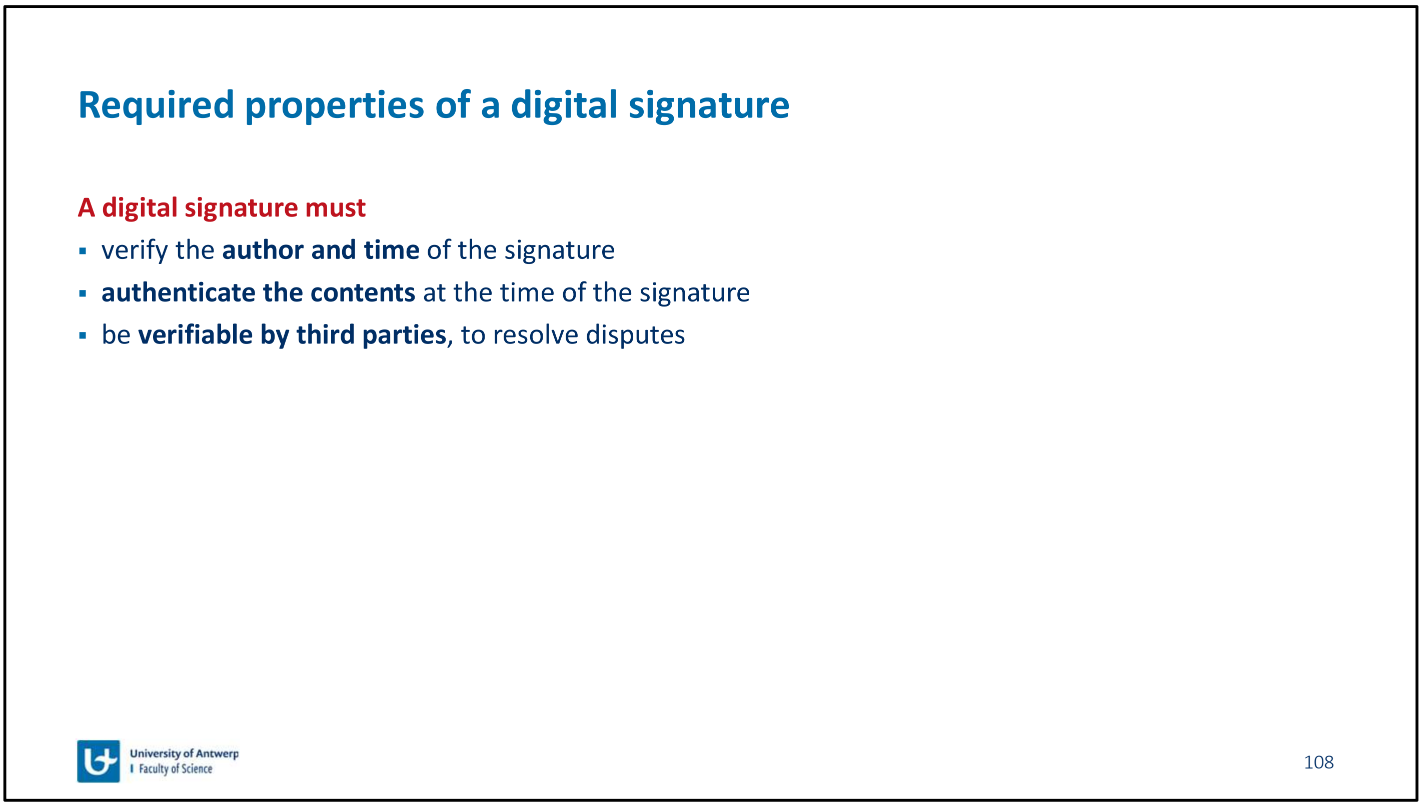
Essential elements of the digital signature process


Signing
Message M
Cryptographic hash function
h
Signer's private key
Signature generation algorithm
Message M
S
signature of M
Signature verification
Message M
S
Cryptographic hash function
h
Signer's public key
Signature verification algorithm
Return signature validity

University of Antwerp | Faculty of Science

The most important development from the work on public-key cryptography is the digital signature. The digital signature provides a set of security capabilities that would be difficult to implement in any other way.

Suppose that Bob wants to send a message to Alice. Although it is not important that the message be kept secret, he wants Alice to be certain that the message is indeed from him. For this purpose, Bob uses a secure hash function, such as SHA-512, to generate a hash value for the message. That hash value, together with Bob's private key serves as input to a digital signature generation algorithm, which produces a short block that functions as a **digital signature**. Bob sends the message with the signature attached. When Alice receives the message plus signature, she (1) calculates a hash value for the message; (2) provides the hash value and Bob's public key as inputs to a digital signature verification algorithm. If the algorithm returns the result that the signature is valid, Alice is assured that the message must have been signed by Bob. No one else has Bob's private key and therefore no one else could have created a signature that could be verified for this message with Bob's public key. In addition, it is impossible to alter the message without access to Bob's private key, so the message is authenticated both in terms of source and in terms of data integrity.



Required properties of a digital signature

A digital signature must

- verify the **author and time** of the signature
- **authenticate the contents** at the time of the signature
- be **verifiable by third parties**, to resolve disputes

University of Antwerp
Faculty of Science

108

Message authentication protects two parties who exchange messages from any third party. However, it does not protect the two parties against each other. Several forms of dispute between the two parties are possible.

For example, suppose that John sends an authenticated message to Mary. Consider the following disputes that could arise:

1. Mary may forge a different message and claim that it came from John. Mary would simply have to create a message and append an authentication code using the key that John and Mary share.
2. John can deny sending the message. Because it is possible for Mary to forge a message, there is no way to prove that John did in fact send the message.

Both scenarios are of legitimate concern. Here is an example of the first scenario: An electronic funds transfer takes place, and the receiver increases the amount of funds transferred and claims that the larger amount had arrived from the sender. An example of the second scenario is that an electronic mail message contains instructions to a stockbroker for a transaction that subsequently turns out badly. The sender pretends that the message was never sent.

In situations where there is not complete trust between sender and receiver, something more than authentication is needed. The most attractive solution to this problem is the digital signature. The digital signature must have the following properties:

- It must verify the author and the date and time of the signature.
- It must authenticate the contents at the time of the signature.

- It must be verifiable by third parties, to resolve disputes.

Thus, the digital signature function includes the authentication function.

1 Digital signature basics

2 Digital Signature Algorithm (DSA)

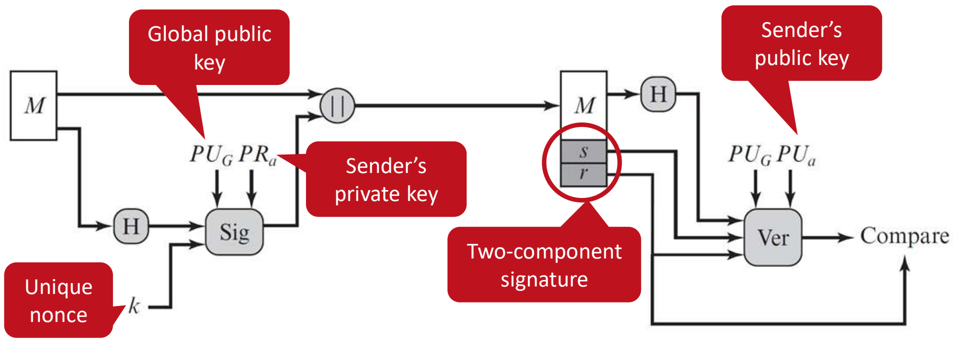
3 Elliptic Curve DSA (ECDSA)

4 RSA Probabilistic Signature Scheme

University of Antwerp
Faculty of Science

NIST Digital Signature Algorithm (DSA)

- DSA was first standardized in 1991 as FIPS 186
- An improved version was standardized in 1993, 1996, 2000, 2009 and 2013
  - Originally based on Elgamal/Schnorr encryption (discrete logarithms) and SHA
  - Latest version supports RSA (RSA-PSS) and elliptic curve variants

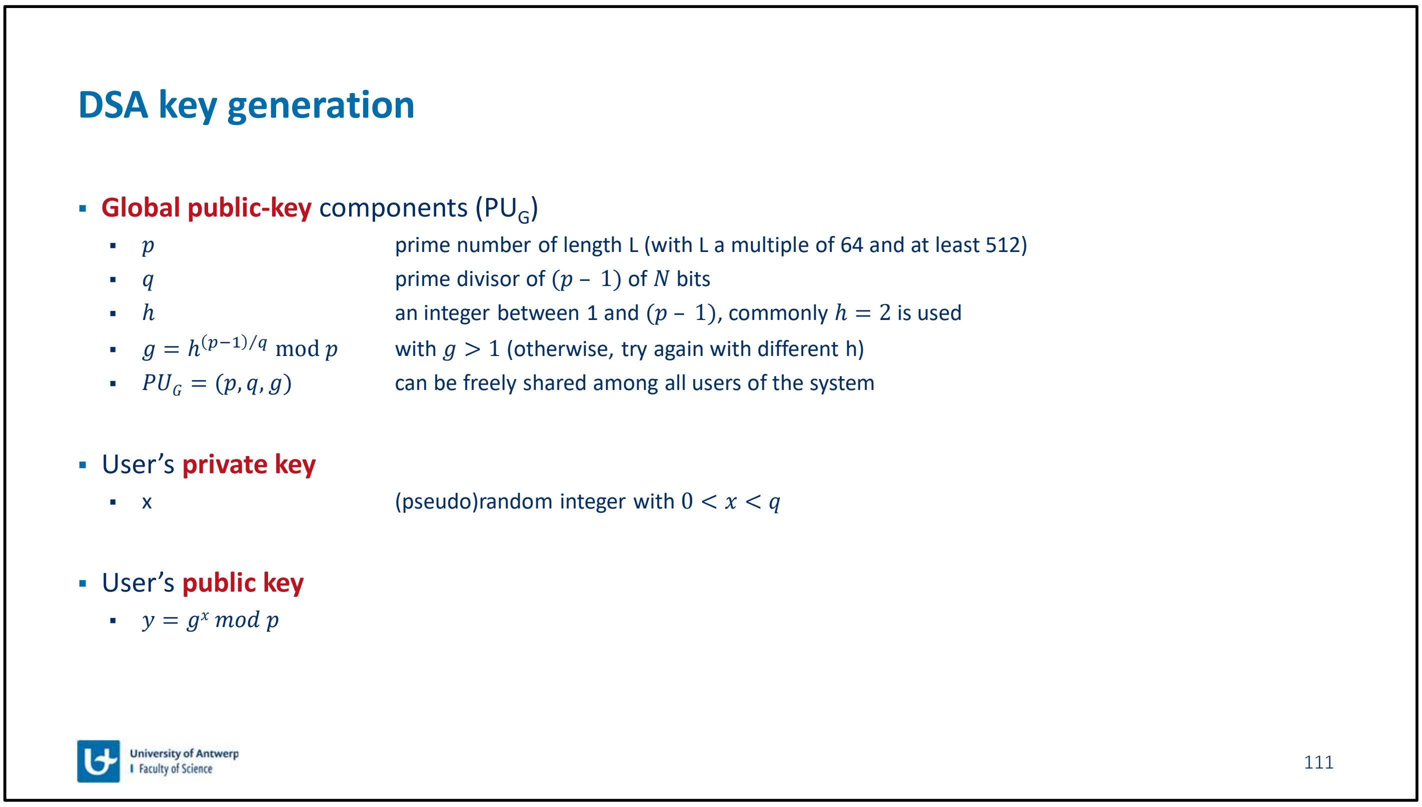


University of Antwerp | Faculty of Science

The National Institute of Standards and Technology (NIST) has published Federal Information Processing Standard FIPS 186, known as the Digital Signature Algorithm (DSA). The DSA makes use of the Secure Hash Algorithm (SHA). The DSA was originally proposed in 1991 and revised in 1993 in response to public feedback concerning the security of the scheme. There was a further minor revision in 1996. In 2000, an expanded version of the standard was issued as FIPS 186-2, subsequently updated to FIPS 186-3 in 2009, and FIPS 186-4 in 2013. This latest version also incorporates digital signature algorithms based on RSA and on elliptic curve cryptography.

The DSA uses an algorithm that is designed to provide only the digital signature function. Unlike RSA, it cannot be used for encryption or key exchange. Nevertheless, it is a public-key technique. The hash code is provided as input to a signature function along with a random number *k* generated for this particular signature. The signature function also depends on the sender's private key (PR_a) and a set of parameters known to a group of communicating principals. We can consider this set to constitute a global public key (PU_G). The result is a signature consisting of two components, labelled *s* and *r*.

At the receiving end, the hash code of the incoming message is generated. The hash code and the signature are inputs to a verification function. The verification function also depends on the global public key as well as the sender's public key (PU_a), which is paired with the sender's private key. The output of the verification function is a value that is equal to the signature component *r* if the signature is valid. The signature function is such that only the sender, with knowledge of the private key, could have produced the valid signature.



DSA key generation

- **Global public-key** components (PU_G)

p	prime number of length L (with L a multiple of 64 and at least 512)
q	prime divisor of $(p - 1)$ of N bits
h	an integer between 1 and $(p - 1)$, commonly $h = 2$ is used
$g = h^{(p-1)/q} \bmod p$	with $g > 1$ (otherwise, try again with different h)
$PU_G = (p, q, g)$	can be freely shared among all users of the system

- User's **private key**

x	(pseudo)random integer with $0 < x < q$

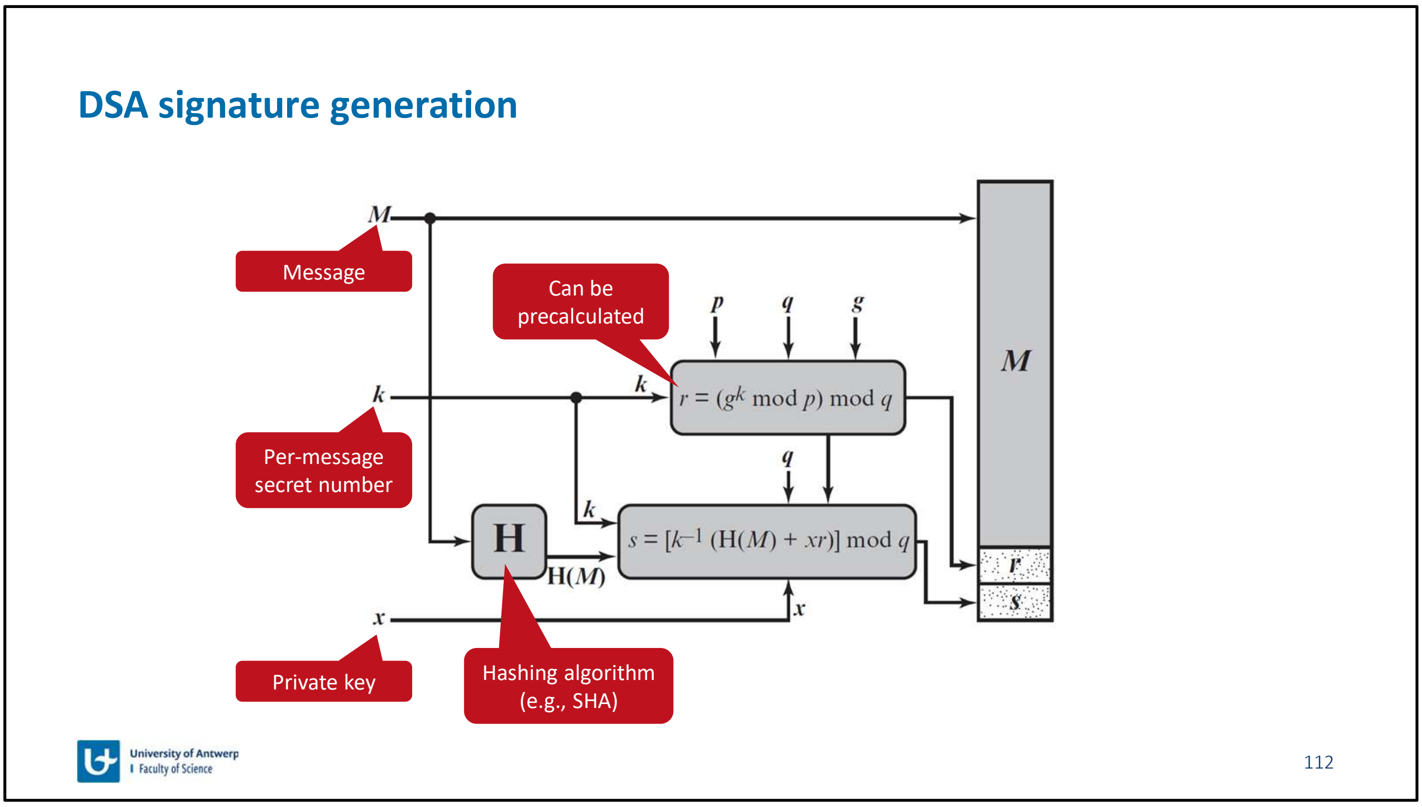
- User's **public key**
  - $y = g^x \, mod \, p$

University of Antwerp | Faculty of Science

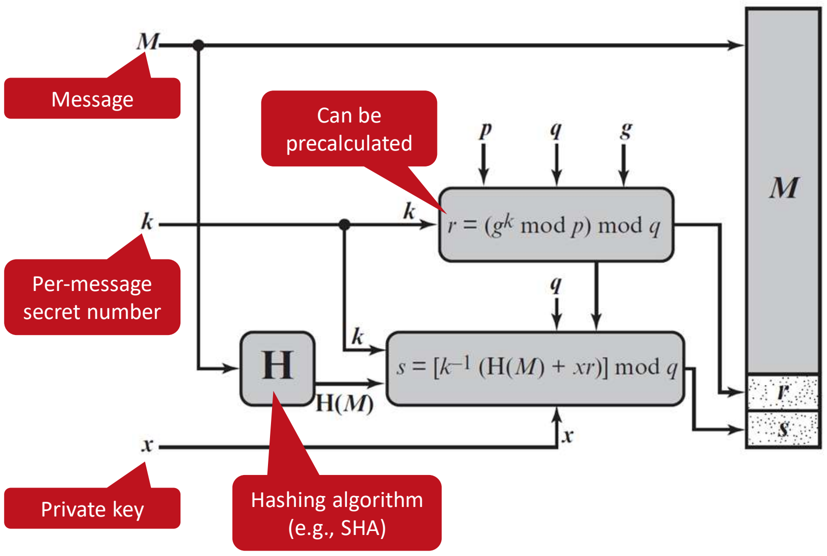
111

There are three parameters that are public and can be common to a group of users. An *N*-bit prime number *q* is chosen. Next, a prime number *p* is selected such that *q* divides (*p* - 1). The original DSA constrained the length (in bits) L of p to be a multiple of 64 between 512 and 1024 inclusive. NIST 800-57 recommends lengths of 2048 (or 3072) for keys with security lifetimes extending beyond 2010 (or 2030). Finally, *g* is chosen to be of the form $h^{(p-1)/q}$ mod *p*, where *h* is an integer between 1 and (*p* - 1) with the restriction that *g* must be greater than 1. Thus, the global public-key components of DSA are the same as in the Schnorr signature scheme.

With these parameters in hand, each user selects a private key and generates a public key. The private key *x* must be a number from 1 to (*q* - 1) and should be chosen randomly or pseudorandomly. The public key is calculated from the private key as $y = g^x$ mod *p*. The calculation of *y* given *x* is relatively straightforward. However, given the public key *y*, it is believed to be computationally infeasible to determine *x*, which is the discrete logarithm of *y* to the base *g*, mod *p.*



DSA signature generation



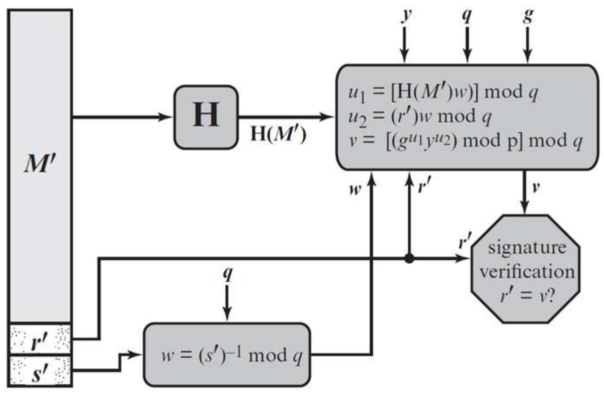
M
Message
Can be precalculated
p
q
g
k
$r = (g^k \bmod p) \bmod q$
M
Per-message secret number
q
k
H
$s = [k^{-1}(H(M) + xr)] \bmod q$
$H(M)$
r
s
x
x
Private key
Hashing algorithm (e.g., SHA)

University of Antwerp
Faculty of Science

112

The signature of a message *M* consists of the pair of numbers *r* and *s*, which are functions of the public key components (*p, q, g*), the user's private key (*x*), the hash code of the message H(*M*), and an additional integer *k* that should be generated randomly or pseudorandomly and be unique for each signing.

DSA signature verification



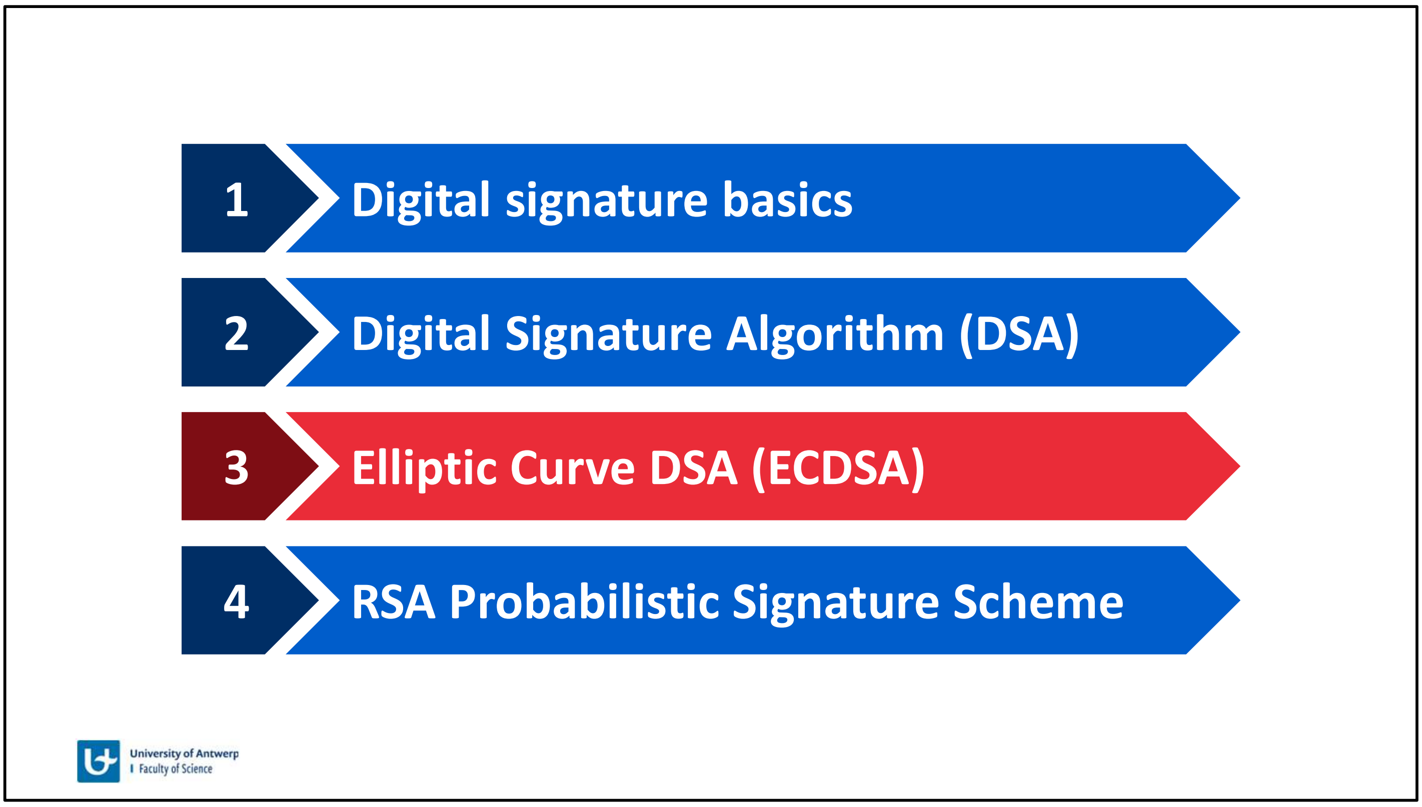
y
q
g
$u_1 = [H(M')w)] \bmod q$
$u_2 = (r')w \bmod q$
$v = [(g^{u_1} y^{u_2}) \bmod p] \bmod q$
H
$H(M')$
M'
w
r'
v
r'
signature verification $r' = v$?
q
$w = (s')^{-1} \bmod q$
r'
s'

Proof that r = v is provided in Appendix K of Stallings

Let *M*, *r*′, and *s*′ be the received versions of *M*, *r*, and *s*, respectively. Verification is performed using the formulas shown in the figure. The receiver generates a quantity *v* that is a function of the public key components, the sender's public key, the hash code of the incoming message, and the received versions of *r* and *s*. If this quantity matches the *r* component of the signature, then the signature is validated.

Note that the test at the end is on the value *r*, which does not depend on the message at all. Instead, *r* is a function of *k* and the three global public-key components. The multiplicative inverse of *k* (mod *q*) is passed to a function that also has as inputs the message hash code and the user's private key. The structure of this function is such that the receiver can recover *r* using the incoming message and signature, the public key of the user, and the global public key.

Given the difficulty of taking discrete logarithms, it is infeasible for an opponent to recover *k* from *r* or to recover *x* from *s*. Another point worth noting is that the only computationally demanding task in signature generation is the exponential calculation g^k mod *p*. Because this value does not depend on the message to be signed, it can be computed ahead of time. Indeed, a user could precalculate a number of values of *r* to be used to sign documents as needed. The only other somewhat demanding task is the determination of a multiplicative inverse, k^{-1}. Again, a number of these values can be precalculated.



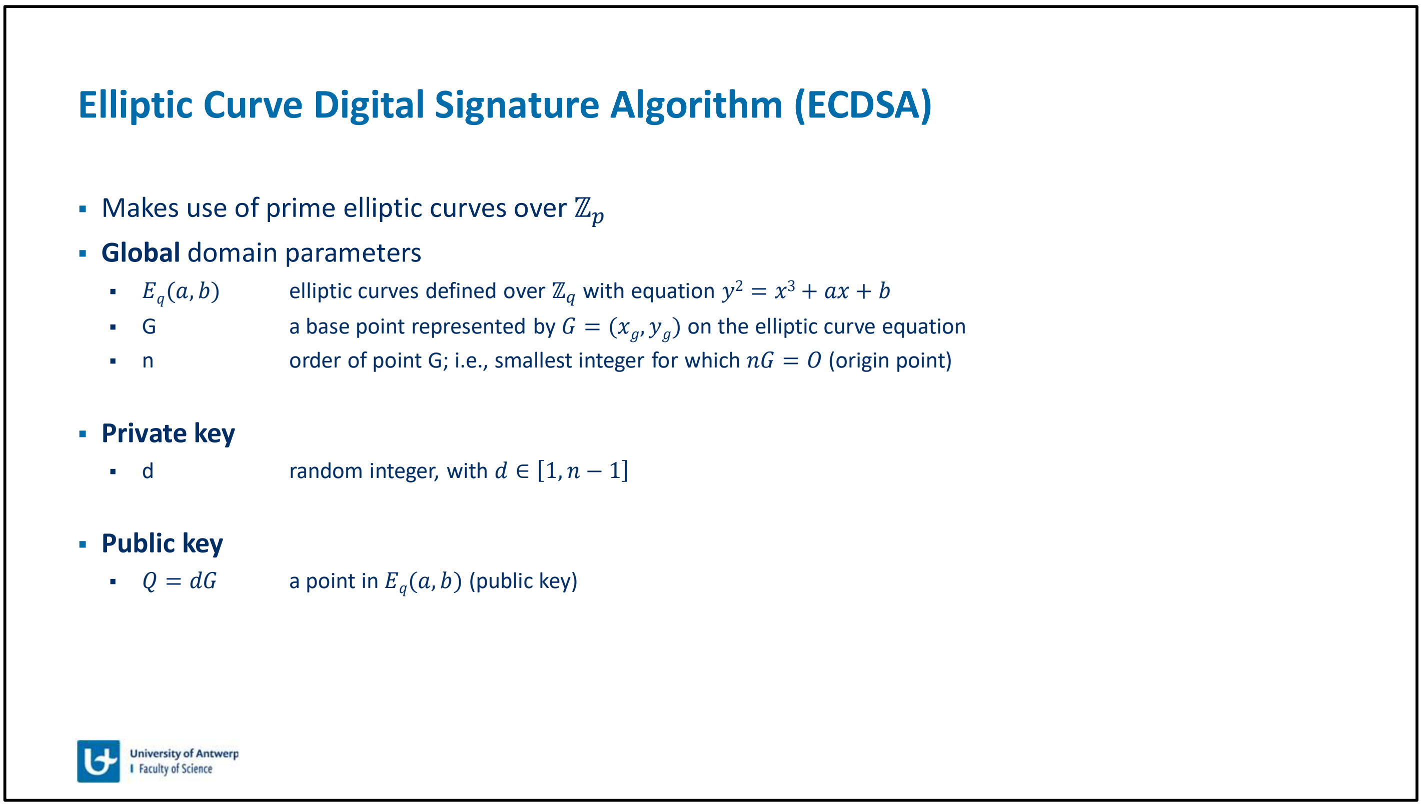
1 Digital signature basics

2 Digital Signature Algorithm (DSA)

3 Elliptic Curve DSA (ECDSA)

4 RSA Probabilistic Signature Scheme

University of Antwerp
Faculty of Science



Elliptic Curve Digital Signature Algorithm (ECDSA)

- Makes use of prime elliptic curves over $\mathbb{Z}_p$
- **Global** domain parameters

$E_q(a, b)$	elliptic curves defined over $\mathbb{Z}_q$ with equation $y^2 = x^3 + ax + b$
G	a base point represented by $G = (x_g, y_g)$ on the elliptic curve equation
n	order of point G; i.e., smallest integer for which $nG = O$ (origin point)

- **Private key**

d	random integer, with $d \in [1, n-1]$

- **Public key**

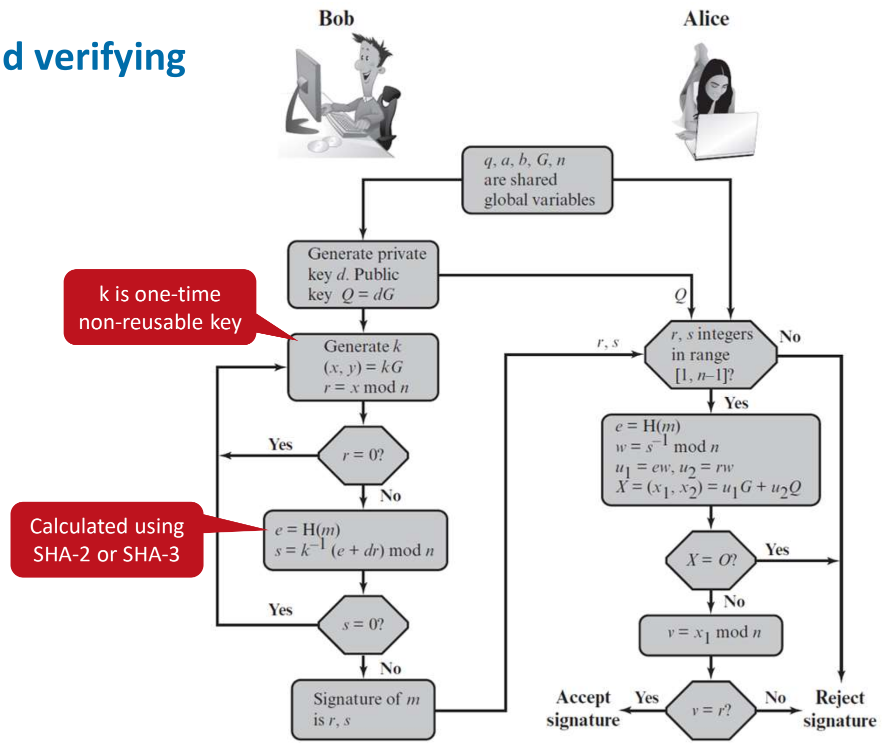
$Q = dG$	a point in $E_q(a, b)$ (public key)

University of Antwerp
I Faculty of Science

As was mentioned, the 2009 version of FIPS 186 includes a new digital signature technique based on elliptic curve cryptography, known as the **Elliptic Curve Digital Signature Algorithm (ECDSA)**. ECDSA is enjoying increasing acceptance due to the efficiency advantage of elliptic curve cryptography, which yields security comparable to that of other schemes with a smaller key bit length.

Recall that two families of elliptic curves are used in cryptographic applications: prime curves over Z_p and binary curves over GF(2^m). For ECDSA, prime curves are used. Each signer must generate a pair of keys, one private and one public.

ECDSA signing and verifying


Bob
Alice
q, a, b, G, n are shared global variables
Generate private key d. Public key $Q = dG$
Q
k is one-time non-reusable key
Generate k
$(x, y) = kG$
$r = x \bmod n$
Yes
$r = 0$?
No
Calculated using SHA-2 or SHA-3
$e = H(m)$
$s = k^{-1}(e + dr) \bmod n$
Yes
$s = 0$?
No
Signature of m is r, s
r, s
r, s integers in range $[1, n–1]$?
No
Yes
$e = H(m)$
$w = s^{-1} \bmod n$
$u_1 = ew, u_2 = rw$
$X = (x_1, x_2) = u_1G + u_2Q$
$X = O$?
Yes
No
$v = x_1 \bmod n$
Accept signature
Yes
$v = r$?
No
Reject signature

116

With the public domain parameters and a private key in hand, Bob generates a digital signature of 320 bytes for message *m* using the following steps:

1. Select a random or pseudorandom integer k, $k \in [1, n - 1]$
2. Compute point $P = (x, y) = kG$ and $r = x \bmod n$. If $r = 0$ then goto step 1
3. Compute $e = H(m)$, where H is one of the SHA-2 or SHA-3 hash functions.
4. Compute $s = k^{-1}(e + dr) \bmod n$. If $s = O$ then goto step 1
5. The signature of message m is the pair (r, s).

Alice knows the public domain parameters and Bob's public key. Alice is presented with Bob's message and digital signature and verifies the signature using
the following steps:

1. Verify that r and s are integers in the range 1 through $n - 1$
2. Using SHA, compute the 160-bit hash value $e = H(m)$
3. Compute $w = s^{-1} \bmod n$
4. Compute $u_1 = ew$ and $u_2 = rw$
5. Compute the point $X = (x_1, y_1) = u_1G + u_2Q$
6. If $X = O$, reject the signature else compute $v = x_1 \bmod n$
7. Accept Bob's signature if and only if $v = r$

ECDSA validation proof

We should show that $v = r$ if the message was indeed signed by Bob

Given: $s = k^{-1}(e + dr) \bmod n$

Then: $k = s^{-1}(e + dr) \bmod n$

$k = (s^{-1}e + s^{-1}dr) \bmod n$

$k = (we + wdr) \bmod n$

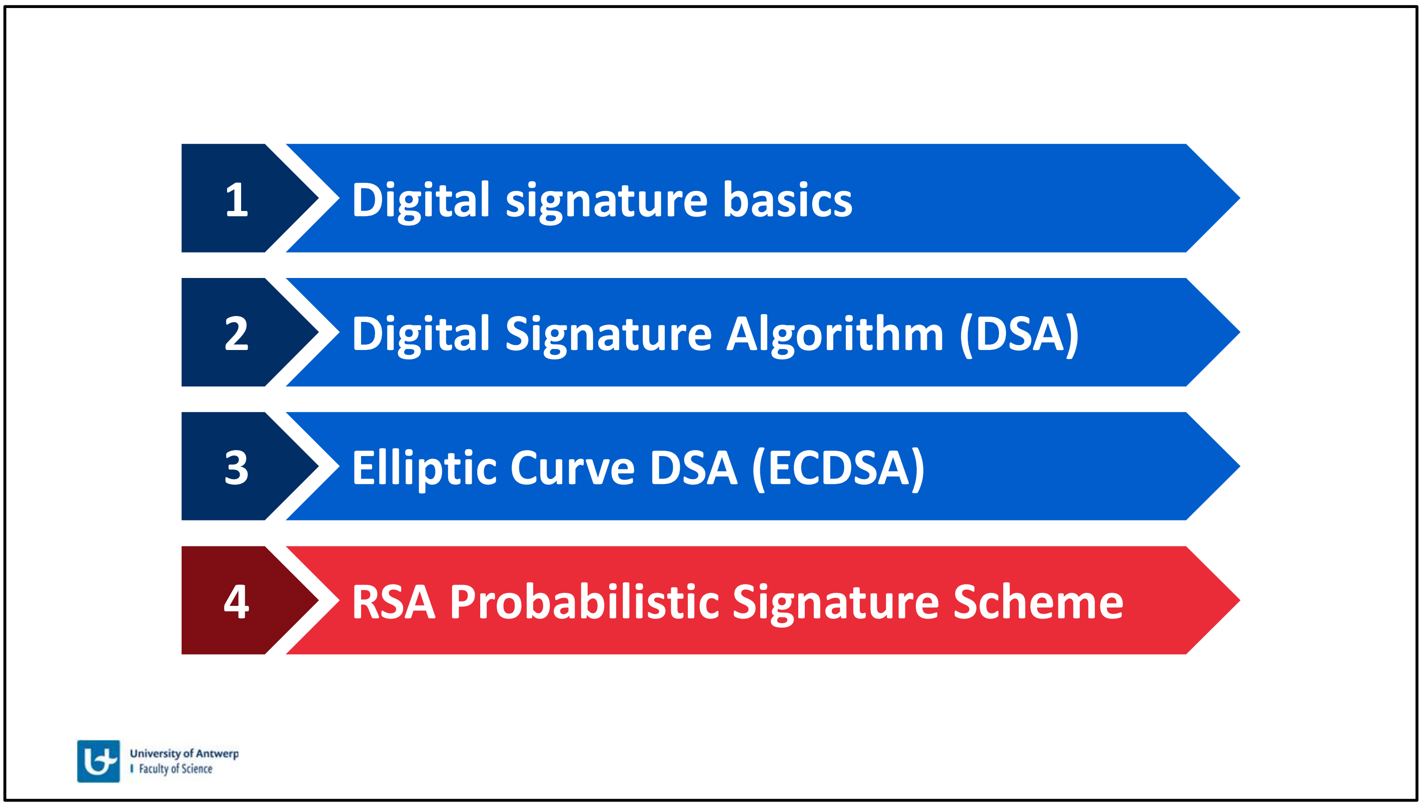
$k = (u_1 + u_2 d) \bmod n$

Also: $\mathrm{X} = (x_1, y_1) = u_1 G + u_2 Q = u_1 G + u_2 dG = (u_1 + u_2 d)G = kG$

Thus: $kG = (x, y) = X = (x_1, y_1) \Rightarrow x = x_1$

Finally: $v = x_1 \bmod n = x \bmod n = r$

University of Antwerp
Faculty of Science



1 Digital signature basics

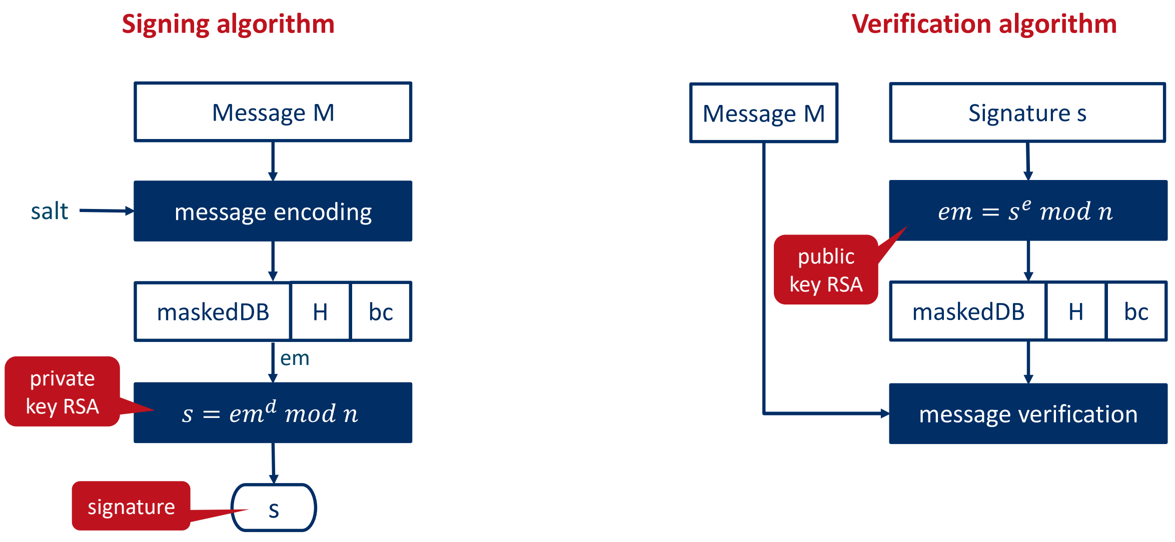
2 Digital Signature Algorithm (DSA)

3 Elliptic Curve DSA (ECDSA)

4 RSA Probabilistic Signature Scheme

University of Antwerp
Faculty of Science

RSA-PSS: Digital signatures based on RSA



119

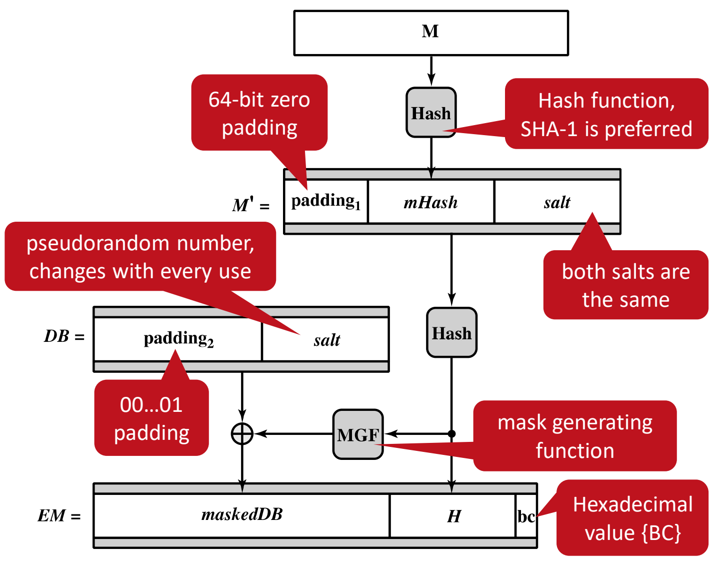
In addition to the NIST Digital Signature Algorithm and ECDSA, the 2009 version of FIPS 186 also includes several techniques based on RSA, all of which were developed by RSA Laboratories and are in wide use. We discuss the RSA Probabilistic Signature Scheme (RSA-PSS), which is the latest of the RSA schemes and the one that RSA Laboratories recommends as the most secure of the RSA schemes.

We show how the signature is formed by a signer with private key {*d*, *n*} and public key {*e*, *n*}. Treat the octet string *EM* as an unsigned, nonnegative binary integer *m*. The signature *s* is formed by encrypting *m* as follows:

$$s = m^d \bmod n$$

Let *k* be the length in octets of the RSA modulus *n*. For example, if the key size for RSA is 2048 bits, then *k* = 2048/8 = 256. Then convert the signature value *s* into the octet string *S* of length *k* octets.

RSA-PSS: Message encoding algorithm



University of Antwerp
Faculty of Science

120

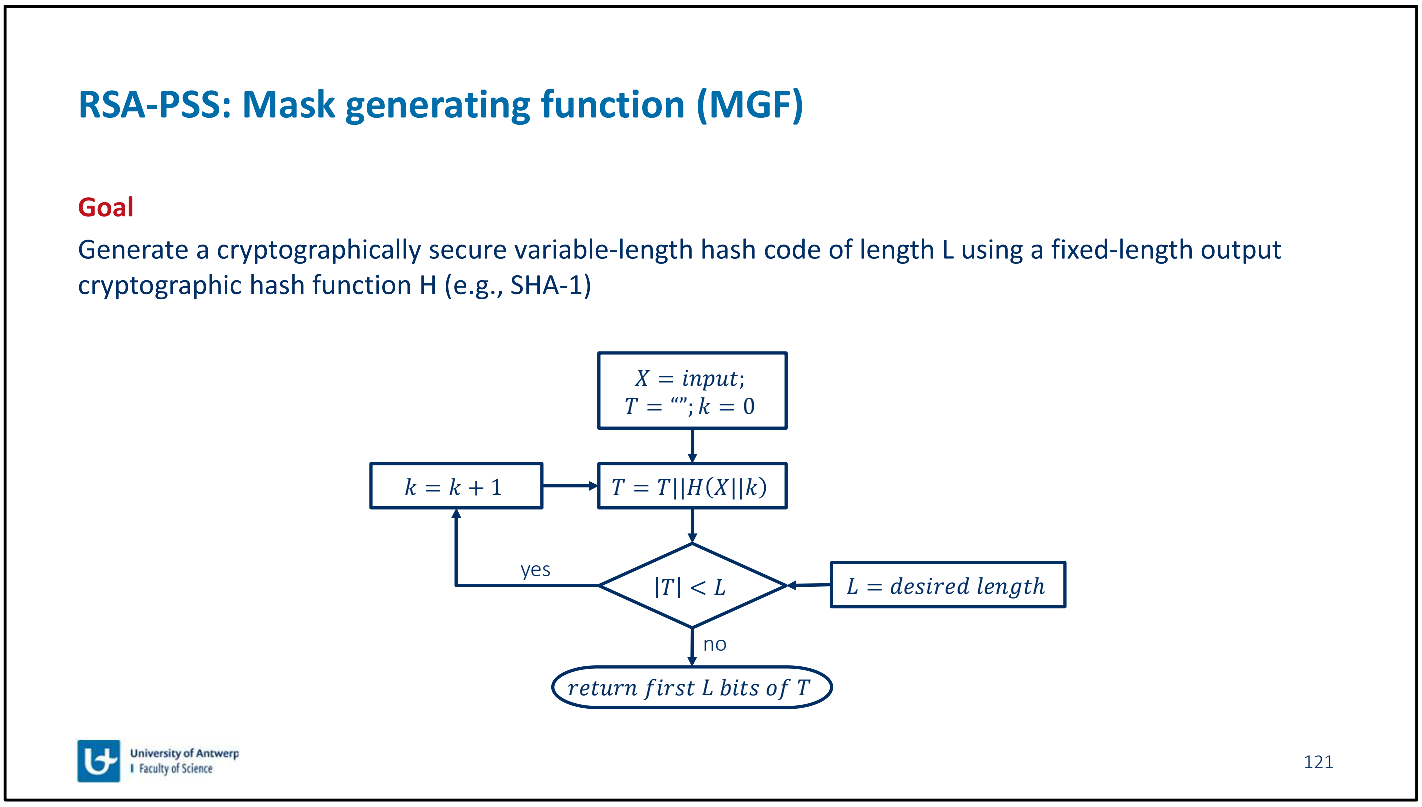
The first stage in generating an RSA-PSS signature of a message *M* is to generate from *M* a fixed-length message digest, called an encoded message. We define the following parameters and functions:

- **Options**
    - **Hash:** hash function with output *hLen* octets. The current preferred alternative is SHA-1, which produces a 20-octet hash value.
    - **MGF:** mask generation function. The current specification calls for MGF1.
    - **sLen:** length in octets of the salt. Typically *sLen* = *hLen*, which for the current version is 20 octets.
- **Input**
    - **M:** message to be encoded for signing.
    - **emBits:** This value is one less than the length in bits of the RSA modulus *n*.
- **Output**
    - **EM:** encoded message. This is the message digest that will be encrypted to form the digital signature.
- **Parameters**
    - ***emLen:*** length of *EM* in octets = $\lceil emBits / 8 \rceil$.
    - **padding$_1$:** hexadecimal string 00 00 00 00 00 00 00 00; that is, a string of 64 zero bits.
    - **padding$_2$:** hexadecimal string of 00 octets with a length (*emLen* - *sLen* - *hLen* - 2) octets, followed by the hexadecimal octet with value 01.
    - **salt:** a pseudorandom number.
    - **bc:** the hexadecimal value BC.

The encoding process consists of the following steps.

1. Generate the hash value of *M*: *mHash* = Hash(*M*)
2. Generate a pseudorandom octet string *salt* and form block *M′* = padding$_1$ || *mHash* || *salt*
3. Generate the hash value of *M′*: *H* = Hash(*M′*)
4. Form data block *DB* = padding$_2$ || *salt*
5. Calculate the MGF value of *H*: *dbMask* = MGF(*H*, *emLen* - *hLen* - 1)
6. Calculate *maskedDB* = *DB* $\oplus$ *dbMsk*
7. Set the leftmost 8 *emLen* - *emBits* bits of the leftmost octet in *maskedDB* to 0
8. *EM* = *maskedDB* || *H* || 0xbc

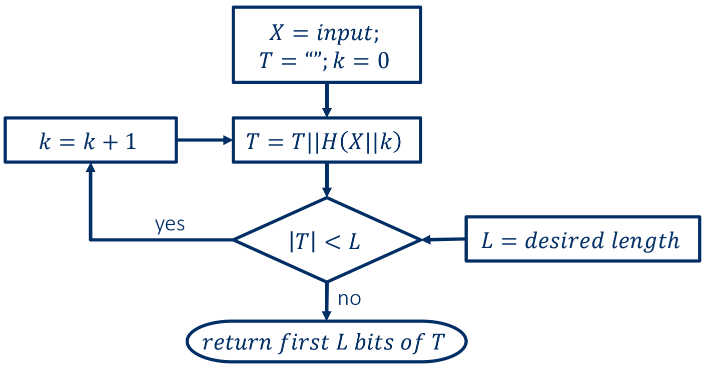
We make several comments about the complex nature of this message digest algorithm. All of the RSA-based standardized digital signature schemes involve appending one or more constants (e.g., padding$_1$ and padding$_2$) in the process of forming the message digest. The objective is to make it more difficult for an adversary to find another message that maps to the same message digest as a given message or to find two messages that map to the same message digest. RSA-PSS also incorporates a pseudorandom number, namely the salt. Because the salt changes with every use, signing the same message twice using the same private key will yield two different signatures. This is an added measure of security.



RSA-PSS: Mask generating function (MGF)

Goal

Generate a cryptographically secure variable-length hash code of length L using a fixed-length output cryptographic hash function H (e.g., SHA-1)



University of Antwerp | Faculty of Science

121

Before explaining the RSA-PSS operation, we need to describe the mask generation function (MGF) used as a building block. MGF(*X*, *maskLen*) is a pseudorandom function that has as input parameters a bit string *X* of any length and the desired length *L* in octets of the output. MGFs are typically based on a secure cryptographic hash function such as SHA-1. An MGF based on a hash function is intended to be a cryptographically secure way of generating a message digest, or hash, of variable length based on an underlying cryptographic hash function that produces a fixed-length output.

The MGF function used in the current specification for RSA-PSS is MGF1, with the following parameters:

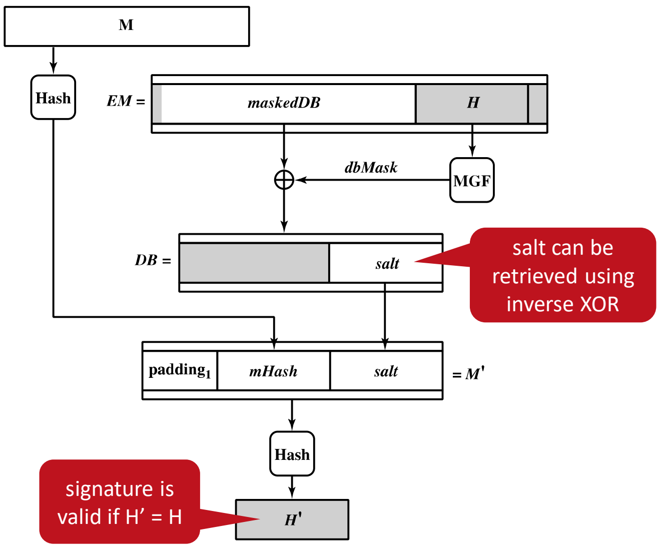
- **Options**
    - **Hash:** hash function with output hLen octets
- **Input**
    - **X:** octet string to be masked
    - **maskLen L:** length in octets of the mask
- **Output**
    - **mask:** an octet string of length maskLen

In essence, MGF1 does the following. If the length of the desired output is equal to the length of the hash value (*maskLen* = *hLen*), then the output is the hash of the input value *X* concatenated with a 32-bit counter value of 0. If *maskLen* is greater than *hLen*, the MGF1 keeps iterating by hashing *X* concatenated with the counter and appending that to the current string *T*. So that the output is

Hash (*X* || 0) || Hash (*X* || 1) || … || Hash(*X* || *k*)

This is repeated until the length of *T* is greater than or equal to *maskLen*, at which point the output is the first *maskLen* octets of *T*.

RSA-PSS: Signature verification algorithm


M
Hash
EM =
maskedDB
H
dbMask
MGF
DB =
salt
salt can be retrieved using inverse XOR
padding$_1$
mHash
salt
= M'
Hash
signature is valid if H' = H
H'

University of Antwerp
Faculty of Science

122

For signature verification, treat the signature *S* as an unsigned, nonnegative binary integer *s*. The message digest *m* is recovered by decrypting *s* as follows:

$m = s^e \bmod n$

Then, convert the message representative *m* to an encoded message *EM* of length $emLen = \lceil (modBits - 1)/8 \rceil$ octets, where *modBits* is the length in bits of the RSA modulus *n*.

The EM verification parameters can be described as follows:

- **Options**
  - **Hash:** hash function with output *hLen* octets.
  - **MGF:** mask generation function.
  - ***sLen:*** length in octets of the salt.
- **Input**
  - ***M:*** message to be verified.
  - ***EM:*** the octet string representing the decrypted signature, with length $\mathrm{em}Len = \lceil emBits/8 \rceil$.
  - ***emBits:*** This value is one less than the length in bits of the RSA modulus *n*.
- **Parameters**
  - **padding$_1$:** hexadecimal string 00 00 00 00 00 00 00 00; that is, a string of 64 zero bits.
  - **padding$_2$:** hexadecimal string of 00 octets with a length (*emLen* - *sLen* - *hLen* - 2) octets, followed by the hexadecimal octet with value 01.

Subsequently, *EM* verification can be described as follows:

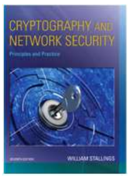
1. Generate the hash value of *M*: *mHash* = Hash(*M*)

2. If *emLen* < *hLen* + *sLen* + 2, output “inconsistent” and stop
3. If the rightmost octet of *EM* does not have hexadecimal value BC, output “inconsistent” and stop
4. Let *maskedDB* be the leftmost *emLen* - *hLen* - 1 octets of *EM*, and let *H* be the next *hLen* octets
5. If the leftmost 8 *emLen* - *emBits* bits of the leftmost octet in *maskedDB* are not all equal to zero, output “inconsistent” and stop
6. Calculate *dbMask* = MGF (*H*, *emLen* - *hLen* - 1)
7. Calculate *DB* = *maskedDB* $\oplus$ *dbMsk*
8. Set the leftmost 8 *emLen* - *emBits* bits of the leftmost octet in *DB* to zero
9. If the leftmost (*emLen* - *hLen* - *sLen* - 1) octets of *DB* are not equal to padding$_2$, output “inconsistent” and stop
10. Let *salt* be the last *sLen* octets of *DB*
11. Form block *M*′ = padding1 || *mHash* || *salt*
12. Generate the hash value of *M*′: H′ = Hash(*M*′)
13. If *H* = *H*′, output “consistent.” Otherwise, output “inconsistent”

The shaded boxes labeled *H* and *H*′ correspond, respectively, to the value contained in the decrypted signature and the value generated from the message *M* associated with the signature. The remaining three shaded areas contain values generated from the decrypted signature and compared to known constants. We can now see more clearly the different roles played by the constants and the pseudorandom value *salt*, all of which are embedded in the *EM* generated by the signer. The constants are known to the verifier, so that the computed constants can be compared to the known constants as an additional check that the signature is valid (in addition to comparing *H* and *H*′). The salt results in a different signature every time a given message is signed with the same private key. The verifier does not know the value of the salt and does not attempt a comparison.

Summary on digital signatures

- Protect two communicating parties from opponents as well as each other
  - Message content authentication
  - Non-repudiation of the source
- Based on public key encryption
  - NIST Digital Signature Algorithm (DSA): Based on Elgamal and SHA
  - Elliptic Curve Digital Signature Algorithm (ECDSA): Based on elliptic curves
  - RSA-PSS: Based on SHA and RSA



Link with the book
- Chapter 13 (Sections 13.1, 13.4, 13.5, 13.6)

University of Antwerp
Faculty of Science

End of Part 4

University of Antwerp
Faculty of Science