Secure Implementation of Cryptographic Algorithms

Chapter 2
Cryptographic Algorithms and their Implementation

Wieland Fischer & Berndt Gammel
Infineon Technologies

Content of Chapter 2

2. Cryptographic Algorithms and their Implementation

   2.1. Symmetric Algorithms
   - AES Definition
   - AES Implementation

   2.2. Asymmetric Algorithms
   - Introduction
   - RSA Definition
   - RSA Implementation

2.1. Symmetric Algorithms
AES Definition
AES Implementation

2.1. Symmetric Algorithms
AES Overview

- AES is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

  FIPS-197, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)". Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001.

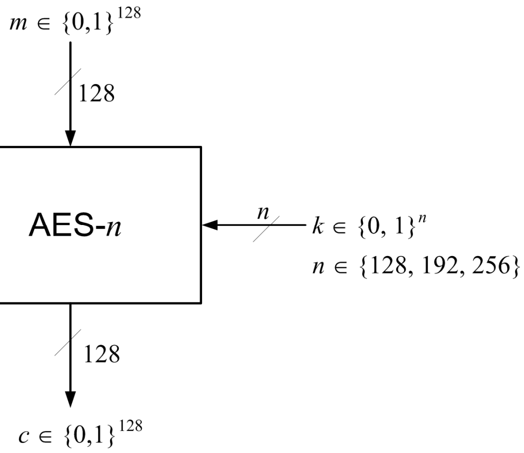
- AES was developed by *Joan Daemen* and *Vincent Rijmen* (Belgium) and initially named "*Rijndael*".
- AES is adopted world wide and is the most widely used symmetric cipher today. It superseedes the DES (1977).
- AES is a *block cipher* with 128-bit block size and 3 supported key lengths (128, 192, 256).
- AES was designed for efficiency in software and hardware.

2.1. Symmetric Algorithms
AES Definition

- AES: $\{0,1\}^{128} \times \{0,1\}^{n} \to \{0,1\}^{128}$

  $(\ m,\ \ k\ ) \to c = \mathrm{AES}_k(m)$ *Encryption*

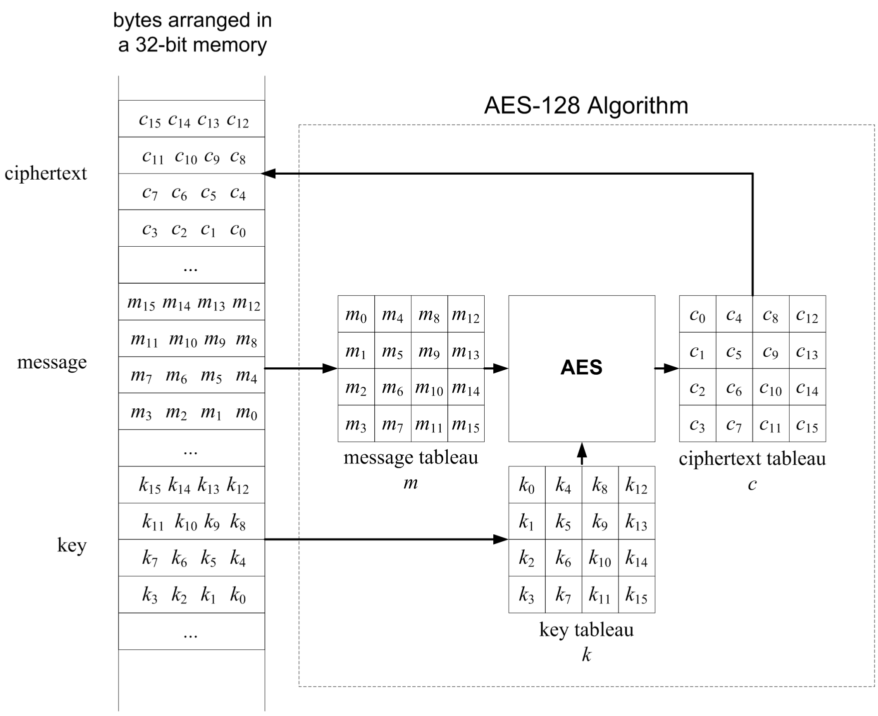
  $m = \mathrm{AES}_k^{-1}(c)$ *Decryption*



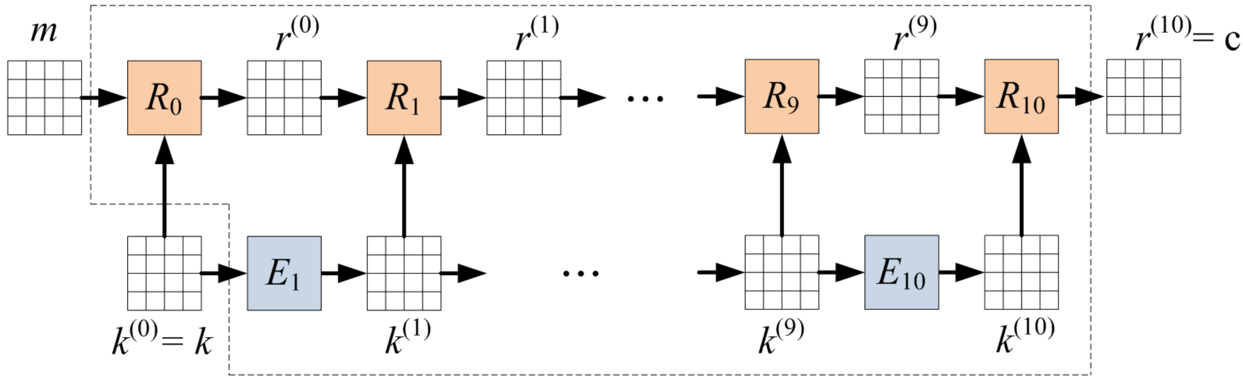
- Throughout this course we consider only key length n=128, i.e. *AES-128* and only the *encryption* direction.

2.1. Symmetric Algorithms
AES Data Structure

- All data are arranged in 4x4 byte tableaus



2.1. Symmetric Algorithms
AES Round Structure



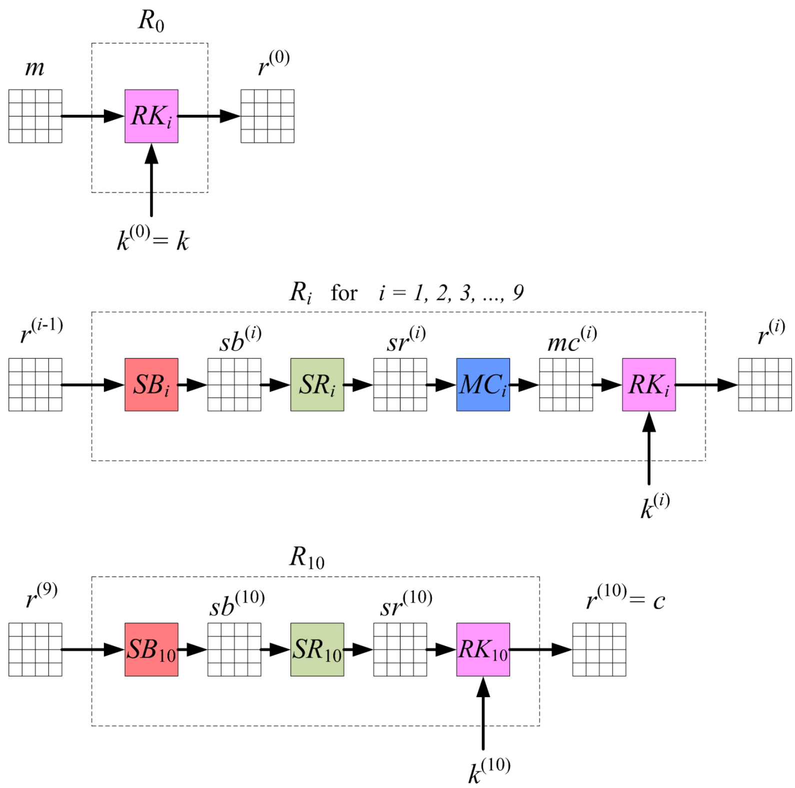
- $r^{(i)}$ is the *state* after ith round and is given by $r^{(i)} = R_i\left(r^{(i-1)}, k^{(i)}\right)$, $i = 0,1, 2, \ldots,10$. We define $r^{(-1)} = m$, $r^{(10)} = c$. R_i is called *round function*.
- $k^{(i)}$ is the *round key* of ith round and is given by $k^{(i)} = E_i\left(k^{(i-1)}\right)$, $i = 0,1, 2, \ldots,10$. We have $k^{(0)} = k$. E_i is called *key expansion function*.

2.1. Symmetric Algorithms
AES Round Functions

- Round Functions:
  - SB: SubBytes
  - SR: ShiftRows
  - MC: MixColumns
  - RK: AddRoundKey

- Naming conventions:

  R_0: "initial" round
  R_1: "first" round
  R_1-R_9: "normal" rounds
  R_9: "2nd last" round
  R_{10}: "last" round



2.1. Symmetric Algorithms
AES Round Function Definitions

- **RK: AddRoundKey (works on bits)**
  - Bitwise XOR of round key and previous state:

$$s_j^{(i)} = s_j^{(i-1)} \oplus k_j^{(i)} \quad \text{for} \quad j = 0, 1, 2, ..., 15$$

($s_j^{(i)}$ denotes the state, i.e. m, $mc_j^{(1)}$, ..., $mc_j^{(9)}$, $sr_j^{(10)}$)

- **SB: SubBytes (works on bytes)**
  - Substitutes each byte of the state by another byte value.
  - Background: The byte substitution is an inversion in GF(2^8) followed by an affine transformation and is called a substitution box "s-*box*" $S(x)$:

$$sb_j^{(i)} = S(r_j^{(i-1)}) \quad \text{for} \quad j = 0, 1, ..., 15, i = 1, 2, ..., 10$$

$$\text{with} \quad S(x) = \mathbf{A}\, x^{-1} + b.$$

  - In software $S(x)$ is usually implemented by table lookup (table length is 2^8 byte)

2.1. Symmetric Algorithms
AES Round Function Definitions

- **SR: ShiftRows (works on rows)**
  - Rotates the rows of the state to the left, with offsets 0, 1, 2, and 3, respectively.

$$\begin{bmatrix} sr_0^{(i)} & sr_4^{(i)} & sr_8^{(i)} & sr_{12}^{(i)} \\ sr_5^{(i)} & sr_9^{(i)} & sr_{13}^{(i)} & sr_1^{(i)} \\ sr_{10}^{(i)} & sr_{14}^{(i)} & sr_2^{(i)} & sr_6^{(i)} \\ sr_{15}^{(i)} & sr_3^{(i)} & sr_7^{(i)} & sr_{11}^{(i)} \end{bmatrix} = SR \begin{bmatrix} sb_0^{(i)} & sb_4^{(i)} & sb_8^{(i)} & sb_{12}^{(i)} \\ sb_1^{(i)} & sb_5^{(i)} & sb_9^{(i)} & sb_{13}^{(i)} \\ sb_2^{(i)} & sb_6^{(i)} & sb_{10}^{(i)} & sb_{14}^{(i)} \\ sb_3^{(i)} & sb_7^{(i)} & sb_{11}^{(i)} & sb_{15}^{(i)} \end{bmatrix}$$

2.1. Symmetric Algorithms
AES Round Function Definitions

- **MC: MixColumns (works on columns)**
  - GF(2^8) -linear transformation mixing each column of the state.
  - Most complex operation in AES software implementations.

The arithmetic on the state bytes s_j is performed in the Galois field

$$GF(2^8) := \mathbf{F}_2[x]/p(x)$$

with the reduction polynomial $p(x) = x^8 + x^4 + x^3 + x + 1$.

$$mc^{(i)} = MC(sr^{(i)}) = \begin{bmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{bmatrix} \cdot \begin{bmatrix} sr_0{}^{(i)} & sr_4{}^{(i)} & sr_8{}^{(i)} & sr_{12}{}^{(i)} \\ sr_1{}^{(i)} & sr_5{}^{(i)} & sr_9{}^{(i)} & sr_{13}{}^{(i)} \\ sr_2{}^{(i)} & sr_6{}^{(i)} & sr_{10}{}^{(i)} & sr_{14}{}^{(i)} \\ sr_3{}^{(i)} & sr_7{}^{(i)} & sr_{11}{}^{(i)} & sr_{15}{}^{(i)} \end{bmatrix}$$

2.1. Symmetric Algorithms
AES Round Function Definitions

- **MC: MixColumns (continued)**
  - Note, in the original specification a hexadecimal representation for the polynomials is used:

$$x = (10)_2 = (02)_{16}$$
$$x + 1 = (11)_2 = (03)_{16}$$
$$x^8 + x^4 + x^3 + x + 1 = (1\ 0001\ 1011)_2 = (11b)_{16}$$

  - This yields the following form for MC:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < Nb. \quad Nb = 4$$

[NIST01]

2.1. Symmetric Algorithms
AES Round Function Definitions

- **MC: MixColumns (continued)**
  - Definition of function *xtime*:

$$x \cdot s_j = (02)_{16} \cdot s_j = \mathrm{xtime}(s_j)$$

  - Multiplication by $x + 1 = (03)_{16}$

$$(x+1) \cdot s_j = \mathrm{xtime}(s_j) \oplus s_j$$

  - Efficient implementation in binary computer arithmetic:

```
x = s << 1;
if (s & 0x80)
  x ^= 0x1b;
```

- shift left
- test MSB
- reduce by p(x)

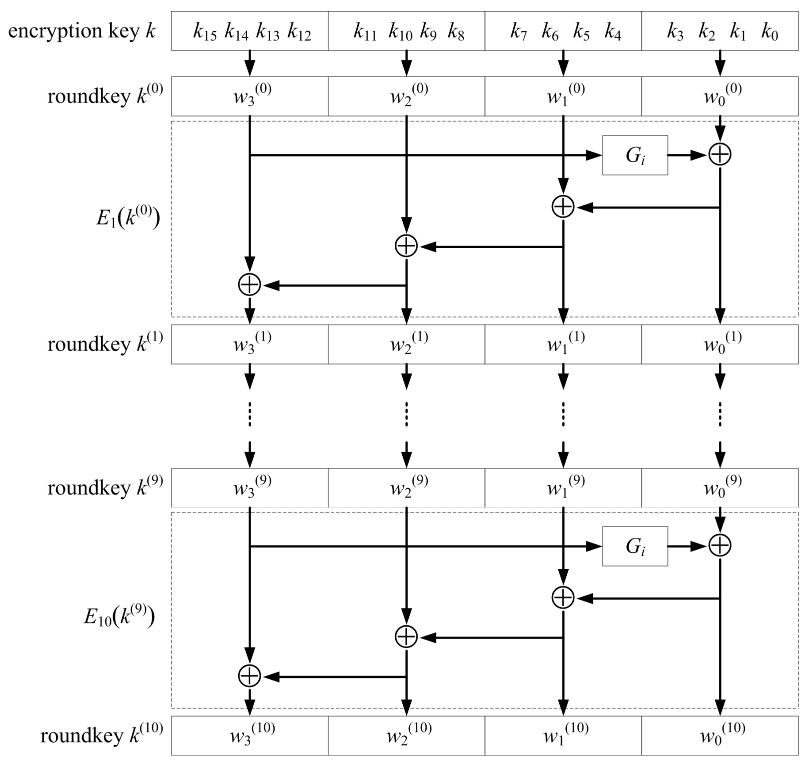
x, s are 8-bit unsigned integer variables

2.1. Symmetric Algorithms
AES Key Expansion

- The encryption key k is expanded to 11 round keys $k^{(i)}$. The expansion function is iteratively applied to k:

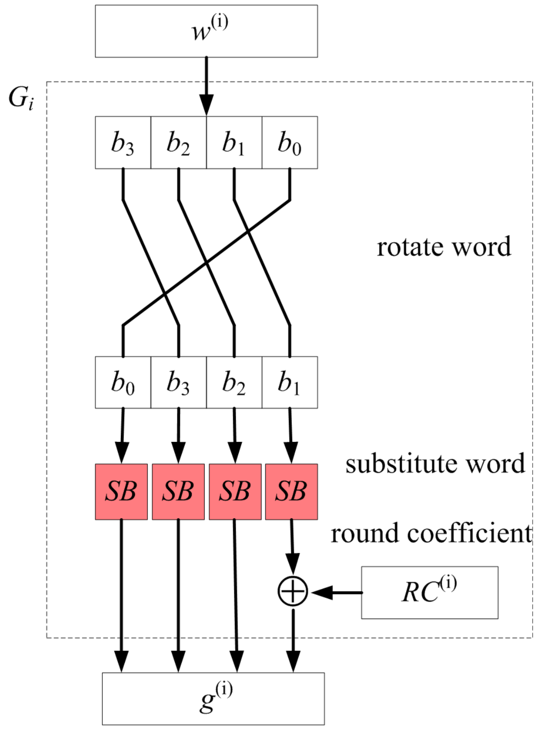
  $k^{(i)} = E_i(k^{(i-1)})$
  for $i = 1, 2, ..., 10$
  $k^{(0)} = k$

- The construction allows to update the key inplace in the same memory.



2.1. Symmetric Algorithms
AES Key Expansion

- Function G_i consists of
  - a rotation of the input word by one byte.
  - a bytewise substitution using the s-box $S(x)$.
  - the addition of a round coefficient $RC_i = x^i$. Note, the multiplications are in $\mathbf{F}_2[x]/p(x)$ and can be implemented by `xtime()`.
- Note, that knowing any round key (and the round number) is equivalent to knowing the encryption key.



2.1. Symmetric Algorithms
AES Implementations

- Optimized AES implementation for 32-bit processors (for details see [DR99, DR02])
- The basic idea is to combine the operations SubBytes, Shiftrows, and Mixcolumns to table lookup operations.
- There are four lookup tables T0, ..., T4. Each table maps an 8 bit input value to a 32 bit output. Definition:

$$T_0[a] = \begin{bmatrix} S[a]\bullet 02 \\ S[a] \\ S[a] \\ S[a]\bullet 03 \end{bmatrix} \quad T_1[a] = \begin{bmatrix} S[a]\bullet 03 \\ S[a]\bullet 02 \\ S[a] \\ S[a] \end{bmatrix} \quad T_2[a] = \begin{bmatrix} S[a] \\ S[a]\bullet 03 \\ S[a]\bullet 02 \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a]\bullet 03 \\ S[a]\bullet 02 \end{bmatrix}.$$

These are 4 tables with 256 4-byte word entries and make up for 4KByte of total space. Using these tables, the round transformation can be expressed as:

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-C1}] \oplus T_2[a_{2,j-C2}] \oplus T_3[a_{3,j-C3}] \oplus k_j.$$ [DR02]

- The tables T0, ..., T4 are used in all rounds except the final round. In the final round, standard AES s-box lookups have to be performed.

2.1. Symmetric Algorithms
AES Implementation - Summary

- AES is quite easy to implement in software.

- It essentially requires the following operations:
  - XOR (AddRoundKey, MixColumns)
  - Shift left (MixColumns)
  - Byte permutations (ShiftRows)
  - Table lookup (SubBytes, SubWord)

- Using 4Kb of memory, the implementation of AES can be done using T-tables. In this case only table lookups and XORs are necessary.

- For a more detailed description of the AES transformations and their mathematical background refer to [DR02, NIST01].

2.1. Symmetric Algorithms
References

[DR02] Joan Daemen and Vincent Rijmen: *The Design of Rijndael: AES - The Advanced Encryption Standard,* Springer Verlag 2002.

[DR99] Joan Daemen and Vincent Rijmen: *AES Proposal: Rijndael*, available online at http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[I10] Intel Corporation: *Intel® Advanced Encryption Standard (AES) Instructions Set*, available online at: http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set

[K09] Çetin Kaya Koç (Ed.), *Cryptographic Engineering*, Springer Verlag, 2009.

[NIST01] National Institute of Standards and Technology (NIST): *FIPS-197: Advanced Encryption Standard*, 2001.

[NIST99] National Institute of Standards and Technology (NIST): *FIPS-46-3: Data Encryption Standard*, 1999.

2.2. Asymmetric Cryptography

Introduction

RSA Definition

RSA Implementation

2.2. Asymmetric Cryptography Terminology

Asymmetric cryptography is the type of cryptography in which different keys are employed for the operations in the crypto system, and where one of the keys can be made public without compromising the secrecy of the other key[s].

It includes various methods:

- Public key encryption
- Digital signature schemes
- Key exchange schemes
- …
- Zero knowledge schemes
- Authentication schemes

2.2. Asymmetric Cryptography
Basic Principles

Asymmetric cryptography needs special techniques such as trapdoor-one-way functions. These are functions

$F = f_k$

that depend on parameters with two properties:

- F can be (publicly) described without the explicit usage of k.
- F can only be inverted, knowing the parameter k.

These functions are derived from difficult problems in number theory or other mathematical disciplines:

- Prime factorization of large integers,
- Computation of discrete logarithm in certain groups,
- Finding short vectors in high dimensional lattices,
- Coloring big graphs.

2.2. Asymmetric Cryptography
Public Key Encryption

Public key encryption is a method to encrypt messages using a non secret key.

Three parts are usually necessary

- Key generation: → public key puk
  → private key prk
- Encryption Algorithm: $E: \mathcal{K} \times \mathcal{M} \rightarrow C$
- Decryption Algorithm: $D: \mathcal{K} \times C \rightarrow \mathcal{M}$

such that

- $D_{\mathrm{prk}}(E_{\mathrm{puk}}(m)) = m$, for all $m \in \mathcal{M}$
- The knowledge of puk does not enable the computation of prk.

2.2. Asymmetric Cryptography
Digital Signature Schemes

Digital signature schemes are techniques to ensure an entity's acknowledgement of having sent a certain message.

Typically the entity has a private key and a corresponding public key. Usually four parts are necessary

- General settings in a common system
- Key generation: → public encryption key puk
  → private decryption key prk
- Signature generation Algorithm: $S: \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{S}$
  $s := S_{prk}(m)$, send $(s,m,[puk])$ to other entity
- Verification Algorithm: $V: \mathcal{K} \times \mathcal{S} \times \mathcal{M} \rightarrow \{0,1\}$
  if $1 = V_{puk}(s,m)$ then accept m, otherwise don't.

2.2. Asymmetric Cryptography
Key Exchange Schemes

Key exchange schemes are methods to securely establish/share a common secret between two or more entities, that then might be used as a secret key.

There are two types:

- Key transport protocols: One entity chooses a secret and securely distributes it to others.
- Key agreement protocols: All entities contribute to a common secret. Ideally the secret (or parts of it) is never transported via a channel between the entities.

2.2. Asymmetric Cryptography Interdependencies

Usually these classes are related in some cases:

- A public encryption system can be used as a digital signature scheme if encryption and decryption are commutative, i.e., if
  $E_{puk}(D_{prk}(m)) = D_{prk}(E_{puk}(m)) = m$, for all $m \in \mathcal{M}$.
  Then one can use
  $S_{prk}(m) := D_{prk}(m)$ and $V_{puk}(s,m) := (E_{puk}(s) == m)$.
- A public key encryption scheme can be used for key transport and agreement protocol.
  - One entity chooses a key and sends it public-encrypted to the other entity
  - Both entity do this and combine the two individual keys to one common (e.g. by hashing them)
- A key exchange protocol with a symmetric cipher might be usable for public key encryption and digital signature schemes.

2.2. Asymmetric Cryptography Most Prominent Examples

The most commonly used asymmetric cryptography systems are:

- Using the prime factorization problem:
  - RSA
  - Fiat-Shamir Identification protocol
- Discrete logarithm problem in $(\mathbf{Z}/n)^*$:
  - Digital signature algorithm, DSA
  - Diffie-Hellman key exchange, DH [DH76]
  - ElGamal Encryption scheme
- Discrete logarithm problem on elliptic curves
  - ECDSA, ECDH [Ko87, Mi86]
- Lattice problems (closest/shortest vector problem)
  - NTRU

RSA Introduction

- RSA was invented 1977 by R. Rivest, A. Shamir, and L. Adleman. [RSA78]

- The strength of RSA relies on the difficulty to factorize large integers.

From http://www.ams.org/samplings/feature-column/fcarc-internet

- It is still widely used and the most common public key algorithm for encryption and more for signature generation.

- State of the art key length is 1024 to 2048.

2.2. Asymmetric Cryptography
RSA Background I

The following points are necessary in order to formulate and understand RSA:

- Elementary Number Theory:
  - Primes
    - Integer p with exactly two divisors 1 and p. ($p=1$ is not a prime)
  - Greatest common divisor $\gcd(a,b)$ of two integers
    - Greatest positive integer g that divides a and b.
  - Relatively prime integers a, b: $\gcd(a,b)=1$
    - no non-trivial common divisor.

2.2. Asymmetric Cryptography
RSA Background II

- Let $a \in \mathbf{Z}$ and $n \in \mathbf{N}=\{1,2,3,4,\ldots\}$, define
  - $\boxed{a \text{ div } n}$: integer quotient of the division of a over n.
    $= [a/n]$ (real quotient, rounded to the next smaller integer)
  - $\boxed{a \bmod n}$: remainder of the integer division, in $\{0,1,\ldots,n-1\}$.

$$a - (a \text{ div } n) \cdot n = (a \bmod n)$$

- Examples:
  - $13 \text{ div } 5 = 2$
    $13 \bmod 5 = 3$
    $13 = 5 \cdot 2 + 3$
  - $-13 \bmod 5 = 2$
    $-13 \text{ div } 5 = -3$
    $-13 = 5 \cdot (-3) + 2$

2.2. Asymmetric Cryptography
RSA Background III

- Definition of $\mathbf{Z}/n\mathbf{Z}$:

  $$\mathbf{Z}/n\mathbf{Z} := \{(0 \bmod n\mathbf{Z}), (1 \bmod n\mathbf{Z}), \ldots, ((n\text{-}1) \bmod n\mathbf{Z})\},$$

  where $(i \bmod n\mathbf{Z})$ are just symbols; or shorter

  $$\mathbf{Z}/n\mathbf{Z} := \{(0 \bmod n), (1 \bmod n), \ldots, ((n\text{-}1) \bmod n)\},$$

  or even shorter, if one is lazy

  $$\boxed{\mathbf{Z}/n\mathbf{Z} := \{0, 1, \ldots, n\text{-}1\}\ .}$$

  - First of all this is just a set of n elements!
  - Define addition ($+$) and multiplication ($\cdot$) on it, by
    - $(a \bmod n\mathbf{Z}) + (b \bmod n\mathbf{Z}) = ((a+b) \bmod n) \bmod n\mathbf{Z}$
    - $(a \bmod n\mathbf{Z}) \cdot (b \bmod n\mathbf{Z}) = ((a \cdot b) \bmod n) \bmod n\mathbf{Z}$
    - The right hand side operations are normal operations in $\mathbf{Z}$, the left operations are something new! This makes $\mathbf{Z}/n\mathbf{Z}$ to a "ring".
  - Other notations are $\mathbf{Z}/n$, or even $\mathbf{Z}_n$ (bad notation!)

2.2. Asymmetric Cryptography
RSA Background IV

- Modular inversion in $\mathbf{Z}/n\mathbf{Z}$:
  - Since there is a multiplication in $\mathbf{Z}/n\mathbf{Z}$, one can ask for a multiplicative inverse: $(a \bmod n\mathbf{Z})^{-1}$, the inverse of a modulo n, is the element of $\mathbf{Z}/n\mathbf{Z}$ which fulfills

    $$(1 \bmod n\mathbf{Z}) = (a \bmod n\mathbf{Z}) \cdot (a \bmod n\mathbf{Z})^{-1}$$

  - Does not always exist. But if it exists, it is unique.

    <u>Theorem</u>: If $\gcd(a,n)=1$ then the inverse exists.

  - Often also written as $(a^{-1} \bmod n)$ or short a^{-1}, where a^{-1} stands for an integer in $\{0,\ldots,n\text{-}1\}$
- Notation: $a \equiv b \bmod n$, means

  $$a \bmod n = b \bmod n,$$

  or equivalently

  $(a\text{-}b)$ is a multiple of n.

- Then, if the inverse of a exists

  $$a \cdot a^{-1} \equiv 1 \bmod n$$

<u>Example:</u>

$12 \equiv 17 \bmod 5$
is correct. Also
$2 \equiv 17 \bmod 5$
and
$2 = 17 \bmod 5$
is correct. But
$12 = 17 \bmod 5$
is wrong.

2.2. Asymmetric Cryptography
RSA Background V

- <u>Fermat's (little) Theorem</u>:
  Is p a prime integer, then for any integer m we have
  $$m^p \equiv m \bmod p,$$
  or more generally
  $$m^{1+x\cdot(p-1)} \equiv m \bmod p,$$
  for any integer x.

- <u>Corollary 1</u>:
  if $m \bmod p \neq 0 \bmod p$ then
  $$m^{x\cdot(p-1)} \equiv 1 \bmod p,$$

- <u>Corollary 2</u>:
  In this case, the inverse of m modulo p is
  $$(m \bmod p\mathbf{Z})^{-1} = m^{(p-2)} \bmod p.$$

2.2. Asymmetric Cryptography
RSA Background VI

- <u>Chinese Remainder Theorem (CRT)</u>:
  - If the integers p, q are relatively prime then the map
    $$\mathbf{Z}/(p\cdot q) \rightarrow \mathbf{Z}/p \times \mathbf{Z}/q$$
    $$a \bmod p\cdot q \qquad (a \bmod p,\ a \bmod q)$$
    is an isomorphism of rings, i.e. bijective and respects addition and multiplication.
  - The inverse function is given by
    $$\mathbf{Z}/p \times \mathbf{Z}/q \rightarrow \mathbf{Z}/(p\cdot q)$$
    $$(a \bmod p,\ b \bmod q) \qquad (a\cdot u+b\cdot v \bmod p\cdot q),$$
    where u is the integer that corresponds to $(1 \bmod p, 0 \bmod q)$
    and v is the integer that corresponds to $(0 \bmod p, 1 \bmod q)$.
  - This is clear, since
    $$(a \bmod p,\ b \bmod q) = a \cdot (1,0) + b \cdot (0,1).$$
    Furthermore
    $u = q\cdot(q^{-1} \bmod p)$ and $v = p\cdot(p^{-1} \bmod q)$.
    Another inverse is given by Garner's formula (later…).

RSA Definition

- Generate 2 random primes: p and q.
- Compute $N := p \cdot q$
- choose a public key $e \in [0, \varphi(N)[$,
  with $\gcd(e,\varphi(N))=1$,
  where $\varphi(N):=(p-1)\cdot(q-1)$.

Non CRT format

- compute

$$d := e^{-1} \bmod \varphi(N)$$

- Public key: (e,N)
- Secret key: (d,N)

CRT format

- compute

$$d_p := e^{-1} \bmod (p-1)$$
$$d_q := e^{-1} \bmod (q-1)$$
$$q_{\mathrm{inv}} := q^{-1} \bmod p$$

- Public key: (e,N)
- Secret key: $(d_p,d_q,p,q, q_{\mathrm{inv}})$

2.2. Asymmetric Cryptography
RSA Signature Generation (Decryption)

Non CRT format

- Secret key: (d,N)
- Signature S of a message m:

$$S := S_d(m)$$
$$:= m^d \bmod N$$

- Decryption of a cipher text c:

$$D_d(c) := c^d \bmod N$$

CRT format

- Secret key: $(d_p,d_q,p,q,q_{\text{inv}})$
- Signature S of a message m:

$$S_p := m^{dp} \bmod p$$
$$S_q := m^{dq} \bmod q$$
$$S := S_q + q \cdot [(S_p - S_q) \cdot q_{\text{inv}} \bmod p]$$

(Garner's formula)

2.2. Asymmetric Cryptography
RSA Verification (Encryption)

- Public key: (e,N)
- Signature S for the message m is given.
  Verification of the signature S:
  - Compute
    $m' := S^e \bmod N$
  - check
    $m' = m$?
- Encryption of a message m:

$$D_e(m) := m^e \bmod N$$

2.2. Asymmetric Cryptography RSA Functional Principle

Why does this work, i.e., why is

(*) $\quad m^{e \cdot d} \equiv m \bmod N$?

This can be seen, using CRT: Since $d := e^{-1} \bmod (p\text{-}1)(q\text{-}1)$, i.e., $e \cdot d \equiv 1 \bmod (p\text{-}1) \cdot (q\text{-}1)$, one can write

$$e \cdot d = 1 + x \cdot (p\text{-}1) \cdot (q\text{-}1),$$

for some integer x. Since it is enough to prove the equality of (*) in $\mathbf{Z}/p$ and $\mathbf{Z}/q$, we only have to show

$$m^{e \cdot d} \equiv m \bmod p \;\text{ and }\; m^{e \cdot d} \equiv m \bmod q.$$

But this is clear due to Fermat's little theorem: E.g.

$$m^{e \cdot d} \equiv m^{1+(x \cdot (q\text{-}1)) \cdot (p\text{-}1)} \equiv m \bmod p.$$

2.2. Asymmetric Cryptography RSA Remarks I

- Complexity: In general, the approximate running time of
  - an addition is linear, i.e., $\approx \mathrm{const}_1 \cdot \mathrm{bl}$,
  - a (modular) multiplication is quadratic, i.e., $\approx \mathrm{const}_2 \cdot \mathrm{bl}^2$,
  - a modular exponentiation is cubic, i.e., $\approx \mathrm{const}_3 \cdot \mathrm{bl}^3$,

  in the bitlength bl of the input, e.g., the bitlength of the module.
- Why using RSA with CRT although it is more complicated?
  - The time for a 2048 bit RSA signature generation: $\approx \mathrm{const}_3 \cdot 2048^3$
  - The time for a 2048 bit RSA signature generation with CRT: $\approx 2 * \mathrm{const}_3 \cdot 1024^3$

  so RSA with CRT is about 4 times as fast, as a direct RSA implementation.

RSA Remarks II

- Small public key e is possible:
  - It is possible to use small public exponent e. There are no cryptographic attacks exploiting the size of e, if used properly.
  - Popular values are 3 (usage is a little bit tricky) and $2^{16}+1 (=: F_4)$.
  - Advantage: The running time for verification/encryption is now again quadratic ($\approx \mathrm{const}_4 \cdot \mathrm{bl}^2$).
- Small private key d is dangerous:
  - If
    $$d < N^{0.292},$$
    then the private key d can be recovered from the public key data.

RSA Implementation

RSA Implementation

2.2. Asymmetric Cryptography
Exponentiation

- The main ingredience of RSA is modular arithmetic:
  - Modular addition: $a + b \bmod N$
  - Modular multiplication: $a \cdot b \bmod N$
  - Modular exponentiation: $a^d \bmod N$
- Modular addition is easy to implement, since one needs only conventional long integer arithmetic:

```
c := a + b;
if c>=N then c:=c-N; end;
return c;
```

- Modular exponentiation is build up of modular multiplications. But since d usually is quite a big integer, $a^d \bmod N$ cannot be realized by $(d\text{-}1)$ times multiplying an integer with a. Other algorithms are used.

2.2. Asymmetric Cryptography
Right to Left Square&Multiply (rlSM)

One way to implement an exponentiation $m^d \bmod N$, or more generally m^d in any multiplicative group G, is the Square&Multiply algorithm, which comes in two flavors:

Let $d = (d_{n\text{-}1}\, d_{n\text{-}2} \dots d_1\, d_0)_2$ be the binary representation of d. Then

$$\boxed{d = d_{n\text{-}1}\cdot 2^{n\text{-}1} + d_{n\text{-}2}\cdot 2^{n\text{-}2} + \cdot\cdot\cdot + d_1\cdot 2^1 + d_0\cdot 2^0}.$$

Hence

$$m^d = (m^{2^{n\text{-}1}})^{d_{n\text{-}1}} \cdot (m^{2^{n\text{-}2}})^{d_{n\text{-}2}} \cdot \;\cdot\cdot\cdot\; \cdot (m^{2^1})^{d_1} \cdot (m^{2^0})^{d_0}\,.$$

So the rlSM method can be described by

```
Z := 1; M := m;
for i:=0 to n-1 do
      if d_i=1 then Z := Z · M;   //Now Z=(m^(2^i d_i))···(m^(2^0 d_0))
      M := M · M;                 //Now M= m^(2^(i+1))
end;
return Z;
```

The algorithm needs $(n+\mathrm{HW}(d))$ multiplications.

2.2. Asymmetric Cryptography
Left to Right Square&Multiply (lrSM)

The second flavor is the left to right method: Write

$$D_i := (d_{n-1}\, d_{n-2} \ldots d_{n-i})_2 \qquad [\&\ D_0 := 0]$$

Then the trick of the lrSM algorithm is the consecutive computation of $m^{D_1}, m^{D_2}, \ldots, m^{D_{n-1}}, m^{D_n} = m^d$ one after the other. We apply

$$D_i = 2 \cdot D_{i-1} + d_{n-i}$$

to the exponentiation,

$$m^{D_i} = m^{2D_{i-1}+d_{n-i}} = (m^{D_{i-1}})^2 \cdot m^{d_{n-i}}$$

So the lrSM algorithm can be written as:

```
Z := 1;                                  //Now Z = m^D0
for i:=n-1 to 0 do
      Z := Z · Z;                        //Now Z = m^2D(n-i-1)
      if d_i=1 then Z := Z · m;          //Now Z = m^D(n-i)
end;                                     //Now Z = m^Dn
return Z;
```

The algorithm needs $(n+\mathrm{HW}(d))$ multiplications.

2.2. Asymmetric Cryptography
Montgomery Ladder

The Montgomery ladder (originally introduced for elliptic curves [Mo87]) is another way to implement an exponentiation. It is done be computing step by step the pairs

$$(m^{D_1}, m^{D_1+1}), (m^{D_2}, m^{D_2+1}), \ldots, (m^{D_n}, m^{D_n+1}).$$

The step $i\text{-}1 \rightarrow i$ is done in the following way:

$$(m^{D_i}, m^{D_i+1}) := (m^{D_{i-1}+(D_{i-1}+d_{n-i})}, m^{(D_{i-1}+d_{n-i})+(D_{i-1}+1)})$$

Therefore the algorithm can be described by:

```
(A_0,A_1) := (1,m);
for i:=n-1 to 0 by -1 do
      (A_0, A_1) := (A_0 · A_di, A_di · A_1);
      // now (A_0, A_1) = (m^D(n-i), m^(D(n-i)+1))
end;
return A_0;
```

The algorithm needs $2 \cdot n$ multiplications, but the two of them within one loop can always be parallelized!

2.2. Asymmetric Cryptography
Modular Multiplication

- The remaining problem for the implementation of RSA in particular and of a modular exponentiation in general is the realization of the modular multiplication:

  $A \cdot B \bmod N$, (preferably) for $A,B \in [0,N\text{-}1[$.

- It is

  $A \cdot B \bmod N = A{\cdot}B - (A{\cdot}B \operatorname{div} N) \cdot N$,

  but an implementation of the integer division $(A{\cdot}B \operatorname{div} N)$ is usually *quite slow*.

- There are several ways to implement it efficiently:
  - Montgomery multiplication
  - Long integer multiplication with Barrett reduction
  - Special purpose hardware (with special algorithms)
  - …

2.2. Asymmetric Cryptography
Montgomery Multiplication I

- Montgomery [Mo85] solved the problem by defining a new operation, namely the Montgomery Multiplication:
  Let $N \in [2^{n\text{-}1},2^n[$ be an n-bit modulus and $Z{:=}2^n$, then define

  $A * B := (A \cdot B \cdot Z^{-1}) \bmod N$,

  to be the Montgomery multiplication (modulo N).

- With this operation, a normal modular multiplication can be implemented by

```
1. A’ := A · Z mod N;      // via A’=A*Z²
   B’ := B · Z mod N;      // via B’=B*Z²
2. C’ := A’ * B’ mod N;    // now C’=(AZ)(BZ)Z⁻¹ mod N
3. C  := C’ · Z⁻¹ mod N;   // via C=C’*1
return C;
```

  where the value $Z^2 := Z{\cdot}Z \bmod N$ has to be precomputed.

- Although this looks complicated: During an exponentiation, step1 is only necessary once at the beginning and 3. only at the end of an exponentiation.

2.2. Asymmetric Cryptography
Montgomery Multiplication II

- Now how can the Montgomery multiplication efficiently be computed on a normal processor? Montgomery presented a way, such that one only needs multi precision (non modular) multiplication:
- Set $N' := (-N \bmod Z)^{-1}$. Then $A*B$ can now be computed for $A,B \in [0,N[$ by:

```
C := A · B;                // C ∈ [0, N²[
D := C · N' mod Z;         // D ∈ [0, Z[
E := C + D · N;            // E ∈ [0, N²+ZN[ & E ≡ 0 mod Z
F := E div Z;              // F ∈ [0, N+N[
if F>=N then F:=F-N;       // extra reduction step
return F;
```

- Note that the computation of D is a modular multiplication modulo $Z=2^n$. There it is just a "half" non-modular multiplication. The division Step for F is also trivial, it is just shifting F by n bits.

2.2. Asymmetric Cryptography
Exponentiation with Montgomery Multiplication

- An exponentiation

  $S := M^d \bmod N$

  for $N \in [2^{n-1}, 2^n[$, $M \in [0,N[$ by using the Montgomery multiplication now looks the following way:
- Precomputation: $Z := 2^n$,
  $Z^2 := Z \cdot Z \bmod N$
  $N' := (-N \bmod Z)^{-1}$.
- Exponentiation: (notation see lrSM)

```
M' := M * Z²;                          // M' = M·Z mod N
X  := Z;                               // X = 1·Z mod N
for i:=n-1 to 0 by -1 do
       X := X * X;                     // X = m^(2D_{n-i-1})·Z mod N
       if d_i=1 then X := X * M';  end;// X = m^(D_{n-i})·Z mod N
end;                                   // X = m^d·Z mod N
S      := X * 1;                       // S = m^d·Z·1·Z^(-1) mod N
return S;
```

2.2. Asymmetric Cryptography References

[DH76] Whitfield Diffie and Martin E. Hellman: New Directions in Cryptography. IEEE Transactions on Information Theory, IT-22(6):644-654, 1976.

[Ko87] Neil Koblitz: *Elliptic Curve Cryptosystems*. Mathematics of Computation, 48:203-209, 1987.

[Mi86] Victor S. Miller: *Use of Elliptic Curves in Cryptography*. Advances in Cryptology - CRYPTO '85, Proceedings, 417-426, 1986.

[Mo85] Peter L. Montgomery: *Modular Multiplication without Trial Division*. Mathematics of Computation, 44:519-521, 1985.

[Mo87] Peter L. Montgomery: *Speeding the Pollard and elliptic curve methods of factorization*. Mathematics of Computation, 48(177):243-264, January 1987.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Commun. ACM, 21(2):120-126, 1978.