Implementation of RSA

F. De Santis
desantis@tum.de

Sichere Implementierung kryptographischer Verfahren WS 2015-2016
Lehrstuhl für Sicherheit in der Informationstechnik
Technische Universität München

28.10.2015

Outline

Section 1

RSA Implementation

RSA

- RSA is one of the most popular public-key cryptosystems, invented by R. Rivest, A. Shamir and L. Adleman in 1977 at MIT

- RSA is based on intractability of the integer factorization problem: find p or q from $n = pq$

- State of the art key size for RSA is 2048-bits for supposed equivalent 112-bit symmetric key security

- RSA can be used for:
  - ➥ Public-key encryption
  - ➥ Key exchange schemes
  - ➥ Digital signature schemes
  - ➥ ...

Background

- Let $\mathbb{Z} = \{\dots, -2, -1, 0, 1, -2, \dots\}$ and $\mathbb{Z}^+ = \{1, 2, 3 \dots\}$
- For two integers a and b, the greatest common divisor of a and b, denoted by $\gcd(a, b)$, is the largest positive integer that divides a and b.
- An integer $p \geq 2$ is *prime* if its only divisors are 1 and p.
- Two integers a, b are *relatively prime* or *coprime* if $\gcd(a, b) = 1$.

- Let $\mod : \mathbb{Z} \times \mathbb{Z}^+ \rightarrow \mathbb{Z}_n$ denote the reminder r of the integer division a/n:

$$\mod(a, n) = a \mod n = \begin{cases} r & \text{if } a \geq 0 \\ n - r & \text{otherwise} \end{cases}$$

- For an integer n, let $\mathbb{Z}_n = \{0, \dots, n-1\}$, $\mathbb{Z}_n^* = \{a : 1 \leq a < n, \gcd(a, n) = 1\}$ and "$\Phi(n) = \#\mathbb{Z}_n^*$".

RSA Signature Scheme

RSA Key Generation

- Given two primes p, q compute $n = pq$ and $\Phi(n) = (p-1)(q-1)$.
- Public key $k_{pub} = (n, e)$ s.t. $e \xleftarrow{\$}]1, \Phi(n)[$ and $\gcd(e, \Phi(n)) = 1$
- Private key $k_{pr} = (n, d)$ s.t. $ed = 1 \mod \Phi(n)$

RSA Signature Generation and Verification

Signing Entity $\mathcal{A}$		Verifying Entity $\mathcal{B}$
$y = \text{sig}^{\mathcal{A}}_{k_{pr}}(x) = e_{k_{pr}}(x) = x^d \bmod n$		
	$\xRightarrow{\text{send}(x,y)}$	
		$\text{ver}^{\mathcal{B}}_{k_{pub}}(x, y) = \begin{cases} \text{accept if} & d_{k_{pub}}(y) = y^e \bmod n = x \\ \text{reject if} & d_{k_{pub}}(y) = y^e \bmod n \neq x \end{cases}$
	$\xLeftarrow{\text{accept or reject}}$	

Implementation: Overview

1. Generate prime numbers p and q:
   - ➥ Sieve algorithms: Sieve of Eratosthenes, ...
   - ➥ Primality testing algorithms: Miller-Rabin primality test, ...
2. Compute greatest common divisor $\gcd(e, \Phi(n))$
   - ➥ Euclidean Algorithm
3. Compute modular inverse $d = e^{-1} \bmod \Phi(n)$.
   - ➥ Extended Euclidean Algorithm
4. Compute modular exponentiation: $x^d \bmod n$
   - ➥ Naïve
   - ➥ Binary Exponentiation Algorithms: Square and Multiply
5. Compute modular multiplication: $xy \bmod n$
   - ➥ Naïve
   - ➥ Montgomery Multiplication

Naïve Modular Exponentiation

- Modular exponentiation: $x^d \mod n$

Algorithm 1 Naïve Modular Exponentiation

Input: $x, d \in \mathbb{Z}, n \in \mathbb{Z}^+$
Output: $y = x^d \mod n$
1: $y \leftarrow \underbrace{x \cdot x \cdot x \cdot \dots \cdot x}_{d-1 \text{ multiplications}} \mod n$
2: **return** y

- Too costly in terms of time: $d - 1$ multiplications
- Too costly in terms of space: huge memory requirements if the reduction modulo n is done at the end

Modular Exponentiation with Binary Exponentiation

Natural binary representation of the exponent $d = \sum_{i=0}^{k-1} d_i 2^i$:

$$x^d \mod n = x^{\sum_{i=0}^{k-1} d_i 2^i} \mod n = \prod_{i:d_i=1} \left(x^{2^i} \mod n\right) \mod n$$

- Key observation: $x^{2^i} = x^{2 \cdot 2^{i-1}} = \left(x^{2^{i-1}}\right)^2$
- Consequence: $x^d \mod n$ be computed by successive squaring operations and conditional multiplications depending on whether the key bit d_i is set.

Modular Exponentiation with Binary Exponentiation

Algorithm 2 Left-to-right Binary Exponentiation Algorithm

Input: $x, n, d = (d_{k-1}, d_{k-2}, \ldots, d_1, d_0)_2$
Output: $y = x^d \pmod{n}$

```
y ← 1
for i = k − 1 downto 0 do          (scan through the key bits d_i from MSB to LSB)
    y ← y^2 (mod n)                (do the squaring always)
    if d_i = 1 then                (if the key bit d_i is set)
        y ← xy (mod n)             (do the conditional multiplication on the key bit d_i)
    end if
end for
return y
```

- k modular squarings $y^2 \bmod n$
- $HW(d)$ modular multiplications $xy \bmod n$

$\Rightarrow k + HW(d)$ modular multiplications

Modular Exponentiation with Binary Exponentiation

Let consider $k = 4$ and $d = 9 = (1,0,0,1)_2$: $y = x^4 \bmod n$

$$d = d_3 2^3 + d_2 2^2 + d_1 2 + d_0 = (d_3 2 + d_2)2^2 + d_1 2 + d_0 = \underbrace{(\underbrace{(\underbrace{(\underbrace{(d_3)}_{D_1}\, 2 + d_2)}_{D_2} 2 + d_1)}_{D_3} 2 + d_0}_{D_4}$$

i	d_i	$D_{k-i} = 2D_{k-i-1} + d_i$	y
3	1	$D_1 = 2D_0 + d_3$	$1^2 \cdot y = x$
2	0	$D_2 = 2D_1 + d_2$	$(x)^2$
1	0	$D_3 = 2D_2 + d_1$	$(x^2)^2 = y^4$
0	1	$D_4 = 2D_3 + d_0$	$(x^4)^2 \cdot x = x^9$

Naïve Modular Multiplication

- Modular multiplication: $ab \mod n$

Algorithm 3 Naïve Modular Multiplication

Input: $a, b \in \mathbb{Z}, n \in \mathbb{Z}^+$
Output: $ab \mod n$
1: $c \leftarrow ab - n\lfloor \frac{ab}{n} \rfloor$
2: **return** c

- The naïve implementation requires an integer division.
- Integer division is not typically available on constrained devices.

Montgomery Multiplication

The Montgomery multiplication *MM* is defined as:

$$MM(a, b, n, z) = abz^{-1} \mod n$$

Algorithm 4 Montgomery Multiplication

Input: $a, b \in \mathbb{Z}, n \in \mathbb{Z}^+, z = 2^k$
Output: $abz^{-1} \mod n$

$n' \leftarrow (-n)^{-1} \mod z$	(precomputation)
$c \leftarrow ab$	(integer multiplication)
$d \leftarrow cn' \mod z$	(truncation to k bits being $z = 2^k$)
$e \leftarrow c + nd$	(integer multiplication and addition)
$f \leftarrow e/z$	(right shift by k bits being $z = 2^k$)
if $f \geq n$ **then**	
$f \leftarrow f - n$	(integer substraction)
end if	
return f	

- No integer division required

Modular Multiplication using the Montgomery Multiplication

Algorithm 5 Modular Multiplication using the *MM*

Input: $a, b \in \mathbb{Z}, n \in \mathbb{Z}^+, z = 2^k$
Output: $c = ab \mod n$

1: $z \leftarrow 2^k$ (precomputation)
2: $z^2 \leftarrow zz \mod n$ (precomputation)
($\rightarrow$ conversion to the Montgomery domain)
3: $a' \leftarrow MM(a, z^2)$ ($a' = az \mod n$)
4: $b' \leftarrow MM(b, z^2)$ ($b' = bz \mod n$)
(multiplication in the Montgomery domain)
5: $c' \leftarrow MM(a', b')$ ($c' = abz \mod n$)
($\leftarrow$ conversion back to the integer domain)
6: $c \leftarrow MM(c', 1)$ ($c = ab \mod n$)
7: **return** c

- 4 *MM*s are required to compute the modular multiplication

Binary Exponetiation with Montgomery Multiplication

Putting things together:

Algorithm 6 Left-to-right Binary Exponentiation using the Montgomery Multiplication

Input: $y, n, d = (d_{k-1}, d_{k-2}, \ldots, k_1, k_0)_2$
Output: $x = y^d \pmod{n}$

$z \leftarrow 2^k$ (precomputation)
$z^2 \leftarrow zz \pmod{n}$ (precomputation)
$n' \leftarrow (-n)^{-1} \pmod{z}$ (precomputation)
$y' \leftarrow MM(y, z^2)$ (→ conversion to the Montgomery domain)
$x' \leftarrow z$ (→ conversion to the Montgomery domain)
for $i = k-1$ downto 0 **do** (scan through the key bits d_i from MSB to LSB)
    $x' \leftarrow MM(x', x')$ (do the squaring always)
    if $d_i = 1$ **then** (if the key bit d_i is set)
        $x' \leftarrow MM(x', y')$ (do the conditional multiplication)
    end if
end for
$x \leftarrow MM(x', 1)$ (← conversion back to the integer domain)
return x

Section 2

Task Description

Task Description

Implementation of the RSA signature generation in Python

- Left-to-right Exponentiation Algorithm
- Montgomery Multiplication
- Non-CRT Format
- **Integers size is 64-bit**

The implementation will be used to mount a timing attack in the next assignment

Section 3

Framework

Framework

- IDE: Ninja IDE (Windows, Linux, MacOS)
  - ➥ `http://ninja-ide.org/downloads/`
- Framework at `https://tueisec-sica.sec.ei.tum.de/`
  - ➥ `project.nja` is the project file for Ninja IDE
  - ➥ `student.py` must be used to implement the `RSA_Decrypt`
  - ➥ `main.py` has **NOT** to be modified
- Functions to be implemented:
  - ➥ `MontgomeryMul` to compute the Montgomery multiplication
  - ➥ `MontgomeryExp` to compute the left-to-right binary exponentiation using the Montgomery multiplication

Note: $n = pq$, d and the extended Euclidean algorithm are given (`modInvEuclid`).

Framework

```
import student, random, sys, time

n = long(0xB935E2B84B83E9EB)
d = long(0xDEADBEEF00211989)

print "n: ", "".join(student.hex64(n)), " (", student.bin64(n), ")"
print "d: ", "".join(student.hex64(d)), " (", student.bin64(d), ")"

start_time = time.clock()

for i in xrange(10):
    y = long(random.getrandbits(64))
    w = pow(y, d, n)
    x = student.RSA_Decrypt(y,d,n)

    print "\n"
    print "y: ", "".join(student.hex64(y)), " (", student.bin64(y), ")"
    print "x: ", "".join(student.hex64(x)), " (", student.bin64(x), ")"
    print "w: ", "".join(student.hex64(w)), " (", student.bin64(w), ")"

    if x != w:
        print "FAILED"
        sys.exit(100)

print "\nALL PASSED"
print "\nExecution time: %f s" % (time.clock()-start_time)
sys.exit(0)
```

Section 4

Submitting Results

Handing in Results

- Hand in `student.py` @ https://tueisec-sica.sec.ei.tum.de/handin/
  - ➥ Authentication using the LRZ Kennung required
- Multiple submissions are possible
  - ➥ Only the last submission is considered (files are overwritten)
- Deadline for submission fixed in 3 weeks
  - ➥ **25.11.15 23:59:59 CET**
- Evaluation process
  - ➥ The implementation will be tested automatically against freshly generated p, q, x, d integers

Rules

- The assignment is passed if the RSA signatures are correct for freshly generated inputs
- Implementing the RSA decryption using the left-to-right binary exponentiation and the Montgomery multiplication algorithm is **STRONGLY** suggested.

Stairway to Heaven

1. Download the framework
2. Implement the RSA signature generation in Python
3. Submit the source code **before 25.11.15 23:59:59 CET**