Studienarbeit

Fachbereich Informatik der TU-Darmstadt Fachgebiet Kryptographische Protokolle Dr. Tsuyoshi Takagi, Juniorprofessor



WS 2002/2003

Efficiency Comparison of Several RSA Variants

Camille Vuillaume

Contents

| Intr | oducti | on | 11 |
|------|---|--|---|
| 1.1 | Why i | s encryption important? | 11 |
| 1.2 | Why f | ast cryptography? | 12 |
| 1.3 | How t | o make cryptographic systems faster? | 12 |
| 1.4 | Goal o | of this paper | 12 |
| Bas | ic mat | hematic algorithms | 15 |
| 2.1 | Asym | ptotic behavior | 15 |
| | 2.1.1 | Analyzing an algorithm | 15 |
| | 2.1.2 | Defining the speed of an algorithm | 15 |
| | 2.1.3 | o-notation | 16 |
| 2.2 | Arithr | netic of integers | 16 |
| | 2.2.1 | Basic operations | 16 |
| | 2.2.2 | Radix representation | 16 |
| | 2.2.3 | Multiple-precision addition and subtraction $\ldots \ldots \ldots \ldots \ldots \ldots \ldots$ | 16 |
| | 2.2.4 | ${\rm Multiple-precision\ multiplication\ }\ldots$ | 18 |
| | 2.2.5 | Multiple-precision division | 18 |
| 2.3 | Modul | ar arithmetic | 18 |
| | 2.3.1 | Modular reduction | 18 |
| | 2.3.2 | Modular addition and subtraction $\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots$ | 19 |
| | 2.3.3 | Modular multiplication | 20 |
| | 2.3.4 | Inversion | 20 |
| 2.4 | Archit | ecture of smart cards | 22 |
| | 2.4.1 | General architecture | 22 |
| | 2.4.2 | Main components of a smart card $\hfill \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots$ | 22 |
| | 1.1 1.2 1.3 1.4 Bas 2.1 2.2 2.2 2.3 | 1.1 Why i 1.2 Why i 1.3 How t 1.4 Goal o Basic mat 2.1 Asymp 2.1.1 2.1.2 2.1.3 2.2 Arithr 2.2.1 2.2.2 2.2.3 2.2.4 2.2.4 2.2.5 2.3 Modul 2.3.1 2.3.2 2.3.3 2.3.4 2.4 Archit 2.4.1 | 1.2 Why fast cryptography? 1.3 How to make cryptographic systems faster? 1.4 Goal of this paper Basic mathematic algorithms 2.1 Asymptotic behavior 2.1.1 Analyzing an algorithm 2.1.2 Defining the speed of an algorithm 2.1.3 o-notation 2.1.4 Arithmetic of integers 2.2.2 Radix representation 2.2.3 Multiple-precision addition and subtraction 2.2.4 Multiple-precision multiplication 2.2.5 Multiple-precision division 2.3 Modular arithmetic 2.3.1 Modular nultiplication 2.3.2 Modular multiplication 2.3.4 Inversion 2.3.4 Inversion |

| | 2.5 | Analy | sis organization | 23 |
|---|----------------|---------|--|-----------|
| | | 2.5.1 | Speed estimation | 23 |
| | | 2.5.2 | Memory consumption | 23 |
| 3 | Clas | ssic RS | SA | 25 |
| | 3.1 | Basic | RSA | 25 |
| | | 3.1.1 | Key generation, encryption and decryption stages | 25 |
| | | 3.1.2 | Size of the exponents, security and speed | 26 |
| | 3.2 | Fast E | Exponentiation | 26 |
| | | 3.2.1 | Description of the left-to-right exponentiation algorithm | 26 |
| | | 3.2.2 | Correctness of the left-to-right exponentiation algorithm | 27 |
| | | 3.2.3 | Performance analysis of the left-to-right exponentiation algorithm | 28 |
| | 3.3 | Chines | se Remainder Theorem | 28 |
| | | 3.3.1 | Description of the Chinese remainder theorem | 28 |
| | | 3.3.2 | Correctness of the theorem | 29 |
| | | 3.3.3 | Performance analysis of the Chinese remainder theorem algorithm \ldots . | 29 |
| | | 3.3.4 | Special case: Chinese remainder theorem for $N = PQ$ | 30 |
| | 3.4 | Garne | r's algorithm | 31 |
| | | 3.4.1 | Description of Garner's algorithm | 31 |
| | | 3.4.2 | Correctness of Garner's algorithm | 31 |
| | | 3.4.3 | Performance analysis | 31 |
| | | 3.4.4 | Comparison with standard CRT algorithm | 32 |
| | 3.5 | Faster | implementation of RSA | 32 |
| | | 3.5.1 | Key generation, encryption and decryption stages | 32 |
| | | 3.5.2 | Description of the decryption stage | 33 |
| | | 3.5.3 | Correctness of the decryption algorithm | 33 |
| | | 3.5.4 | Performance analysis | 34 |
| | | 3.5.5 | Comparison with basic RSA | 35 |
| 4 | \mathbf{Reb} | alance | ed RSA | 37 |
| | 4.1 | | eneration, encryption and decryption stages | 37 |
| | | 4.1.1 | Key generation | 37 |
| | | 4.1.2 | Encryption | 39 |
| | | 4.1.3 | Decryption | 39 |
| | 4.2 | Perfor | mances of encryption and decryption stages | 39 |
| | | | | |

| 4 | .3 | Comp | arison with other variants of RSA | 40 |
|-----|-----|---------|--|----|
| 5 N | Mul | lti-Pri | me RSA | 41 |
| 5 | 5.1 | Garne | er's algorithm extension when $N = PQR$ | 41 |
| | | 5.1.1 | Description of Garner's algorithm | 41 |
| | | 5.1.2 | Correctness of Garner's algorithm | 42 |
| | | 5.1.3 | Performances of Garner's algorithm | 43 |
| 5 | 5.2 | Multi | Prime RSA modulo $N = PQR$ | 43 |
| | | 5.2.1 | Description of the Multi-Prime RSA cryptosystem | 43 |
| | | 5.2.2 | Description of the decryption stage | 44 |
| | | 5.2.3 | Performances of the Multi-Prime RSA decryption algorithm | 44 |
| | | 5.2.4 | Comparison with standard RSA decryption algorithms | 45 |
| 5 | 5.3 | Gener | al Garner's algorithm | 46 |
| | | 5.3.1 | Description of Garner's algorithm with $N = \prod_{i=1}^{b} P_i \dots \dots \dots \dots$ | 46 |
| | | 5.3.2 | Efficiency of Garner's algorithm | 47 |
| 5 | 6.4 | Multi | Prime RSA modulo $N = \prod_{i=1}^{b} P_i$ | 47 |
| | | 5.4.1 | Description of the RSA Multi-Prime cryptosystem | 47 |
| | | 5.4.2 | Description of the decryption stage | 48 |
| | | 5.4.3 | Performances of RSA Multi-Prime decryption algorithm | 48 |
| 6 N | Mul | lti-Pov | ver RSA | 51 |
| 6 | 5.1 | Hense | l lifting | 51 |
| | | 6.1.1 | Basic idea of Hensel lifting | 51 |
| | | 6.1.2 | Mathematical justification | 52 |
| | | 6.1.3 | Description of Hensel lifting | 52 |
| | | 6.1.4 | Performances of Hensel lifting | 53 |
| 6 | 5.2 | Impro | wing Hensel lifting | 54 |
| | | 6.2.1 | Inconvenient of the previous algorithm | 54 |
| | | 6.2.2 | Description of the improved Hensel lifting algorithm | 54 |
| | | 6.2.3 | Performance analysis of improved Hensel lifting | 55 |
| | | 6.2.4 | Comparison of the two versions | 55 |
| 6 | 5.3 | Multi | Power RSA modulo $N = P^2 Q$ | 56 |
| | | 6.3.1 | Algorithm | 56 |
| | | 6.3.2 | Description the decryption stage | 56 |
| | | 6.3.3 | Performances of the decryption stage | 57 |

5

CONTENTS

| | | 6.3.4 | Comparison with classic RSA | 58 |
|---|-----|--------|--|----|
| | 6.4 | Succes | ssive Hensel liftings | 58 |
| | | 6.4.1 | Main idea | 58 |
| | | 6.4.2 | Description of successive Hensel lifting | 59 |
| | | 6.4.3 | Performance analysis of successive Hensel liftings | 60 |
| | 6.5 | Multi- | Power RSA modulo $N = P^b Q$ | 61 |
| | | 6.5.1 | Algorithm | 61 |
| | | 6.5.2 | Description of the decryption stage | 61 |
| | | 6.5.3 | Performances of the decryption stage | 62 |
| 7 | Bat | ch RS. | Α | 65 |
| | 7.1 | Main i | idea | 65 |
| | | 7.1.1 | Batching two RSA decryptions | 65 |
| | | 7.1.2 | General case: batching p RSA decryptions | 66 |
| | | 7.1.3 | Correctness of Batch RSA | 66 |
| | | 7.1.4 | Performance analysis of batch RSA | 68 |
| | | 7.1.5 | Comparison with classic RSA | 69 |
| | 7.2 | How t | o increase efficiency? | 69 |
| | | 7.2.1 | Chinese remainder theorem | 70 |
| | | 7.2.2 | Computing multiple inverses | 70 |
| | | 7.2.3 | Computing multiple exponentiations | 70 |
| | 7.3 | Impro | ving batch RSA: Montgomery's trick | 70 |
| | | 7.3.1 | Description of Montgomery's Trick for multiple inversions $\ldots \ldots \ldots$ | 70 |
| | | 7.3.2 | Correctness of Montgomery's Trick | 71 |
| | | 7.3.3 | Performance analysis of Montgomery's Trick | 71 |
| | 7.4 | Impro | ving batch RSA: Shamir's Trick | 72 |
| | | 7.4.1 | Description of Shamir's Trick for multiple exponentiations | 72 |
| | | 7.4.2 | Explanations | 72 |
| | | 7.4.3 | Performance analysis of Shamir's Trick | 73 |
| | 7.5 | Batch | RSA using Shamir's and Montgomery's Tricks | 74 |
| | | 7.5.1 | Description of improved Batch RSA | 74 |
| | | 7.5.2 | Performance analysis of improved Batch RSA | 76 |
| | 7.6 | Batch | RSA using the Chinese remainder theorem | 77 |
| | | 7.6.1 | Description of Batch RSA using the Chinese remainder theorem $\ . \ . \ .$. | 77 |

CONTENTS

| | 7.7 | 7.6.2 Performances of Batch RSA using the Chinese remainder theorem Comparison with classic RSA | |
|---|-----|---|----|
| 8 | Cor | nclusion | 81 |
| | 8.1 | Speed comparisons | 81 |
| | 8.2 | Memory comparisons | 82 |

CONTENTS

List of Tables

| Algorithm 1: Multiple-precision addition | 17 |
|--|----|
| Algorithm 2: Multiple-precision subtraction | 17 |
| Algorithm 3: Multiple-precision multiplication | 18 |
| Algorithm 4: Multiple-precision division | 19 |
| Algorithm 5: Modular addition | 19 |
| Algorithm 6: Modular subtraction | 20 |
| Algorithm 7: Modular multiplication | 20 |
| Algorithm 8: Binary extended gcd algorithm | 21 |
| Algorithm 9: Modular inversion | 21 |
| Alar with a 10. I aft to wight any an artistical alar with a | 70 |
| | 27 |
| Algorithm 11: Chinese remainder theorem | 29 |
| Algorithm 12: Chinese remainder theorem with N=PQ | 30 |
| Algorithm 13: Garner's algorithm for CRT | 31 |
| Algorithm 14: RSA using CRT 3 | 33 |
| Algorithm 15: Rebalanced RSA: key generation | 38 |
| Algorithm 16: Garner's algorithm for $N = PQR$ | 42 |
| Algorithm 17: Multi-Prime RSA decryption with $N = PQR$ | 44 |
| Algorithm 18: Garner's algorithm pre-computations | 46 |
| Algorithm 19: Garner's algorithm with $N = \prod_{i=1}^{b} P_i$ | 46 |
| Algorithm 20: Multi-prime RSA decryption with $N = \prod_{i=1}^{b} P_i$ | 48 |
| Algorithm 21: Hensel Lifting 5 | 53 |
| 6 | 55 |
| Algorithm 23: RSA decryption modulo $N = P^2 Q$ | 57 |
| | |

| Algorithm 24: | Successive Hensel liftings | 60 |
|---------------|---|----|
| Algorithm 25: | RSA decryption modulo $N = P^b Q$ | 62 |
| Algorithm 26. | Batch RSA | 67 |
| - | | |
| Algorithm 27: | Montgomery's Trick for inversions | 71 |
| Algorithm 28: | Shamir's Trick for multiple exponentiations | 73 |
| Algorithm 29: | Improved Batch RSA | 75 |
| Algorithm 30: | Batch RSA using the Chinese remainder theorem | 77 |

Chapter 1

Introduction

Contents

| 1.1 | Why is encryption important? | 11 |
|-----|---|-----------|
| 1.2 | Why fast cryptography? | 12 |
| 1.3 | How to make cryptographic systems faster? | 12 |
| 1.4 | Goal of this paper | 12 |

1.1 Why is encryption important?

Nowadays, electronic communication is being widely used. There are many commercial applications that require security, but the distributed architecture of communication networks allows many attacks against private communications. In a first approximation, we can classify these attacks regarding four non-disjoint security categories.

- Authentification: an attacker can pretend to be someone else in order to get private information. Therefore, in a secure communication, the two parties may want to be sure of the identity of their interlocutor.
- **Integrity**: it is often wanted to check if a message has been modified by an unauthorized third party.
- **Privacy**: a message can contain some private information which must not be read by an unauthorized person.
- Non-repudiation: if we want a message to have a juridic value, for example to be an evidence in a trial, the author of the message must not be able to repudiate the content of this message.

Cryptography offers many algorithms and protocols to satisfy these criteria. For example, data encryption protects the privacy of many commercial transactions either on the Internet or on bank terminals. Cryptography can be seen as a tree whose roots are the mathematical algorithms, whose trunk is the concrete implementation of mathematical descriptions, whose branches are the protocols which use the implementations, and whose leaves are the cryptographic applications.

1.2 Why fast cryptography?

In the case of Internet banking, SSL servers are often overloaded with many simultaneous requests. Instead of developing the hardware, it is always possible to look for faster algorithms in order to speed up encryption and decryption stages. In the bank terminal case, smart cards only have light embedded systems and must be cheap but also secure. It is necessary to develop fast cryptographic algorithms in order to fit the poor calculation capabilities of such embedded systems. In general, it is always wanted to keep the hardware cheap and the system fast and secure. But there is an *a priori* contradiction between speed and security: the main security parameter of a cryptosystem is the length of the key. This key is used to encrypt and/or decrypt messages; the longer it is, the more secure is the communication. However, increasing the length of the key can drastically slow down encryption and decryption stages. That is why we have to investigate solutions allowing us to improve the security without slowing down the system or requiring expensive hardware. If we achieve the creation of a fast cryptosystem, we can use keys long enough to make the system resistant against the current attacks running on the fastest computers.

1.3 How to make cryptographic systems faster?

We can investigate three directions in order to speed up a cryptographic system.

- **Developing the hardware**: it is possible to embed cryptographic devices with fast arithmetic units or co-processors which consequently speed-up the cryptographic computations.
- **Optimizing the software**: the compilator can be optimized in order to speed-up the programs.
- Finding new mathematical solutions: the algorithms themselves can be modified in order to achieve a better computation speed.

While developing a cryptographic system, all of these three solutions are practically taken into account. But in this paper, we will only describe mathematical solutions enhancing algorithms which are already widely used.

1.4 Goal of this paper

In this paper, we will show a comprehensive description of fast algorithms based on the *de facto* standard RSA, namely:

- RSA using the Chinese remainder theorem: using a modulus like N = PQ, where P and Q are primes, it is possible to speed up the decryption stage.
- **Rebalanced RSA**: under some conditions, the secret key *d* can be chosen so that the decryption becomes much faster, at the expense of the encryption [Wie90].
- Multi-Prime RSA: this is the extension of RSA using the Chinese remainder theorem with a modulus N = PQR [CHLS97].

1.4. GOAL OF THIS PAPER

- Multi-Power RSA: this cryptosystem is currently being developed and offers interesting properties [Tak98].
- Batch RSA: batching several decryptions allows an overall speed-up, sparing a lot of resources [Fia89].

Chapter 2

Basic mathematic algorithms

Contents

| 2.1 | Asymptotic behavior | 15 |
|------------|-----------------------------|-----------|
| 2.2 | Arithmetic of integers | 16 |
| 2.3 | Modular arithmetic | 18 |
| 2.4 | Architecture of smart cards | 22 |
| 2.5 | Analysis organization | 23 |

2.1 Asymptotic behavior

2.1.1 Analyzing an algorithm

The main security parameter of current cryptographic algorithms is the key length. A big key will ensure a high level of security but the operations will also take a long time and consume a lot of memory. Currently, the recommended key length for the big standard cryptosystem RSA is 1024 bits. This key length is believed to be secure regarding the current computing abilities of our computers. But this length will probably be sooner or later too short. That is the reason why we should not analyze algorithms with a fixed key length; we rather evaluate speed and memory requirements depending on the key length, so that our results won't be out of date if the recommended key length becomes larger.

2.1.2 Defining the speed of an algorithm

We must choose the critical operation as criteria to estimate the speed of an algorithm: it could be for example the comparison operation (that is the case for sorting algorithms). But we are dealing with integer arithmetic and the critical operation is the multiplication. However, we cannot compare two multiplications of integers whose bit lengths are different; we will have to go deeper into the level of complexity and consider *atomic* operations of multiplications. These atomic operations are called **single-precision** operations. They will be our criteria to estimate the speed of cryptographic algorithms.

2.1.3 o-notation

We will often use the o-notation in order to evaluate the speed of a given algorithm. The mathematical definition of the o-notation is as follows:

$$f = o\left(g\right) \Leftrightarrow \frac{f(x)}{g(x)} \to 0, x \to +\infty$$

The number of single-precision operations, which depends on the bit length of the key, is typically a polynomial function: $P(n) = \sum_{i=0}^{k} a_i * n^i$. If the key length is long enough, the smallest degrees are negligible; we only need the greatest degrees, for example n^k and n^{k-1} . Then we can write:

$$P(n) = a_k * n^k + a_{k-1} * n^{k-1} + o(n^{k-1})$$

The o-notation shows us which terms have been neglected.

2.2 Arithmetic of integers

2.2.1 Basic operations

Among the operations of the instruction set of a processor, we have several basic operations at our disposal, like logical operations, reading or writing memory, evaluating conditions or jumping in the program. However, none of these allow to directly compute complex operations like multiplying two *n*-bit integers. Multiplication or division of such integers are called **multiple-precision** operations. The instruction set only provides bitwise arithmetic operations, which are called **single-precision** operations. However, we can design complex programs from all these basic instructions.

2.2.2 Radix representation

Positive integers can be represented in many ways [MOV97, p. 592]. The most common representation is called base 10: an integer is represented by series of digits from 0 to 9. For example, a = 123 means $a = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$. By extension, it is also possible to represent an integer using base b: $a = a_k * b^k + a_{k-1} * b^{k-1} + \ldots + a_0 * b^0$. Given an integer a and a basis b, this representation always exists and is unique. As computers only work with 0 and 1, base 2 (commonly called **binary representation**) is extensively used. We will use the following notation:

$$a = (111011)_2 = 2^5 + 2^4 + 2^3 + 0 \cdot 2^2 + 2^1 + 2^0$$

2.2.3 Multiple-precision addition and subtraction

Algorithms 1 and 2 perform additions [MOV97, p. 594] and subtractions [MOV97, p. 595] on two integers x and y having the same number of base b digits. If the integers have different lengths, the smaller of the two integers is padded with zeros on the left to achieve the same length.

The operations at step 2b and 2b are respectively called single-precision addition and subtraction. In order to perform a multiple-precision addition or subtraction on base b integers whose length is n, we respectively need n single-precision additions or subtractions.

Algorithm 1: Multiple-precision addition

UNIT: +; INPUT: two positive integers x and y represented in base b, having n base b digits; OUTPUT: $x + y = (w_n \dots w_0)_b$;

1. $c \leftarrow 0$; 2. for *i* from 0 to n-1 do (a) $w_i \leftarrow (x_i + y_i + c) \mod b$; (b) if $(x_i + y_i + c) < b$ then $c \leftarrow 0$; (c) else $c \leftarrow 1$; 3. $w_n \leftarrow c$; 4. return $((w_n \dots w_0)_b)$;

Algorithm 2: Multiple-precision subtraction

UNIT: -; INPUT: two positive integers x and y represented in base b, having n base b digits; OUTPUT: $x - y = (w_{n-1} \dots w_0)_b$;

1. $c \leftarrow 0;$

- 2. for i from 0 to (n-1) do
 - (a) $w_i \leftarrow (x_i + y_i + c) \mod b;$
 - (b) if $(x_i y_i + c) \leq 0$ then $c \leftarrow 0$;

(c) else $c \leftarrow -1$;

3. **return** $((w_{n-1} \dots w_0)_b);$

2.2.4 Multiple-precision multiplication

As input we have two positive integers $x = (x_{n-1} \dots x_0)_b$ and $y = (y_{t-1} \dots y_0)_b$ represented in base b. The product x * y has at most (n + t) base b digits. Algorithm 3 is a direct implementation of the standard pencil-and-paper method. See [MOV97, p. 595] for further information.

Algorithm 3: Multiple-precision multiplication

UNIT: *; INPUT: two positive integers x and y represented in base b, having respectively n and t base b digits; OUTPUT: $x * y = (w_{n+t-1} \dots w_0)_b$;

```
1. for i from 0 to (n + t - 1) do w_i \leftarrow 0;

2. for i from 0 to (t - 1) do

(a) c \leftarrow 0;

(b) for j from 0 to (n - 1) do

i. (uv)_b \leftarrow w_{i+j} + x_j * y_i + c;

ii. w_{i+j} \leftarrow v;

iii. c \leftarrow u;

(c) w_{i+n} \leftarrow u;

3. return((w_{n+t-1} \dots w_0)_b);
```

If x_j and y_i are two base *b* digits, the product $x_j * y_i$ is called single-precision multiplication and its result can be written as $(uv)_b$. Algorithm 3 performs n * t single-precision multiplications [MOV97, p. 596]. But what we are interested in is modular multiplication, because most of cryptosystems are based on finite groups arithmetic.

2.2.5 Multiple-precision division

In order to introduce modular arithmetic, we must be able to perform multiple-precision divisions. The division algorithm 4 takes as input two integers $x = (x_{n-1} \dots x_0)_b$ and $y = (y_{t-1} \dots y_0)_b$ where $n \ge t \ge 1$ and $y_{t-1} \ne 0$, then it computes the quotient $q = (q_{n-t} \dots q_0)_b$ and the remainder $r = (r_{t-1} \dots r_0)_b$ such that x = q * y + r and $0 \le r < y$ (see [MOV97, p. 598]).

Algorithm 4 requires about (n - t)(t + 2) single-precision multiplications (see [MOV97, p. 599]).

2.3 Modular arithmetic

2.3.1 Modular reduction

Most of cryptosystems are based on finite group arithmetic; for example $\mathbb{Z}/N\mathbb{Z}$. Each element x of this group can be seen as the representatives of the equivalent class $\{x \equiv y : y = x + k * N, k \in \mathbb{Z}\}$. We typically choose the representatives in the set $\{0, \ldots, N-1\}$. Then we can define the modulo operator as follows:

$$x = r \mod N \Leftrightarrow \exists q \in \mathbb{Z}, x = r + q * N$$

18

Algorithm 4: Multiple-precision division

UNIT: /; INPUT: $x = (x_{n-1} \dots x_0)_b$ and $y = (y_{t-1} \dots y_0)_b$, having $n \ge t \ge 1$ and $y_{t-1} \ne 0$; OUTPUT: $q = (q_{n-t} \dots q_0)_2$ and $r = (r_{t-1} \dots r_0)_2$ verifying x = q * y + r and $0 \le r < y$; 1. for *i* from 0 to (n-t) do $q_i \leftarrow 0$; 2. while $(x \ge y * b^{n-t})$ do $q_{n-t} \leftarrow q_{n-t} + 1$; $x \leftarrow x - y * b^{n-t}$; 3. for *i* from (n-1) down to *t* do (a) if $(x_i = y_{t-1})$ then $q_{i-t} \leftarrow b - 1$; (b) else $q_{i-t} \leftarrow \lfloor (x_i * b + x_{i-1})/y_{t-1} \rfloor$; (c) while $(q_{i-t}(y_{t-1} * b + y_{t-2} > x_i * b^2 + x_{i-1} * b + x_{i-2})$ do $q_{i-t} \leftarrow q_{i-t} - 1$; (d) $x \leftarrow x - q_{i-t} * y * b^{i-t}$; (e) if (x < 0) then $x \leftarrow x + y * b^{i-t}$; $q_{i-t} \leftarrow q_{i-t} - 1$; 4. $r \leftarrow x$; 5. return(q, r);

Given an integer x, if we want to find its representative, we just have to compute the remainder of the division x/N. This remainder verifies $x = q * N + r, r \in \{0, ..., N - 1\}$; it also means that $x = r \mod N$

2.3.2 Modular addition and subtraction

Let $x = (x_{n-1} \dots x_0)_b$ and $y = (y_{n-1} \dots y_0)_b$ be to integers verifying x < N and y < N. Then x + y < 2N; if $x + y \ge N$ the result of the modular addition modulo N will be x + y - N computed with the multiple-precision addition algorithm. Otherwise the result is simply x + y computed with the multiple-precision addition algorithm [MOV97, p. 600].

Algorithm 5: Modular addition

UNIT: $+ \mod N$; INPUT: x, y and N; OUTPUT: $z = x + y \mod N$;

compute z = x + y using the multiple-precision addition
 if (z ≤ N) then z ← z - N
 return(z)

Having the condition $x \ge y$, x - y is always smaller than N, the modular subtraction can be computed directly with the multiple-precision subtraction; if x < y then the result of the modular subtraction is x + N - y using the multiple-precision subtraction [MOV97, p. 600].

```
Algorithm 6: Modular subtraction
```

UNIT: $- \mod N$; INPUT: x, y and N; OUTPUT: $z = x - y \mod N$;

1. if $(x \le y)$ then $z \leftarrow x - y$ using the multiple-precision subtraction 2. if (x < y) then $z \leftarrow x + N - y$ using the multiple-precision subtraction 3. return(z)

2.3.3 Modular multiplication

Instead of simply computing x * y, we need the remainder of the division of x * y by N because we work in the finite group $\mathbb{Z}/N\mathbb{Z}$. As input we have two binary integers x and y and a modulus N and we want to compute $x * y \mod N$ [MOV97, p. 600].

| Algorithm | 7: | Modul | lar mu | ltip | lication |
|-----------|----|-------|--------|------|----------|
|-----------|----|-------|--------|------|----------|

UNIT: $* \mod N$; INPUT: x, y and N; OUTPUT: $x * y \mod N$;

1. compute x * y using the multiple-precision multiplication

2. compute r, remainder when x * y is divided by N

3. $\mathbf{return}(r)$

Assuming that the length of x and y is n, the multiple-precision multiplication x * y requires n^2 single-precision multiplications whereas the multiple-precision division requires n(n+2) single-precision multiplications. We have a total of $2n^2 + 2n$ single-precision multiplications for the whole modular multiplication algorithm.

2.3.4 Inversion

Inversion can be computed thanks to extended Euclide algorithm [MOV97, p. 608], which not only computes the greatest common divisor of two integers x and y, but also the two unique integers a and b such that gcd(x, y) = a * x + b * y.

Inversion is then simply computed with extended Euclidean algorithm as follows: if we want to find the inverse of x modulo N (assuming that gcd(x, N) = 1) then we compute $\operatorname{XGCD}(x, N)$ and get 1 = a * x + b * N. Modulo N this expression becomes $a * x = 1 \mod N$; in other words $a = x^{-1} \mod N$.

Algorithm 8: Binary extended gcd algorithm

UNIT: XGCD; INPUT: x and y, positive integers; OUTPUT: gcd(x, y) and two integers a and b such that gcd(x, y) = a * x + b * y;

```
1. g \leftarrow 1;
2. while x and y are even do
     (a) x \leftarrow x/2;
     (b) y \leftarrow y/2;
     (c) g \leftarrow 2g;
3. u \leftarrow x; v \leftarrow y;
4. A \leftarrow 1; B \leftarrow 0; C \leftarrow 0; D \leftarrow 1;
5. while u is even do
     (a) u \leftarrow u/2;
     (b) if A = B = 0 \mod 2 then A \leftarrow A/2; B \leftarrow B/2;
     (c) else A \leftarrow (A+y)/2; B \leftarrow (B-x)/2;
6. while v is even do
     (a) v \leftarrow v/2;
     (b) if C = D = 0 \mod 2 then C \leftarrow C/2; D \leftarrow D/2;
     (c) else C \leftarrow (C+y)/2; D \leftarrow (D-x)/2;
7. if (u \ge v) then u \leftarrow u - v; A \leftarrow A - C; B \leftarrow B - D;
8. else v \leftarrow v - u; C \leftarrow C - A; D \leftarrow D - B;
9. if u = 0 return(C, D, g * v) else goto 5;
```

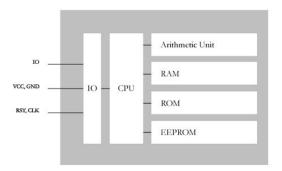
Algorithm 9: Modular inversion

UNIT: ${}^{-1} \mod N$; INPUT: $x \in \mathbb{Z}/n\mathbb{Z}$ and N; OUTPUT: $z = x^{-1} \mod N$; 1. compute a and b such that 1 = a * x + b * N using extended Euclidean algorithm 2. set $z \leftarrow a$ 3. return(z)

2.4 Architecture of smart cards

2.4.1 General architecture

The main application of efficient cryptography algorithms is the smart card technology, because computation and storage resources are limited. We have to be aware of the specific architecture of these systems in order to develop applications that will fit the resources and to predict how they will interact with the system.



2.4.2 Main components of a smart card

The previous picture shows us the different components that are to be found in a typical smart card architecture. Here is a more complete description of each component:

• Central Processing Unit:

The central processing unit (CPU) is a 8 or 16 bit controller. Typical CPU are for example Motorola 6805 or Intel 8051. The programming of CPU is done in assembler language.

• Read-Only Memory:

The read-only memory (ROM) is non-volatile memory; it is written once and permanently with a photographic mask. It contains the operating system (OS), the transmission protocol and commands, the security algorithms and several applications. Typical values for the read-only memory of smart cards are 8-16 kBytes, and at most 48 kBytes.

• Random Access Memory:

The random-access memory is a volatile memory, it means that all data get lost when the power supply is switched off. Therefore it is used as a buffer for storing transmission data and as a very fast access memory for workspace. Reading and writing a byte takes only a few microseconds. It is very fast but much more expensive than ROM. Typical values for its size are 128-256 Bytes and at most 3 kBytes.

• EEPROM:

The EEPROM is a non-volatile programmable memory; if the power supply is turned off, the data will not get lost and besides it allows about 100,000 update (*i.e.* erase/write) cycles. However writing into EEPROM is about 1,000 times slower than writing into RAM. This kind of memory is used in order to store the secret key and the cryptographic parameters. Typical EEPROM sizes are 2-8 kBytes, and at most 12 kBytes.

2.5. ANALYSIS ORGANIZATION

• Arithmetic Unit:

It is the cryptographic co-processor, which computes the integer arithmetic (addition, multiplication, modular exponentiation). The transfer speed between the AU and memory is not negligible. The development cost is very expensive.

2.5 Analysis organization

In the following, we will analyze some fast variants of RSA in terms of speed and memory consumption. First, we will write the algorithm using some basic operations like modular multiplications, modular additions, but also basic looping and conditional statements. Then we will eventually give mathematical justifications. At the end, we will estimate the speed and the memory requirements.

2.5.1 Speed estimation

Giving some hints about the estimation, we will typically write the estimation as a polynomial function depending on some parameters such as the bit length of some input values, using the o-notation described in §2.1.3. We will use the following costs for our estimations:

- Addition or subtraction of two n-bit integers: n
- Multiplication of two *n*-bit integers: n^2
- Division, where the bit length of the dividend is n and the bit length of the divisor is t: (n-t)*(t+2)
- Modular reduction of a t-bit integer modulo a n bit integer: equivalent to a division
- Modular addition or subtraction of two n-bit integers: 2n
- Modular multiplication of two *n*-bit integers: $2n^2 + 2n$
- Inversion of n-bit integers: equivalent to 20 modular multiplications

2.5.2 Memory consumption

We will differentiate **System parameters** and **Accumulators**. System parameters of high-level algorithms are usually stored in EEPROM memory as they are secret parameters, whereas system parameters of subroutines should not be taken into account for memory consumption estimation because they have already been defined in the calling routine. Accumulators are dynamically stored in RAM, and they strongly depend on the implementation. We will give here an upper bound of accumulators requirements, assuming that the accumulator memory is only freed when the subroutine ends (and not dynamically).

Chapter 3

Classic RSA

RSA is the standard asymmetric encryption algorithm. It was developed by Ron Rivest, Adi Shamir and Leonard Adleman in 1978 ([RSA78]). Since then, many companies adopted it, so that it is now a *de facto* cryptography standard. In this chapter, we will see the main properties of RSA. The security of RSA is based on factoring integers; currently composite integers longer than 1024 bits are secure against fast factoring algorithms. The exponentiation is the critical operation of RSA and it is important to develop a fast exponentiation algorithm, as shown in the following. Then we describe a fast variant using the Chinese remainder theorem, which can make an RSA decryption 4 times faster.

Contents

| 3.1 | Basic RSA | |
|------------|------------------------------|--|
| 3.2 | Fast Exponentiation | |
| 3.3 | Chinese Remainder Theorem 28 | |
| 3.4 | Garner's algorithm 31 | |
| 3.5 | Faster implementation of RSA | |

3.1 Basic RSA

In the following, we describe the algorithms of RSA cryptosystems, namely the key generation, encryption and decryption. The RSA permutation described in [RSA78] is the first candidate trapdoor function for cryptography, *i.e.* a function which is easy to compute but hard to invert, unless we know some *trapdoor* information.

3.1.1 Key generation, encryption and decryption stages

• Key generation:

The algorithm takes a security parameter n, which is the bit length of the modulus. Typically, n = 1024 bits is chosen. It next picks two primes P and Q whose bit length is n/2 and multiplies them to get the modulus N = PQ. It then pick some relatively small value e which is prime to $\phi(N) = (P-1)(Q-1)$. A typical value is e = 65537. The two integers

 $\langle N, e \rangle$ are the public key. To generate the secret key, the algorithm computes the inverse of $e \mod \phi(N)$: $d = e^{-1} \mod \phi(N)$.

• Encryption:

The message to be encrypted must be first converted to an integer $M \in \mathbb{Z}/N\mathbb{Z}$. There are standard algorithms to perform this operation, for example in the PKCS#1 standard. Having the plaintext M, the ciphertext is simply computed as $C = M^e \mod N$. Hence, the public key consists of $\langle N, e \rangle$.

• Decryption:

The secret key is simply the secret exponent d. Knowing d and the ciphertext C, we can have the plaintext back if we compute C^d modulo N:

$$C^d = M^{ed} \bmod N$$

We know that $ed = 1 \mod \phi(N)$, in other words, there is an integer k such that $ed = 1 + k * \phi(N)$.

$$C^{d} = M^{1+k*\phi(N)} \mod N$$
$$= M*(M^{\phi(N)})^{k} \mod N$$

Besides Fermat's little theorem shows us that $M^{\phi(N)} = 1 \mod N$. Finally:

 $C^d = M \bmod N$

3.1.2 Size of the exponents, security and speed

The encryption stage is relatively fast because e can be chosen to be small. It is not possible to choose d small otherwise RSA would not be secure ([BG00]), but it is often wanted to have a fast decryption or signature stage, for example in smart cards, because we cannot embed fast processors in cheap smart cards. Therefore, some faster decryption algorithms will be described in this paper.

3.2 Fast Exponentiation

3.2.1 Description of the left-to-right exponentiation algorithm

This fast exponentiation method (see also [MOV97, p. 615]) is based on the binary representation of the exponent and allows to compute an exponentiation with few multiplications and square computations. We take the basis M, the exponent e and the modulus N as input, and the result of the exponentiation $C = M^e \mod N$ as output. Besides, we have the binary representation of $e = \sum_{i=0}^{k-1} e_i * 2^i$, where e_i is either 0 or 1. We can now transform the exponent to the following expression:

$$M^{e} = M^{\sum_{i=0}^{k-1} e_{i} * 2^{i}}$$
$$= \prod_{i=0}^{k-1} M^{e_{i} * 2^{i}}$$

3.2. FAST EXPONENTIATION

Using this relationship, we recursively compute the result: first we can initialize it with 1. Then we go through the binary representation of the exponent. If the digit at position i is 1, we multiply the result with M^{2^i} (*i.e.* M squared i times). But it can be faster: we initialize the result with 1 and go through the binary representation of the exponent from the left (*i.e.* the most significant bit). At each step we square the result and multiply it with M if the current digit is 1. Doing this, if the digit at position i is 1, at the end we will have multiplied the result with M^{2^i} . We summarize the algorithm as follows:

Algorithm 10: Left-to-right exponentiation algorithm

UNIT: $x^y \mod N$; INPUT: $M, e = (e_{k-1} \dots e_0)_2, N$; OUTPUT: $C = M^e \mod N$;

C ← 1;
 for *i* from (k − 1) down to 0 do

 (a) C ← C * C mod N;
 (b) if (e_i = 1) then C ← C * M mod N;

 return(C);

3.2.2 Correctness of the left-to-right exponentiation algorithm

We will prove by induction that the algorithm works. Let's assume that at step i, we have $C_i = \prod_{j=i}^{k-1} M^{e_j * 2^{j-i}} \mod N$. At the next step, we must perform a square computation and a multiplication if $e_{i-1} = 1$. Therefore,

$$C_{i-1} = M^{e_{i-1}} * C_i^2 \mod N$$

= $M^{e_{i-1}} * \left(\prod_{j=i}^{k-1} M^{e_j * 2^{j-i}}\right)^2 \mod N$
= $M^{e_{i-1}} * \prod_{j=i}^{k-1} M^{e_j * 2^{j-i+1}} \mod N$
= $M^{e_{i-1}} * \prod_{j=i}^{k-1} M^{e_j * 2^{j+(i-1)}} \mod N$
= $\prod_{j=i-1}^{k-1} M^{e_j * 2^{j+(i-1)}} \mod N$

At the beginning of the loop, at step k - 1, the assumption is is true because $C = M^{e_{k-1}} \mod N$. By induction, the assumption is true until i = 0. And for i = 0, we get:

$$C = \prod_{j=0}^{k-1} M^{e_j * 2^j} \mod N$$

3.2.3 Performance analysis of the left-to-right exponentiation algorithm

• Notations:

k is the bit length of the exponent e and n the bit length of the modulus N

• Cost estimation:

At step *i* of the main loop, we perform one squaring, and one multiplication if the binary e_i digit of the exponent equals 1. This happens with the probability 1/2, therefore we have:

- -1/2 multiplication at each step of the loop on average (step 2b)
- 1 square computation at each step of the loop (step 2a), except at the beginning of the algorithm, where C = 1.

We have k iterations for the main loop and 3k/2 - 1 multiplications in total. Asymptotic behavior: $(3k-2)(n^2+n)$ and $3n^3 + n^2 + o(n^2)$ if k = n

• Memory cost:

| _ | - System parameters | | | | | | |
|---|---------------------|-----------|---------------------|--|--|--|--|
| | Register names | Bits | Number of registers | | | | |
| | C, M, N | n | 3 | | | | |
| | e | k | 1 | | | | |
| | Subtotal | 3n+k bits | | | | | |

- Total memory cost: 3n + k bits and 4n bits if k = n

3.3 Chinese Remainder Theorem

3.3.1 Description of the Chinese remainder theorem

The Chinese remainder theorem allow us to decrease the computation time of a classical RSA decryption. More generally, the Chinese remainder theorem establishes a bijection between $\mathbb{Z}/m\mathbb{Z}$ and the cartesian product

$$\prod_{i=0}^{k} \mathbb{Z}/m_i \mathbb{Z}$$

assuming that $m = \prod_{i=0}^{k} m_i$ and that the m_i don't have any common divider (see [Buc01, §2.15]). For example, there is an isomorphism between $\mathbb{Z}/n\mathbb{Z}$ and $\mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/q\mathbb{Z}$, assuming that p and q are primes and n = pq. In other words, we can compute in $\mathbb{Z}/p\mathbb{Z}$ and $\mathbb{Z}/q\mathbb{Z}$ instead of $\mathbb{Z}/n\mathbb{Z}$. Assuming that the bit length of both p and q is the half of the bit length of n, the cost of a multiplication can be theoretically divided by two. It is quite simple to compute the image of a given a, but if the system $a = a_i \mod m_i$ is given, the reciprocal computation of a is not trivial. The goal of the Chinese remainder theorem is to give us the unique solution of this system.

We want to solve the following system:

$$\begin{cases} a = a_1 \mod m_1 \\ \vdots \\ a = a_k \mod m_k \end{cases}$$

Hence we take the residues a_1, \ldots, a_k and the moduli m_1, \ldots, m_k as input and compute a, the solution of the system.

Algorithm 11: Chinese remainder theorem

UNIT: CRT; INPUT: $a_1, \ldots, a_k, m_1, \ldots, m_k$; OUTPUT: a verifying $a = a_i \mod m_i$ for i = 1 to k; 1. $a \leftarrow 0$; 2. $m \leftarrow m_1$; 3. for i from 2 to k do (a) $m \leftarrow m * m_i$; 4. for i from 1 to k do (a) $M_i \leftarrow m/m_i$; (b) $y_i \leftarrow M_i^{-1} \mod m_i$; (c) $G \leftarrow y_i * M_i \mod m$; (d) $G \leftarrow G * a_i \mod m$; (e) $a \leftarrow a + G \mod m$; 5. return(a);

3.3.2 Correctness of the theorem

We are looking for an a such that $a = a_i \mod m_i$ for i = 1 to k. Let's prove that $a = \sum_{i=0}^k a_i * M_i * y_i$ is a solution of the system.

 $M_i = \prod_{i=0, i \neq i}^k m_j$ and $M_j = 0 \mod m_i$ if $j \neq i$. That is why

$$a = a_i * M_i * y_i \mod m_i$$

But we have chosen y_i such that $y_i = M_i^{-1} \mod m_i$; in other words $y_i * M_i = 1 \mod m_i$. At last, we get for i = 1 to k:

$$a = a_i \mod m_i$$

3.3.3 Performance analysis of the Chinese remainder theorem algorithm

• Notations:

n is the bit length of $m = \prod_{i=0}^{k} m_i$ and we assume that the bit length of each of the m_i is n/k.

• Cost Estimation:

At step 3a, we compute $m = \prod_{i=0}^{k} m_i$ and we have to perform (k-1) multiplications in \mathbb{Z} of n/k-bit integers. In total it costs $(k-1)n^2/k^2$.

The division at step 4a is a simple division in \mathbb{Z} . The bit length of m and m_i are respectively n and n/k; therefore each division costs $(n^2 + 2kn)(k-1)/k^2$. We have in total k divisions and the cost of step 4a is $(n^2 + 2kn)(k-1)/k$.

At step 4b we compute each time an inversion modulo m_i . It is equivalent to twenty multiplications of n/k-bit integers and it is done k times: $40(n^2/k + n)$

At steps 4c and 4d we compute a multiplication of *n*-bit integers and it is done k time. In total, it costs $4k(n^2 + n)$.

Asymptotic behavior: $n^2 (4k + 1 + 40/k - 1/k^2) + o(n^2)$

- Memory Cost:
 - System parameters

| Register names | Bits Number of registers | | |
|----------------|--------------------------|---|--|
| | n/k | k | |
| m_i | n/k k | | |
| a | n | 1 | |
| Subtotal | 3n bits | | |

- Accumulators

| Register names | Bits | Number of registers |
|----------------|--------------------------|---------------------|
| m, G | n | 2 |
| M_i | n/k | k |
| y_i | $\mid n/k \mid \qquad k$ | |
| Subtotal | 4n bits | |

- Total memory cost: 7n bits

3.3.4 Special case: Chinese remainder theorem for N = PQ

This is a special case of the Chinese remainder theorem when the modulus is the product of two primes: N = PQ. We want to compute M such that $M = M_P \mod P$ and $M = M_Q \mod Q$.

| Algorithm 12: Chinese remainder theorem with $N=PQ$ |
|---|
| UNIT: CRT_PQ ; INPUT: M_P, M_Q, P, Q, N ; |
| $\frac{\text{Output: } M;}{1 \dots 2^{-1} \dots 1^{D}}$ |
| 1. $y \leftarrow Q^{-1} \mod P;$ 2. $M \leftarrow y * Q \mod N;$ 2. $M \leftarrow M \leftarrow M$ |
| 3. $M \leftarrow M * M_P \mod N;$ 4. $y \leftarrow P^{-1} \mod Q;$ 5. $Q \leftarrow P^{-1} \mod Q;$ |
| 5. $G \leftarrow y * P \mod N;$ 6. $G \leftarrow G * M_Q \mod N;$ 7. $M \leftarrow M + G$ |
| 7. $M \leftarrow M + G;$ 8. $\mathbf{return}(M);$ |

We have to compute two inverses of n/2-bit integers at steps 1 and 4 and four multiplications of *n*-bit integers at steps 2, 3, 5 and 6. Each inversion is equivalent to 20 multiplications of n/2bit integers; we finally get a complexity of $28n^2 + o(n^2)$ and a memory utilization of 4n (system parameters) + 3n/2 (accumulators).

30

3.4 Garner's algorithm

3.4.1 Description of Garner's algorithm

The classical utilization of the Chinese remainder theorem in cryptography only requires a modulus N product of two primes P and Q. It is possible to eliminate the computation of the two inverses thanks to Garner's algorithm. This enhancement costs however more memory for the system parameters. Like in the previous algorithm, we want to find M such that $M = M_P \mod P$ and $M = M_Q \mod Q$. We have the same input parameters except $(P_-inv_-Q) = P^{-1} \mod Q$ which is a specific precomputed parameter which allows us to avoid computing any inversion. See [MOV97, p. 612] for a more general description.

Algorithm 13: Garner's algorithm for CRT

UNIT: GARNER; INPUT: $M_P, M_Q, P, Q, (P_inv_Q), N;$ OUTPUT: M;

1. $V \leftarrow M_Q - M_P \mod Q;$ 2. $V \leftarrow V * (P_i n v_Q) \mod Q;$ 3. $M \leftarrow V * P \mod N;$ 4. $M \leftarrow M + M_P \mod N;$ 5. return(M);

3.4.2 Correctness of Garner's algorithm

We want to compute M such that $M = M_P \mod P$ and $M = M_Q \mod Q$. We have precomputed $(P_i inv_Q) = P^{-1} \mod Q$. Let's prove that

$$M = M_P + V * P \mod N$$

where $V = (P_{-}inv_{-}Q) * (M_Q - M_P) \mod Q$. Obviously $M = M_P \mod P$. And:

$$P * V = P * (P_-inv_-Q) * (M_Q - M_P) \mod Q$$
$$= M_Q - M_P \mod Q$$

Finally:

$$M = M_P + M_Q - M_P \mod Q$$
$$= M_Q \mod Q$$

3.4.3 Performance analysis

• Notations:

n is the bit length of N and we assume that the bit length of P and Q is n/2

• Cost Estimation:

We have 1 multiplication of n/2-bit integers for the computation of V at step 2 and 1 multiplication of n-bit integers for the computation of M at step 3. Asymptotic behavior: $5n^2/2 + o(n^2)$

• Memory Cost:

– System parameters

| System parameters | | | |
|-------------------|-----------|---------------------|--|
| Register names | Bits | Number of registers | |
| N, M | n | 2 | |
| M_P, M_Q, P, Q | n/2 | 4 | |
| (PinvQ) | n/2 | 1 | |
| Subtotal | 9n/2 bits | | |

- Accumulator

| Register names | Bits | Number of registers |
|----------------|----------|---------------------|
| V | n/2 | 1 |
| Subtotal | n/2 bits | |

– Total memory cost: 5n bits

3.4.4 Comparison with standard CRT algorithm

• <u>Cost Estimation:</u>

| | Speed | Estimated speed-up |
|---------------------------|-------------------|--------------------|
| Chinese remainder theorem | $28n^2 + o(n^2)$ | 1.0 |
| Garner's algorithm | $5n^2/2 + o(n^2)$ | 11.2 |

• Memory:

| | Total Memory | System Parameters | Accumulators |
|---------------------------|--------------|-------------------|--------------|
| Chinese remainder theorem | 11n/2 bits | 4n bits | 3n/2 bits |
| Garner's algorithm | 5n bits | 9n/2 bits | n/2 bits |

Compared to the classical Chinese remainder theorem, Garner's algorithm is much faster at the expense of system parameters memory. For a modulus of 1024 bits, we require 4608 bits instead of 4096 bits for system parameters, but it is about 11 times faster. Besides we don't need any modular inversion implementation, hence saving development costs and ROM memory.

3.5 Faster implementation of RSA

We show an optimization of RSA using the Chinese remainder theorem and the fast exponentiation algorithm. The two inverses that are normally needed for the Chinese remainder theorem are reduced in one inversion, which is precomputed. For a small loss in terms of memory, we manage to speed up the RSA decryption by four.

3.5.1 Key generation, encryption and decryption stages

• Key generation:

The key generation is the same as in standard RSA: we have as input a security parameter n,

32

3.5. FASTER IMPLEMENTATION OF RSA

we choose two primes whose bit length is n/2 and multiply them to get the modulus N. Then we pick some integer e such that $\phi(N)$ and e are relatively prime. We finally compute d such that $ed = 1 \mod \phi(N)$ and then reduce it modulo (P-1) and (Q-1). We also pre-compute $(P_{-}inv_{-}Q) = P^{-1} \mod Q$: all these values are fixed and needed for each decryption. The two integers $\langle N, e \rangle$ are the public key and $\langle P, Q, d_P, d_Q, (P_{-}inv_{-}Q) \rangle$ is the secret key. Please see §3.1 for more information.

• Encryption:

The message to be encrypted must be first converted to an integer $M \in \mathbb{Z}/N\mathbb{Z}$. There are standard algorithms, for example in the PKCS#1 standard ([Lab02]). Having the plaintext M, the ciphertext is simply computed as $C = M^e \mod N$.

• Decryption:

Instead of directly computing $M = C^d \mod N$, the decryption algorithm evaluates $M_P = C_P^{d_P} \mod P$ and $M_Q = C_Q^{d_Q} \mod Q$ where $C_P = C \mod P$, $d_P = d \mod P - 1$ and $C_Q = C \mod Q$, $d_Q = d \mod Q - 1$. It is then possible to recover the plaintext M thanks to the Chinese remainder theorem. This method is faster because it computes two exponentiations of n/2-bit integers instead of one exponentiation of n-bit integers. Thus we can theoretically speed up the decryption by four.

3.5.2 Description of the decryption stage

We take the ciphertext C, the modulus and its factorization N = PQ and the reduced exponents $dP = d \mod P - 1$ and $dQ = d \mod Q - 1$ where d is the secret exponent. $(P_{-}inv_{-}Q)$ is a precomputed parameter for Garner's algorithm. Please see §3.4 for further explanations about Garner's algorithm. We want to recover the plaintext M such that $M^{e} = C \mod N$.

Algorithm 14: RSA using CRT

UNIT: RSA_CRT; INPUT: $C, N, d_P, d_Q, P, Q, (P_inv_Q);$ OUTPUT: M;

1. $C_P \leftarrow C \mod P$; 2. $C_Q \leftarrow C \mod Q$; 3. $M_P \leftarrow C_P^{d_P} \mod P$; 4. $M_Q \leftarrow C_Q^{d_Q} \mod Q$; 5. $M \leftarrow \text{GARNER}(M_P, M_Q, P, Q, (P_-inv_-Q), N)$; 6. return(M);

Steps 1 and 2 are the reduction steps; steps 3 and 4 the exponentiation steps and step 5 the Chinese remainder theorem step using Garner's algorithm.

3.5.3 Correctness of the decryption algorithm

Given the standard RSA parameters (C: encrypted message, N: modulus, d: secret key, e: public key), we want to make use of the known factorization N = PQ. If we have computed $M_P =$

 $C^d \mod P$ and $M_Q = C^d \mod Q$, and we can recover M thanks to the Chinese remainder theorem. First, $M_P = C^d \mod P = C_P^d \mod P$, where $C_P = C \mod P$. Besides, $d_P = d \mod P - 1$: there is an integer k such that $d = d_P + k(P - 1)$. We can now write:

$$M_P = C_P^d \mod P$$

= $C_P^{d_P+k(P-1)} \mod P$
= $C_P^{d_P} * (C_P^{P-1})^k \mod P$

Fermat's little theorem shows us that $a^{\phi(n)} = 1 \mod n$ if gcd(a, n) = 1. In our case, P is a prime: $\phi(P) = P - 1$ and $gcd(C_P, P) = 1$. That is why $C_P^{P-1} = 1 \mod P$. Finally:

$$M_P = C_P^{d_P} \mod P$$

Of course, We can do the same modulo Q:

$$M_Q = C_Q^{d_Q} \bmod Q$$

Given M_P and M_Q , we are now able to recover M thanks to Garner's algorithm.

3.5.4 Performance analysis

The critical part of the algorithm is the two exponentiations at step 3 and 4. However, we compute here with moduli whose size is half of the initial modulus size. Thus, we can achieve a speed-up of factor 4.

• Notations:

n is the bit length of N and We assume that the bit length of P and Q is n/2.

• Cost Estimation:

We perform two modular reductions of *n*-bit integers modulo n/2-bit integers. A modular reduction being equivalent to a division means we have here an asymptotic cost of $n^2/2 + o(n^2)$.

In the exponentiation steps 3 and 4, we compute two exponentiations of n/2-bit integers; each of these steps costs $3n^3/8 + n^2/4 + o(n^2)$. Please see §3.1 for more information about the asymptotic behavior of exponentiation.

The recovering using Garner's algorithm at step 5 costs $5n^2/2 + o(n^2)$. Please see §3.4 for further information about Garner's algorithm.

Asymptotic behavior: $3n^3/4 + 7n^2/2 + o(n^2)$

• Memory Cost:

– System parameters

| Register names | Bits | Number of registers |
|-------------------|------------|---------------------|
| C, M, N | n | 3 |
| d_P, d_Q, P, Q | n/2 | 4 |
| $(P_{-}inv_{-}Q)$ | n/2 | 1 |
| Subtotal | 11n/2 bits | |

- Accumulators

| Register names | Bits Number of registers | | |
|----------------------|--------------------------|---|--|
| C_P, C_Q, M_P, M_Q | n/2 | 4 | |
| GARNER | n/2 1 | | |
| Subtotal | 5n/2 bits | | |

- Total memory cost: 8n bits

3.5.5 Comparison with basic RSA

• Cost Estimation:

| | Speed | Speed-up for $n=1024$ bits |
|-----------------|----------------------------|----------------------------|
| RSA without CRT | $3n^3 + n^2 + o(n^2)$ | 1.0 |
| RSA with CRT | $3n^3/4 + 7n^2/2 + o(n^2)$ | 3.98 |

• Memory:

| | Total Memory | System Parameters | Accumulators |
|-----------------|--------------|-------------------|--------------|
| RSA without CRT | 4n bits | 4n bits | 0 bit |
| n=1024 bits | 4096 bits | 4096 bits | 0 bits |
| RSA with CRT | 8n bits | 11n/2 bits | 5n/2 bits |
| n=1024 bits | 8192 bits | 5632 bits | 2560 bits |

RSA decryption is about 4 times faster with the Chinese remainder theorem. However this enhancement costs memory: the total memory cost is twice as large. For a modulus whose bit length is 1024, we only need 4096 bits without Chinese remainder theorem for storing system parameters, whereas here we need 5632 bits plus some extra RAM memory for accumulators. Even if this solution is significantly faster, we will have to think how to correctly develop a smart card architecture that fit these algorithms well: the RAM and EEPROM requirements are higher in this version.

Chapter 4

Rebalanced RSA

A fast decryption or signing stage is often wanted: typically, SSL servers (performing RSA decryption) are overloaded whereas SSL browsers (performing RSA encryption) have idle cycles. Therefore a fast decryption stage would speed up the whole process, even if it is at the expense of the encryption stage. This is exactly the main idea of a rebalanced RSA algorithm, where the work is shifted to the encrypter whereas the decryption is much faster. Instead of choosing a small secret exponent d, what is known to be insecure as soon as $d < N^{0.292}$ ([BG00], [Wie90]), we can pick d such that $d \mod P - 1$ and $d \mod Q - 1$ are small. These values are practically used to get the plain text M back; the smaller they are, the faster the exponentiation stage will be. See [Wie90] for more information about rebalanced RSA.

Contents

| 4.1 | Key generation, encryption and decryption stages | 37 |
|-----|--|-----------|
| 4.2 | Performances of encryption and decryption stages | 39 |
| 4.3 | Comparison with other variants of RSA | 40 |

4.1 Key generation, encryption and decryption stages

4.1.1 Key generation

The algorithm takes two security parameters as input: n (typically 1024) and k (typically 160) where k < n/2. It next picks two primes P and Q verifying gcd(P-1, Q-1) = 2 and whose bit length is n/2. The modulus N is N = PQ. It then randomly picks two k-bit values dP and dQ such that gcd(dP, P-1) = 1, gcd(dQ, Q-1) = 1 and $dP = dQ \mod 2$. The secret exponent d must verify: $d = dP \mod P - 1$ and $d = dQ \mod Q - 1$. We cannot directly compute d with the Chinese remainder theorem because P-1 and Q-1 are not relatively prime (they are both even!). But we chose them such that gcd(P-1, Q-1) = 2, therefore:

$$gcd(\frac{P-1}{2},\frac{Q-1}{2}) = 1$$

We also know that $dP = dQ \mod 2$; let $a = dP \mod 2$. Thanks to the Chinese remainder theorem, we can now compute a value d' verifying:

$$d' = \frac{dP-a}{2} \mod \frac{P-1}{2}$$
$$d' = \frac{dQ-a}{2} \mod \frac{Q-1}{2}$$

In other words, there exist two integers k_1 and k_2 such that:

$$d' = \frac{dP - a}{2} + k_1 * \frac{P - 1}{2}$$
$$d' = \frac{dQ - a}{2} + k_2 * \frac{Q - 1}{2}$$

Let d = 2d' + a. d verifies:

$$d = (dP - a) + k_1 * (P - 1) + a$$

$$d = (dQ - a) + k_2 * (Q - 1) + a$$

Modulo P-1 and Q-1, we get:

$$d = dP \mod P - 1$$

$$d = dQ \mod Q - 1$$

To compute the public exponent e, we just have to compute the inverse of d modulo $\phi(N) = (P-1)(Q-1)$. This is allowed because gcd(dP, P-1) = gcd(dQ, Q-1) = 1. Therefore, gcd(d, P-1) = gcd(d, Q-1) = 1. And finally gcd(d, (P-1)(Q-1)) = 1. We have no control over e which is of the order of N. The encryption won't be as fast as in standard RSA but we manage to increase the speed of the decryption stage.

Algorithm 15: Rebalanced RSA: key generation

UNIT: KEY; INPUT: n, k; OUTPUT: P, Q, e, d;

1. Pick primes P and Q whose bit length is n/2 and verifying gcd(P-1, Q-1) = 2

Pick dP and dQ such that:

 (a) the bit length of dP and dQ is k
 (b) gcd(dP, P - 1) = gcd(dQ, Q - 1) = 1
 (c) dP = dQ = a mod 2

 compute d' such that:

 (a) d' = (dP - a)/2 mod (P - 1)/2
 (b) d' = (dQ - a)/2 mod (Q - 1)/2

 compute d = 2d' + a
 compute e = d⁻¹ mod (P - 1)(Q - 1)
 return(P,Q, dP, dQ, e, d)

38

4.1.2 Encryption

This is exactly the same as in standard RSA, except that e is much larger. The public key is $\langle N, e \rangle$. Please note that Microsoft Internet Explorer does not accept this method because it does not accept public exponents e larger than 32 bits.

4.1.3 Decryption

The private key is $\langle P, Q, dP, dQ \rangle$. One can decrypt a ciphertext C by computing $M_P = C^{dP} \mod P$ and $M_Q = C^{dQ} \mod Q$. Using the Chinese remainder theorem, we are then able to recover the plaintext M, verifying $M = M_P \mod P$ and $M = M_Q \mod Q$.

4.2 Performances of encryption and decryption stages

Regarding the implementation of both of encryption and decryption stages, rebalanced RSA is exactly the same as standard RSA. We just used a trick in order to reduce the size of the exponents in the exponentiation steps in the decryption stage. However, the asymptotic behavior of the encryption step is relatively bad and there is no possible improvement: we perform an exponentiation of *n*-bit integers, which costs $3n^3 + 3n^2$. Let's consider the decryption stage:

• Notations:

n is the bit length of *N*, *k* is the bit length of dP and dQ and, as usual, we assume that the bit length of *P* and *Q* is n/2.

• Cost Estimation:

The decryption stage is the same as in RSA with CRT but we have managed to reduce the size of the secret exponent, therefore the exponentiations steps are faster. The two reductions cost $n^2/2 + o(n^2)$. Then we want to perform two exponentiations of n/2-bit basis, n/2-bit moduli and k-bit exponents. Therefore we have an asymptotic behavior of: $(3k/2-1)n^2+o(n^2)$. Please see §3.2 for further explanations about exponentiations. Garner's step costs $5n^2/2 + o(n^2)$. See §3.4 for more information about Garner's algorithm. Asymptotic behavior: $n^2(3k/2+2) + o(n^2)$

- Memory Cost:
 - System parameters:

| Register names | Bits | Number of registers |
|----------------|------|---------------------|
| C, M, N | n | 3 |
| d_P, d_Q | k | 2 |
| P,Q,(PinvQ) | n/2 | 3 |
| Subtotal | | 9n/2 + 2k bits |
| | | |

- Accumulators:

| | Register names | Bits | Number of registers |
|---|----------------|------|---------------------|
| ſ | C_P, C_Q | n/2 | 2 |
| | M_P, M_Q | n/2 | 2 |
| | GARNER | n/2 | 1 |
| | Subtotal | | 5n/2 bits |
| | | | |

- Total memory cost: 7n + 2k bits

| • | • Cost Estimation of the encryption stage: | | | | | |
|---|--|---------------------|----------------------------------|--|--|--|
| | - | | Speed-up for $k = 160, n = 1024$ | | | |
| | Classic RSA | $(3k-2)*n^2+o(n^2)$ | 1.0 | | | |
| | Rebalanced RSA | $3n^3 + n^2$ | 0.16 | | | |

4.3 Comparison with other variants of RSA

• Cost Estimation of the decryption stage:

| 2 obt 250mmation of the acciption stage. | | | | |
|--|----------------------------|----------------------------------|--|--|
| Speed | | Speed-up for $k = 160, n = 1024$ | | |
| RSA without CRT | $3n^3 + n^2$ | 1.0 | | |
| RSA with CRT | $3n^3/4 + 7n^2/2 + o(n^2)$ | 3.98 | | |
| Rebalanced RSA | $n^2(3k/2+2) + o(n^2)$ | 12.70 | | |

• Memory:

| | Total Memory | System Parameters | Accumulators |
|-----------------|--------------|-------------------|--------------|
| RSA without CRT | 4n bits | 4n bits | 0 bit |
| n=1024 bits | 4096 bits | 4096 bits | 0 bits |
| RSA with CRT | 8n bits | 11n/2 bits | 5n/2 bits |
| n=1024 bits | 8192 bits | 5632 bits | 2560 bits |
| Rebalanced RSA | 7n+2k bits | 9n/2 + 2k bits | 5n/2 bits |
| n=1024 bits | 7200 bits | 4640 bits | 2560 bits |

Compared to RSA with the Chinese remainder theorem decryption, we have a speed-up factor of about n/2k. Typical values for n and k are 1024 and 160; we hence get a theoretical speed-up of 3.2. However, the encryption stage is a lot slower because e is of the order of N: the speed-up factor is here about k/n (with the typical values of n and k we get a theoretical speed-up factor of 0.16; it means that rebalanced RSA encryption is 6.4 times slower than a normal RSA encryption).

Chapter 5

Multi-Prime RSA

The decryption speed of RSA can be increased thanks to the Chinese remainder theorem. Instead of a modulus such as N = PQ, we can use more primes; for example N = PQR ([CHLS97]). However, with a bit length of 1024, it is not secure anymore to use a decomposition of more than three primes, because 256-bit factors would be within the range of RSA-512 factoring project [CDL⁺00], using Elliptic Curve Method (ECM, see [SSW93]).

Contents

| 5.1 | Garner's algorithm extension when $N = PQR$ | 41 |
|------------|--|-----------|
| 5.2 | Multi-Prime RSA modulo $N = PQR$ | 43 |
| 5.3 | General Garner's algorithm | 46 |
| 5.4 | Multi-Prime RSA modulo $N = \prod_{i=1}^{b} P_i \dots \dots \dots \dots \dots$ | 47 |

5.1 Garner's algorithm extension when N = PQR

5.1.1 Description of Garner's algorithm

Garner's algorithm is an optimized version of the Chinese remainder theorem, which allows us to speed up the computations while consuming a little bit more memory. No inverses are computed. It has already been described for a modulus product of two primes; let's consider the case of a modulus product of three primes. Algorithm 16 takes M_P , M_Q , M_R , P, Q, R and N, where N = PQR, as input . We have also two extra input parameters, namely (PQ_inv_R) and (P_inv_Q) , verifying $(PQ_inv_R) = (PQ)^{-1} \mod R$ and $(P_inv_Q) = P^{-1} \mod Q$. We compute M such that:

$$\begin{cases} M = M_P \mod P \\ M = M_Q \mod Q \\ M = M_R \mod R \end{cases}$$

Algorithm 16: Garner's algorithm for N = PQR

UNIT: GARNER_PQR; INPUT: $M_P, M_Q, M_R, P, Q, R, N, (PQ_inv_R), (P_inv_Q);$ OUTPUT: M verifying $M = M_P \mod P, M = M_Q \mod Q$ and $M = M_R \mod R;$

1. $V \leftarrow M_Q - M_P \mod Q;$ 2. $V \leftarrow V * (P_-inv_-Q) \mod Q;$ 3. $M_{PQ} \leftarrow V * P \mod PQ;$ 4. $M_{PQ} \leftarrow M_{PQ} + M_P \mod PQ;$ 5. $V \leftarrow M_R - M_{PQ} \mod R;$ 6. $V \leftarrow V * (PQ_-inv_-R) \mod R;$ 7. $M \leftarrow V * P \mod N;$ 8. $M \leftarrow M * Q \mod N;$ 9. $M \leftarrow M_{PQ} + M \mod N;$ 10. return(M);

5.1.2 Correctness of Garner's algorithm

We want to compute M such that $M = M_P \mod P$, $M = M_Q \mod Q$ and $M = M_R \mod R$. We have precomputed $(P_i inv_Q) = P^{-1} \mod Q$ and $(PQ_i inv_R) = (PQ)^{-1} \mod R$. M verifies:

$$M = M_{PQ} + P * Q * (M_R - M_{PQ}) * (PQ_i inv_R) \mod N$$

Let's take this equation modulo R: We know that $P * Q * (PQ_inv_R) = 1 \mod R$. We get:

 $M = M_{PQ} + (P * Q * (PQ_-inv_-R)) * (M_R - M_{PQ}) \mod R$

 $M = M_{PQ} + M_R - M_{PQ} \mod R$

And finally $M = M_R \mod R$ Let's take the equation modulo Q:

$$M = M_{PQ} + Q * (P * (PQ_-inv_-R)) * (M_R - M_{PQ}) \mod Q$$
$$M = M_{PQ} \mod Q$$

Besides $M_{PQ} = M_P + (P * (P_i inv_Q)) * (M_Q - M_P) \mod PQ$. We also know that $P * (P_i inv_Q) = 1 \mod Q$. Modulo Q, we get now:

$$M_{PQ} = M_P + M_Q - M_P \mod Q$$

And finally $M = M_Q \mod Q$. Let's take the equation modulo P:

$$M = M_{PQ} + P * (Q * (PQ_i inv_R)) * (M_R - M_{PQ}) \mod P$$
$$M = M_{PQ} \mod P$$

And $M_{PQ} = M_P + P * (P_i inv_Q) * (M_Q - M_P) \mod PQ$. Modulo P we get:

$$M_{PQ} = M_P \mod P$$

Finally $M = M_P \mod P$

5.1.3 Performances of Garner's algorithm

• Notations:

Let n be the bit length of N; we assume that the bit length of P, Q and R is n/3

- Cost Estimation:
 - -2 multiplication of n/3-bit integers at steps 2 and 6
 - -1 multiplication of 2n/3-bit integers at step 3
 - -2 multiplication of *n*-bit integers at steps 7 and 8

Asymptotic behavior: $16n^2/3 + o(n^2)$

- Memory Cost:
 - System Parameters

| Register names | Bits | Number of registers |
|-------------------|------|---------------------|
| N, M | n | 2 |
| M_P, M_Q, M_R | n/3 | 3 |
| P,Q,R | n/3 | 3 |
| (PinvQ), (PQinvR) | n/3 | 2 |
| Subtotal | | 14n/3 bits |

- Accumulators

| Register names | Bits | Number of registers |
|----------------|---------------|---------------------|
| V | n/3 | 1 |
| M_{PQ} | 2n/3 | 1 |
| Subtotal | <i>n</i> bits | |

- Total memory cost: 17n/3 bits

5.2 Multi-Prime RSA modulo N = PQR

Multi-Prime RSA with three primes N = PQR offers performances that are comparable to a classic elliptic curve based cryptosystem. The encryption stage is exactly the same as in classic RSA encryption schemes and the decryption stage is faster. For few modifications, we can obtain an efficient and secure cryptosystem. See [CHLS97] for more information about Multi-Prime RSA.

5.2.1 Description of the Multi-Prime RSA cryptosystem

• Key generation:

The key generation is the same as in classic RSA schemes but with three primes instead of two: we have as input a security parameter n, we choose three primes P, Q and R whose bit length is n/3 and multiply them to get the modulus N = PQR. Then we pick some integer e such that $\phi(N) = (P-1)(Q-1)(R-1)$ and e are relatively prime. We finally compute d such that $ed = 1 \mod \phi(N)$ and $dP = d \mod P - 1$, $dQ = d \mod Q - 1$ and $dR = d \mod R - 1$. We also pre-compute $(PQ_{-}inv_{-}R) = (PQ)^{-1} \mod R$ and $(P_{-}inv_{-}Q) = P^{-1} \mod Q$. The two integers $\langle N, e \rangle$ are the public key and $\langle P, Q, R, dP, dQ, dR, (PQ_{-}inv_{-}R), (P_{-}inv_{-}Q) \rangle$ is the secret key. Please see §3.1 for more information about standard RSA.

• Encryption:

The encryption stage is the same as in classic RSA encryption schemes.

• Decryption:

The principle is basically the same as in RSA using the Chinese remainder theorem described in §3.5, unless the composite modulus N has 3 prime factors N = PQR. Then we can compute in $\mathbb{Z}/P\mathbb{Z}$, $\mathbb{Z}/Q\mathbb{Z}$ and $\mathbb{Z}/R\mathbb{Z}$ instead of $\mathbb{Z}/N\mathbb{Z}$ and then get the plaintext back thanks to Chinese remainder theorem. This is faster because the bit length of integers we are working with is shorter, therefore we need less operations to compute multiplications and exponentiations. We need to compute 3 exponentiations of n/3-bit integers, hence we can expect decryptions to be 9 times faster than RSA decryptions without Chinese remainder theorem.

5.2.2 Description of the decryption stage

Algorithm 17 performs a decryption, using a composite modulus N = PQR. We have the classic input parameters: ciphertext C, modulus N, but also its prime factors P, Q and R. Instead of storing the *n*-bit private key d in EEPROM memory, we store $d_P = d \mod P - 1$, $d_Q = d \mod Q - 1$ and $d_R = d \mod R - 1$: these n/3-bit integers don't require more memory while avoiding reductions. As extra input parameters, we need $(PQ_inv_R) = (PQ)^{-1} \mod R$ and $(P_inv_Q) = P^{-1} \mod Q$ for Garner's algorithm.

Algorithm 17: Multi-Prime RSA decryption with N = PQR

UNIT: RSA_PQR; INPUT: $C, N, P, Q, R, d_P, d_Q, d_R, (PQ_inv_R), (P_inv_Q);$ OUTPUT: M;

1. $C_P \leftarrow C \mod P$; 2. $C_Q \leftarrow C \mod Q$; 3. $C_R \leftarrow C \mod R$; 4. $M_P \leftarrow C_P^{d_P} \mod P$; 5. $M_Q \leftarrow C_Q^{d_Q} \mod Q$; 6. $M_R \leftarrow C_R^{d_R} \mod R$; 7. $M \leftarrow \text{GARNER}_P \text{QR} (M_P, M_Q, M_R, P, Q, R, (P_inv_Q), (PQ_inv_R), N)$ 8. return(M);

5.2.3 Performances of the Multi-Prime RSA decryption algorithm

The critical part of the algorithm is the exponentiation step, where three exponentiations of n/3-bit integers are computed; the reduction step is negligible. We are computing with n/3-bit integers, therefore the exponentiation stage is nine times faster than in the classical RSA decryption algorithm without the Chinese remainder theorem.

• Notations:

n is the bit length of N and we assume that the bit length of P, Q and R is n/3.

44

5.2. MULTI-PRIME RSA MODULO N = PQR

• Cost Estimation:

At the reduction stage, we compute 3 reductions of a *n*-bit integer with n/3-bit moduli, which costs $2n^2/3 + o(n^2)$. At the exponentiation stage, we compute 3 exponentiations of n/3-bit integers at step 4, 5 and 6, which costs $n^3/3 + n^2/3$. At Garner's stage, we compute some auxiliary multiplications, which cost $16n^2/3 + o(n^2)$ (see §5.1.3) Asymptotic behavior: $n^3/3 + 19n^2/3 + o(n^2)$

• Memory Cost:

– System Parameters

| <i>j</i> - | | |
|--|------|---------------------|
| Register names | Bits | Number of registers |
| C, N, M | n | 3 |
| d_P, d_Q, d_R | n/3 | 3 |
| P,Q,R | n/3 | 3 |
| (PinvQ), (PQinvR) | n/3 | 2 |
| Subtotal | | 17n/3 bits |

- <u>Accumulators</u>

| | recumulators | | | |
|-----------------|--------------|---------------------|--|--|
| Register names | Bits | Number of registers | | |
| C_P, C_Q, C_R | n/3 | 3 | | |
| M_P, M_Q, M_R | n/3 | 3 | | |
| GARNER | n | 1 | | |
| Subtotal | 3n bits | | | |

- Total memory cost: 26n/3 bits

5.2.4 Comparison with standard RSA decryption algorithms

We will here compare a Multi-Prime RSA decryption and classic RSA decryptions with and without the Chinese remainder theorem, using a bit length n = 1024 bits for the public modulus N. RSA Multi-Prime is about 8.8 times faster than RSA without Chinese remainder theorem and 2.2 times faster than RSA with Chinese remainder theorem, while only consuming a little bit more memory.

| Cost Estimation: | Speed | Speed-up for $n=1024$ bits |
|------------------|----------------------------|----------------------------|
| RSA without CRT | $3n^3 + n^2$ | 1.0 |
| RSA with CRT | $3n^3/4 + n^2/2 + o(n^2)$ | 3.98 |
| RSA modulo PQR | $n^3/3 + 19n^2/3 + o(n^2)$ | 8.84 |

• Cost Estimation:

• Memory:

| | Total Memory | System Parameters | Accumulators |
|------------------|--------------|-------------------|--------------|
| RSA without CRT | 4n bits | 4n bits | 0 bit |
| n=1024 bits | 4096 bits | 4096 bits | 0 bits |
| RSA with CRT | 8n bits | 11n/2 bits | 5n/2 bits |
| n=1024 bits | 8192 bits | 5632 bits | 2560 bits |
| RSA modulo PQR | 17n/3 bits | 3n bits | 26n/3 bits |
| n=1024 bits | 8875 bits | 5803 bits | 3072 bits |

5.3 General Garner's algorithm

Although a factorization of more than three prime would not be secure with modulus whose bit length is only 1024, the recommended key length for RSA may be longer in the future, allowing factorizations of b primes. In the following, we describe Garner's algorithm in the general case, having a modulus $N = \prod_{i=1}^{b} P_i$.

5.3.1 Description of Garner's algorithm with $N = \prod_{i=1}^{b} P_i$

Algorithm 19 takes the modulus N and its prime factors P_1, \ldots, P_b , and the residues M_1, \ldots, M_b as input. We also have b-1 more system parameters, namely A_2, \ldots, A_b computed as it follows:

Algorithm 18: Garner's algorithm pre-computations

UNIT: MULTI_ GARNER; INPUT: P_1, \ldots, P_b ; OUTPUT: A_2, \ldots, A_b ;

```
1. for i from 2 to b do

(a) A_i \leftarrow 1;

(b) for j from 1 to (i - 1) do

i. U \leftarrow P_j^{-1} \mod P_i;

ii. A_i \leftarrow U * A_i;

2. return(A_2, \ldots, A_b);
```

We want to compute M such that $M_i = M \mod P_i$ for all i from 1 to b. See [MOV97, p. 612] for more information.

Algorithm 19: Garner's algorithm with $N = \prod_{i=1}^{b} P_i$

```
UNIT: MULTI_ GARNER;
INPUT: M_1, \ldots, M_b, P_1, \ldots, P_b, N, A_2, \ldots, A_b;
OUTPUT: M;
```

1. $U \leftarrow M_1$; 2. $M \leftarrow U$; 3. $P \leftarrow P_1$; 4. for *i* from 2 to *b* do (a) $U \leftarrow (M_i - M) * C_i \mod P_i$; (b) $M \leftarrow M + U * P$; (c) $P \leftarrow P * P_i$; 5. return(*M*);

5.3.2 Efficiency of Garner's algorithm

• Notations:

n is the bit length of N and we assume that the bit length of P_i is n/b.

• Cost Estimation:

We compute a modular multiplication of n/b-bit integers and two multiplications in \mathbb{Z} at each step of the loop.

Asymptotic behavior: $n^2 * 2 * (b - 1 + 1/b - 1/b^2) + o(n^2)$

• Memory Cost:

| _ | Syst | tem | Parame | \mathbf{ters} |
|---|------|-----|--------|-----------------|
| | D | • • | | DU |

| Bits | Number of registers |
|------|--|
| n | 2 |
| n/b | b |
| n/b | b |
| n | b - 1 |
| | (b+3)n bits |
| | $egin{array}{c} n \ n/b \ n/b \ n/b \end{array}$ |

– <u>Accumulators</u>

| Register names | Bits | Number of registers |
|----------------|---------------|---------------------|
| U | n/b | 1 |
| Р | n | 1 |
| Subtotal | n(1+1/b) bits | |

- Total memory cost: n(b+4+1/b) bits

5.4 Multi-Prime RSA modulo $N = \prod_{i=1}^{b} P_i$

Currently, the recommended key length is 1024 bits. But in the future, we can expect it to be longer. Then it will be possible to have a modulus with more than three prime factors.

5.4.1 Description of the RSA Multi-Prime cryptosystem

• Key generation:

The key generation is the same as in classic RSA schemes but with b primes instead of two: we have as input a security parameter n, we choose b primes P, Q and R whose bit length is n/b and multiply them to get the modulus N. Then we pick some integer e such that $\phi(N) = \prod_{i=1}^{b} (P_i - 1)$ and e are relatively prime. We finally compute d such that $ed = 1 \mod \phi(N)$ and $d_i = d \mod P_i - 1$. We also pre-compute A_2, \ldots, A_b with algorithm 18. The two integers $\langle N, e \rangle$ are the public key and $\langle P_1, \ldots, P_b, d_1, \ldots, d_b, A_2, \ldots, A_b \rangle$ is the secret key. Please see §3.1 for more information about standard RSA.

• Encryption:

The encryption stage is the same as in classic RSA encryption schemes.

• Decryption:

The principle is basically the same as in RSA using the Chinese remainder theorem described in §3.5, unless that the composite modulus N has b prime factors $N = \prod_{i=1}^{b} P_i$. We need

to compute b exponentiations of n/b-bit integers, hence we can expect decryptions to be b^2 times faster than RSA decryptions without Chinese remainder theorem.

5.4.2 Description of the decryption stage

Algorithm 20 performs a decryption, using a composite modulus $N = \prod_{i=1}^{b} P_i$. We have the classic input parameters: ciphertext C, modulus N, but also its prime factors P_i , the secret exponents d_i and Garner's pre-computed parameters A_i .

Algorithm 20: Multi-prime RSA decryption with $N = \prod_{i=1}^{b} P_i$

```
UNIT: RSA_PQR;
INPUT: C, N, P_1, \ldots, P_b, d_1, \ldots, d_b, A_2, \ldots, A_b;
OUTPUT: M;
```

1. for *i* from 1 to *b* do (a) $C_i \leftarrow C \mod P_i$; (b) $M_i \leftarrow C_i^{d_i} \mod P_i$; 2. $M \leftarrow \text{GARNER}_- \text{GEN} (M_1, \dots, M_b, P_1, \dots, P_b, N, A_2, \dots, A_b)$ 3. return(M);

5.4.3 Performances of RSA Multi-Prime decryption algorithm

The critical part of the algorithm is the exponentiation step, where b exponentiations of n/b-bit integers are computed. We can expect a speed-up of b^2 over a RSA decryption without the Chinese remainder theorem.

• Notations:

n is the bit length of N and we assume that the bit length of P_i is n/b.

• Cost Estimation:

At the reduction stage, b reductions of a n-bit integer by n/b-bit moduli are computed, which costs:

$$n^2(1-1/b) + o(n^2)$$

At the exponentiation stage, we compute b exponentiations of n/b-bit integers, which costs:

$$3 * n^3/b^2 + n^2/b + o(n^2)$$

At Garner's stage (see §5.3), we compute some auxiliary multiplications, which costs:

$$n^{2} * 2 * (b - 1 + 1/b - 1/b^{2}) + o(n^{2})$$

Asymptotic behavior: $3n^3/b^2 + n^2 * (2b - 1 + 2/b - 2/b^2) + o(n^2)$

• Memory Cost:

48

– System Parameters

| Register names | Bits | Number of registers |
|------------------|------|---------------------|
| C, N, M | n | 3 |
| d_1,\ldots,d_b | n/b | b |
| P_1,\ldots,P_b | n/b | b |
| A_2,\ldots,A_b | n | b-1 |
| Subtotal | | n(b+4) bits |

- <u>Accumulators</u>

| Register names | Bits | Number of registers |
|------------------|---------------|---------------------|
| C_1,\ldots,C_b | n/b | b |
| M_1,\ldots,M_b | n/b | b |
| GARNER | n(1+1/b) | 1 |
| Subtotal | n(3+1/b) bits | |

– Total memory cost: n(b+7+1/b) bits

CHAPTER 5. MULTI-PRIME RSA

Chapter 6

Multi-Power RSA

In this variant of Multi-Factor RSA, we use a modulus such as $N = P^b Q$ ([Tak98]). If the bit length of N is 1024, b must be smaller or equal to 2 (*i.e.* $N = P^2 Q$ at most), otherwise the factors would be within the range of elliptic curve factoring methods. This cryptosystem has however interesting properties: it is not only fast but it also spares memory compared to other Multi-Factor cryptosystems. In this paper, we describe an improved version which does not require any inversion implementation.

Contents

| 6.1 | Hensel lifting | 51 |
|-----|------------------------------------|-----------|
| 6.2 | Improving Hensel lifting | 54 |
| 6.3 | Multi-Power RSA modulo $N = P^2 Q$ | 56 |
| 6.4 | Successive Hensel liftings | 58 |
| 6.5 | Multi-Power RSA modulo $N = P^b Q$ | 61 |

6.1 Hensel lifting

6.1.1 Basic idea of Hensel lifting

Using a modulus like $N = P^2Q$ with the classical Chinese remainder theorem, performing a RSA decryption would be slower than when using a modulus like N = PQ. While computing in $\mathbb{Z}/P^2\mathbb{Z}$, we are dealing with 2n/3-bit integers; in $\mathbb{Z}/Q\mathbb{Z}$, integers have a bit length of n/3. Using the standard fast exponentiation algorithm, the asymptotic behavior would be $n^3 + o(n^3)$ instead of $3n^3/4$ for RSA using Chinese remainder theorem. But we can take advantage of the *P*-adic representation of the message instead in order to compute M_P such that $C_P = M_P^e \mod P^2$, where $C_P = C \mod P^2$. This method is known as Hensel lifting and allows us to design a cryptosystem faster than Multi-Prime RSA while consuming less memory.

6.1.2 Mathematical justification

Let's write the *P*-adic representation of the plaintext M_P modulo P^2 :

$$M_P = K_0 + P * K_1 \bmod P^2$$

If we now compute $C_P = M_P^e$, we get:

$$C_P = (K_0 + P * K_1)^e \mod P^2$$

= $\sum_{i=0}^e {e \choose i} * K_0^i * P^{e-i} * K_1^{e-i} \mod P^2$

All the terms with a power of P greater than 2 are equal to zero because of the reduction modulo P^2 .

$$C_P = K_0^e + e * P * K_0^{e-1} * K_1 \bmod P^2$$

Let's take this equation modulo P: if $i \neq e$, the term of the sum is equal to zero modulo P:

$$C_P = K_0^e \mod P$$

Like in classic RSA, instead of directly using the private key d where $ed = 1 \mod (P-1)(Q-1)$, we can first reduce it modulo (P-1): $dP = d \mod P - 1$. For further explanations, please see §3.5.3. We are now able to compute K_0 :

$$K_0 = C_P^{dP} \mod P$$

We have now:

$$C - K_0^e = e * P * K_0^{e-1} * K_1 \bmod P^2$$

Let $A = C - K_0^e \mod P^2$, we also have $A = e * P * K_0^{e-1} * K_1 \mod P^2$. In other words there is an integer k such that:

$$A = e * P * K_0^{e-1} * K_1 + k * P^2$$

Therefore, A is divisible by P. We define A_1 as it follows: $A = A_1 * P$, with $0 \le A_1 < P$. Then we also know that:

$$A_1 = e * K_0^{e-1} * K_1 \bmod P$$

And finally:

$$K_1 = A_1 * \left(e * K_0^{e-1}\right)^{-1} \mod P$$

We can then recover the message modulo P^2 : $M_P = K_0 + P * K_1 \mod P^2$.

6.1.3 Description of Hensel lifting

As input we take the ciphertext $C_P = C \mod P^2$, the secret key $d_P = d \mod P - 1$ and the factor P of the modulus N. The output is the plaintext $M_P = M \mod P^2$. The trick is to compute in $\mathbb{Z}/P\mathbb{Z}$ instead of $\mathbb{Z}/P^2\mathbb{Z}$, and then recover the plaintext in $\mathbb{Z}/P^2\mathbb{Z}$.

Algorithm 21 shows an implementation of Hensel lifting. We first compute K_0 , the lowest digit in the P-adic representation of the message reduced modulo P^2 , M_P . This is done at step 2. At step 4, we compute $A = C - K_0^e \mod P^2$. Then we perform a simple division in \mathbb{Z} at step 5 in order to get A_1 such that $A = A_1 * P$. K_1 is computed at step 9 with the formula $K_1 = A_1 * (K_0^{e-1} * e)^{-1} \mod P$ and finally M_P equals $K_0 + P * K_1 \mod P^2$.

Algorithm 21: Hensel Lifting

UNIT: HENSEL; INPUT: C_P, d_P, P ; OUTPUT: M_P ;

1. $P^{2} \leftarrow P * P;$ 2. $K_{0} \leftarrow C_{P}^{d_{P}} \mod P;$ 3. $A \leftarrow -K_{0}^{e} \mod P^{2};$ 4. $A \leftarrow A + C \mod P^{2};$ 5. $A \leftarrow A/P;$ 6. $K_{1} \leftarrow K_{0}^{e-1} \mod P;$ 7. $K_{1} \leftarrow K_{1} * e \mod P;$ 8. $K_{1} \leftarrow (K_{1})^{-1} \mod P;$ 9. $K_{1} \leftarrow K_{1} * A \mod P;$ 10. $M_{P} \leftarrow P * K_{1} \mod P^{2};$ 11. $M_{P} \leftarrow M_{P} + K_{0} \mod P^{2};$ 12. $\mathbf{return}(M_{P});$

6.1.4 Performances of Hensel lifting

The critical part is the exponentiation at step 2. There are also exponentiations at step 3 and 6, but if the public exponent e is kept small (for example $2^{16} - 1$ or even 3), the extra computations are negligible. For security issues and more information about RSA decryption using a modulus P^bQ , please refer to [Tak98].

• Notations:

n is the bit length of the public modulus $N = P^2 Q$ and *k* the bit length of the public exponent *e*. We assume that the bit length of *P* is n/3.

• Cost Estimation:

The exponentiation at step 2 is the critical part of the algorithm. We perform here an exponentiation of n/3-bit integers, which costs $n^3/9 + n^2/9 + o(n^2)$.

There are two more exponentiations at step 3 and step 6. At step 3, the basis is a 2n/3-bit integer whereas at step 6 the basis is a n/3-bit integer. However, the exponent can be kept small in both of them so that the computational costs are cheap, namely:

- $-4/9(3k-2)n^2 + o(n^2)$ for the first exponentiation at step 3
- $-1/9(3k-2)n^2 + o(n^2)$ for the second exponentiation at step 6

If k is for example 16 or even 2, they are negligible compared to the exponentiation at step 2.

At step 5, we perform a division in \mathbb{Z} ; the bit length of A is 2n/3 and the bit length of P is n/3, therefore it costs $n^2/9 + o(n^2)$. The inversion at step 8 is equivalent to 20 multiplications of n/3-bit integers. There are also some multiplications at intermediary steps.

Asymptotic behavior: $n^3/9 + n^2 (5k/3 + 53/9) + o(n^2)$

• Memory Cost:

- System Parameters

| Register names | Bits | Number of registers |
|----------------|---------|---------------------|
| dP, P | n/3 | 2 |
| M_P, C_P | 2n/3 | 2 |
| Subtotal | 2n bits | |

– <u>Accumulators</u>

| riccamator | | |
|----------------|---------|---------------------|
| Register names | Bits | Number of registers |
| P^2 | 2n/3 | 1 |
| A | 2n/3 | 1 |
| K_0, K_1 | n/3 | 2 |
| Subtotal | 2n bits | |

- Total memory cost: 4n bits

6.2 Improving Hensel lifting

6.2.1 Inconvenient of the previous algorithm

In the previous version of Hensel lifting, there are several points which can be slightly improved. We can first avoid the inversion at step 8; on the one hand, the algorithm will be faster and on the other hand, we can spare ROM memory and development time since we do not need any implementation of modular inversion. Besides, it is possible to avoid the computation of the exponentiation at step 6. We want to compute $K_1 = A_1 * e^{-1} * K_0^{1-e} \mod P$; assuming that $(e_{-inv_-}P) = e^{-1} \mod P$ has been precomputed, we still need $K_0^{1-e} \mod P$, but it is possible to transform this expression:

$$K_0^{1-e} = K_0 * (K_0^e)^{-1} \mod P$$

= $C_P^{dP} * (C_P)^{-1} \mod P$
= $C_P^{dP-1} \mod P$

But we also need $C_P^{dP} \mod P$, therefore we can first perform the exponentiation $C_P^{dP-1} \mod P$ and then simply multiply the result with C_P in order to get $C_P^{dP} \mod P$. We can have both expressions for the cost of a single exponentiation.

6.2.2 Description of the improved Hensel lifting algorithm

The inputs and outputs of the algorithm are the same as in the previous version, except the extra input $(e_{-}inv_{-}P) = e^{-1} \mod P$.

Algorithm 22 is a possible implementation of this new version of Hensel lifting. At step 2 we perform the exponentiation we need in order to get both of K_0 and $K_1 * P$, as explained above. Then we compute K_0 with a single multiplication at the following step 3. We know that $K_1 * P = (C - K_0^e) * e^{-1} * C_P^{dP-1} \mod P^2$. Since we already know $e^{-1} \mod P^2$ and $C_P^{dP-1} \mod P^2$, we only need to compute $C - K_0^e \mod P^2$ to finally get K_1 .

Algorithm 22: Improved Hensel Lifting

UNIT: I_HENSEL; INPUT: $C_P, d_P, P, (e_inv_P);$ OUTPUT: $M_P;$

1. $P^2 \leftarrow P * P;$ 2. $M_P \leftarrow C_P^{d_P-1} \mod P;$ 3. $K_0 \leftarrow M_P * C_P \mod P;$ 4. $A \leftarrow -K_0^e \mod P^2;$ 5. $A \leftarrow A + C \mod P^2;$ 6. $M_P \leftarrow M_P * A \mod P^2;$ 7. $M_P \leftarrow M_P * (e_-inv_-P) \mod P^2;$ 8. $M_P \leftarrow M_P + K_0 \mod P^2;$ 9. return $(M_P);$

6.2.3 Performance analysis of improved Hensel lifting

• Notations:

n is the bit length of the public modulus $N = P^2 Q$ and *k* the bit length of the public exponent *e*. We assume that the bit length of *P* is n/3.

• Cost Estimation:

The exponentiation at step 2 deals with n/3-bit integers and costs $n^3/9 + n^2/9$. The second exponentiation at step 4 can be very cheap if e is small; its basis is a 2n/3-bit integer: it costs $4/9(3k-2)n^2 + o(n^2)$. There are some multiplications at intermediary steps, but we don't need to compute any inverse this time.

Asymptotic behavior: $n^3/9 + n^2 (4k/3 + 20/9) + o(n^2)$

- Memory Cost:
 - System Parameters Register names Number of registers Bits dP, P2 n/3 $\mathbf{2}$ M_P, C_P 2n/3 $(e_{-}inv_{-}P)$ n/31 7n/3 bits Subtotal Accumulators Number of registers Register names Bits P^2 2n/31 n/31 K_0 Subtotal n bits
 - Total memory cost: 10n/3 bits

6.2.4 Comparison of the two versions

We can see in the following table that the improved version of Hensel lifting is slightly faster while consuming less memory. However, it requires an additional system parameter, $(e_{-}inv_{-}P)$, which

is stored in EEPROM for the case of smart cards. But the main interest of the improved version is that we don't need any implementation of the modular inversion.

• Cost Estimation:

| | Speed | Speed-up for $n = 1024, k = 16$ |
|------------------|--|---------------------------------|
| Normal version | $n^{3}/9 + n^{2}(5k/3 + 53/9) + o(n^{2})$ | 1.0 |
| Improved version | $n^{3}/9 + n^{2} (4k/3 + 20/9) + o(n^{2})$ | 1.07 |

• Memory:

| | Total Memory | System Parameters | Accumulators |
|------------------|--------------|-------------------|--------------|
| Normal version | 4n bits | 2n bits | 2n bits |
| n = 1024 | 4096 bits | 2048 bits | 2048 bits |
| Improved version | 10n/3 bits | 7n/3 bits | n bits |
| n = 1024 | 3414 bits | 2390 bits | 1024 bits |

6.3 Multi-Power RSA modulo $N = P^2 Q$

6.3.1 Algorithm

• Key generation:

We have as input a security parameter n, we choose two primes P and Q whose bit length is n/3 and compute the modulus $N = P^2Q$. Then we pick some integer e like in standard RSA and compute d such that $ed = 1 \mod (P-1)(Q-1)$. Finally, we compute $dP = d \mod P-1$ and $dQ = d \mod Q - 1$. We also pre-compute $(P^2_inv_Q)$ and (e_inv_P) .

The two integers $\langle N, e \rangle$ are the public key and $\langle dP, dQ, P, Q, (P^2_{-}inv_{-}Q), (e_{-}inv_{-}P) \rangle$ is the secret key. Please see §3.1 for more information about classic RSA.

• Encryption:

The encryption stage is the same as in classic RSA.

• Decryption:

There is a priori no advantages in computing with a modulus $N = P^2Q$ because the bit length of P^2 is 2n/3. With a standard exponentiation method, it is slower than the cryptosystem proposed in §3.5. But we can rather compute $M_P = C_P^d \mod P^2$, where $C_P = C \mod P^2$ and $M_P = M \mod P^2$ using Hensel lifting. It allows us to compute an exponentiation of a 2n/3-bit integer for about the cost of an exponentiation of a n/3-bit integer. We can finally recover the plain text thanks to the Chinese remainder theorem. This method performs a RSA decryption for the cost of two exponentiations of n/3-bit integers and some auxiliary and relatively cheap operations. Therefore we can expect this algorithm to be about 27/2 = 13.5faster than a simple RSA decryption.

6.3.2 Description the decryption stage

The principles of RSA with a modulus $N = P^2Q$ and RSA using the Chinese remainder theorem are basically the same. We just have to compute the exponentiation modulo P^2 with Hensel lifting instead of the classic fast exponentiation method. For further explanations, please see §3.5.3 (RSA using Chinese remainder theorem) and §6.1.2 (Hensel lifting). Algorithm 23 shows an implementation of RSA using a modulus such as $N = P^2Q$.

Algorithm 23: RSA decryption modulo $N = P^2 Q$

UNIT: RSA_P2Q; INPUT: $C, N, P, Q, d_P, d_Q, (P^2_inv_Q), (e_inv_P);$ OUTPUT: M;

1. $P^2 \leftarrow P * P;$ 2. $C_P \leftarrow C \mod P^2;$ 3. $C_Q \leftarrow C \mod Q;$ 4. $M_P \leftarrow I_-\text{HENSEL}(C_P, d_P, P, (e_-inv_-P));$ 5. $M_Q \leftarrow C_Q^{d_Q} \mod Q;$ 6. $V \leftarrow M_Q - M_P \mod Q;$ 7. $V \leftarrow V * (P^2_-inv_-Q) \mod Q;$ 8. $M \leftarrow V * P^2 \mod N;$ 9. $M \leftarrow M + M_P \mod N;$ 10. return(M);

6.3.3 Performances of the decryption stage

The critical parts of the algorithm are the Hensel lifting at step 4 and the exponentiation at step 5. However, here we compute with moduli whose size is a third of the initial modulus size. Thus, we can achieve a theoretical factor 13.5 speed-up.

- Notations:
 - -n: bit length of N
 - We assume that the bit length of P and Q is n/3
- Cost Estimation:

We first compute a multiplication in \mathbb{Z} and two reductions of a *n*-bit integer by a 2n/3-bit and a n/3-bit modulus.

Hensel lifting at step 4 costs $n^3/9 + n^2 (4k/3 + 20/9) + o(n^2)$, whereas the exponentiation at step 5 costs $n^3/9 + n^2/9$.

At Garner's step, we compute 1 multiplication of n/3-bit integers and 1 multiplication of n-bit integers.

Asymptotic behavior: $2n^3/9 + n^2(4k/3 + 6) + o(n^2)$

- Memory Cost:
 - System Parameters

| Register names | Bits | Number of registers |
|----------------------|---------|---------------------|
| C, M, N | n | 3 |
| dP, dQ | n/3 | 2 |
| P,Q | n/3 | 2 |
| $(P^2invQ), (einvP)$ | n/3 | 2 |
| Subtotal | 5n bits | |

| Register names | Bits | Number of registers |
|----------------|------------|---------------------|
| P^2 | 2n/3 | 1 |
| V | n/3 | 1 |
| C_P, M_P | 2n/3 | 2 |
| C_Q, M_Q | n/3 | 2 |
| HENSEL | n/3 | 1 |
| Subtotal | 10n/3 bits | |

- Total memory cost: 25n/3 bits

6.3.4 Comparison with classic RSA

While consuming about as much memory as RSA using the Chinese remainder theorem, RSA modulo $N = P^2 Q$ is about 3 times faster (and 12 times faster than RSA without Chinese remainder theorem). Implemented in a smart card, compared to RSA using Chinese remainder theorem, RSA modulo $N = P^2 Q$ would require less EEPROM memory for system parameters, but more RAM memory for accumulators.

• Cost Estimation:

| | Speed | Speed-up for $n=1024$, $k=16$ |
|-------------------|---|--------------------------------|
| RSA without CRT | $3n^3 + n^2$ | 1.0 |
| RSA with CRT | $3n^3/4 + 7n^2/2 + o(n^2)$ | 3.98 |
| RSA modulo PQR | $n^3/3 + 19n^2/3 + o(n^2)$ | 8.84 |
| RSA modulo P^2Q | $2n^{3}/9 + n^{2}(4k/3 + 6) + o(n^{2})$ | 12.06 |

• Memory:

| | Total Memory | System Parameters | Accumulators |
|-------------------|--------------|-------------------|--------------|
| RSA without CRT | 4n bits | 4n bits | 0 bit |
| n=1024 bits | 4096 bits | 4096 bits | 0 bits |
| RSA with CRT | 8n bits | 11n/2 bits | 5n/2 bits |
| n=1024 bits | 8192 bits | 5632 bits | 2560 bits |
| RSA modulo PQR | 26n/3 bits | 17n/3 bits | 3n bits |
| n=1024 bits | 8875 bits | 5083 bits | 3072 bits |
| RSA modulo P^2Q | 25n/3 bits | 5n bits | 10n/3 bits |
| n=1024 bits | 8534 bits | 5120 bits | 3414 bits |

6.4 Successive Hensel liftings

Although we are now limited to b = 2 in $N = P^bQ$, it is possible that a bit length greater than 1024 will be recommended in the future, allowing greater values for b. In the following, we introduce the general case with successive Hensel liftings.

6.4.1 Main idea

Having $M \mod P$, we are able to recover $M \mod P^2$ thanks to Hensel lifting. But we can keep on executing the algorithm in a loop to recover the message modulo a greater power of P. After each

6.4. SUCCESSIVE HENSEL LIFTINGS

loop, we get $M \mod P^i$, starting from i = 1, until i = b. At the end, we get $M \mod P^b$. We assume that we have $M_{P,i}$, the plaintext modulo P^i ; we show that we can recover it modulo P^{i+1} . Let's write the P^i -adic representation of $M_{P,i+1}$, the plaintext modulo P^{i+1} :

$$M_{P,i+1} = K_0 + K_1 * P^i \mod P^{i+1}$$

We can write $C_{P,i+1}$, the ciphertext modulo P^{i+1} as it follows:

$$C_{P,i+1} = M_{P,i+1}^{e} \mod P^{i+1}$$

= $(K_0 + K_1 * P^i)^e \mod P^{i+1}$
= $\sum_{j=0}^{e} {e \choose j} * K_0^j * P^{i*(e-j)} * K_1^{e-j} \mod P^{i+1}$

The reduction modulo P^{i+1} ensures that for j < e - 1, the terms of the sum are equal to zero. At the end, we only have:

$$C_{P,i+1} = K_0^e + e * K_0^{e-1} * P^i * K_1 \mod P^{i+1}$$

But $K_0 = C_{P,i+1} \mod P^i = M_{P,i}$:

$$C_{P,i+1} = M_{P,i}^e + e * M_{P,i}^{e-1} * P^i * K_1 \mod P^{i+1}$$

There is an integer k such that:

$$C_{P,i+1} - M_{P,i}^e = e * M_{P,i}^{e-1} * P^i * K_1 + k * P^{i+1}$$

We divide the equation above by P^i and compute it modulo P:

$$\frac{C_{P,i+1} - M_{P,i}^e}{P^i} = e * M_{P,i}^{e-1} * K_1 \mod P$$

Then:

$$K_1 = \frac{C_{P,i+1} - M_{P,i}^e}{P^i} * \left(e * M_{P,i}^{e-1}\right)^{-1} \bmod P$$

And finally,

$$M_{P,i+1} = M_{P,i} + K_1 * P^i \mod P^{i+1}$$

6.4.2 Description of successive Hensel lifting

Algorithm 24 is the general version of algorithm 22, which does not compute any inversion. It takes the ciphertext $C_P = M^e \mod P^b$, the private key $d_P = d \mod P - 1$ and P, where $N = P^b Q$. It computes the plaintext $M_P = C^d \mod P^b$, using the same tricks as in algorithm 22, namely:

- We pre-compute $(e_{-}inv_{-}P) = e^{-1} \mod P$
- We first compute $K = C_P^{d_P-1} \mod P$, then $M_P = K * C_P \mod P$ and we can make use of K in order to compute the successive K_1 :

$$K_{1} = \frac{C_{P,i+1} - M_{P,i}^{e}}{P^{i}} * (e_{-}inv_{-}P) * M_{P,i}^{1-e} \mod P$$

$$= \frac{C_{P,i+1} - M_{P,i}^{e}}{P^{i}} * (e_{-}inv_{-}P) * C_{P}^{dP-1} \mod P$$

$$= \frac{C_{P,i+1} - M_{P,i}^{e}}{P^{i}} * (e_{-}inv_{-}P) * K \mod P$$

Hence we compute the following expression $M_{P,i+1} = M_{P,i} + E \mod P^{i+1}$ where:

 $E = (C_{P,i+1} - M_{P,i}^e) (e_{-}inv_{-}P) * K \mod P^{i+1}$

Algorithm 24: Successive Hensel liftings

UNIT: HENSEL_GEN; INPUT: $C_P, d_P, P, (e_-inv_-P);$ OUTPUT: $M_P;$

1. $K \leftarrow C_P^{d_P-1} \mod P$; 2. $M_P \leftarrow K * C_P \mod P$; 3. $P_{power} \leftarrow P$; 4. for *i* from 1 to (b-1) do (a) $P_{power} \leftarrow P_{power} * P$; (b) $F \leftarrow M_P^e \mod P_{power}$; (c) $E \leftarrow C_P - F \mod P_{power}$; (d) $E \leftarrow E * K \mod P_{power}$; (e) $E \leftarrow E * (e_inv_P) \mod P_{power}$; (f) $M_P \leftarrow M_P + E \mod P_{power}$; 5. return (M_P) ;

6.4.3 Performance analysis of successive Hensel liftings

• Notations:

n is the bit length of the public modulus $N = P^b Q$ and *k* the bit length of the public exponent *e*. We assume that the bit length of *P* and *Q* is n/(b+1).

• Cost Estimation:

The exponentiation at step 1 deals with n/(b+1)-bit integers and costs $3n^3/(b+1)^3 + n^2/(b+1)^2 + o(n^2)$. The second exponentiation at step 4b can be kept relatively cheap if e is small; its basis is a (i+1) * n/(b+1)-bit integer and it costs:

$$(3k-2)n^{2}(i+1)^{2}/(b+1)^{2} + o(n^{2})$$

For the whole loop, we have the following cost:

$$(3k-2)n^2/(b+1)^2\sum_{i=1}^{b-1}(i+1)^2+o(n^2)$$

and

$$\sum_{i=1}^{b-1} (i+1)^2 = b^3/3 + b^2/2 + b/6 - 1$$

There are also some multiplications at intermediary steps. Asymptotic behavior:

$$\frac{3n^3}{(b+1)^3} + \frac{n^2}{(b+1)^2} * \left(5b^3/3 + 2b^2 - 2b/3 + k * \left(b^3 + 3b^2/2 + b/2 - 3\right)\right) + o(n^2)$$

• Memory Cost:

– System Parameters

| Register names | Bits | Number of registers |
|-------------------|--------------------|---------------------|
| dP, P | n/(b+1) | 2 |
| M_P, C_P | bn/(b+1) | 2 |
| $(e_{-}inv_{-}P)$ | n/(b+1) | 1 |
| Subtotal | (2b+3)n/(b+1) bits | |

- Accumulators

| Register names | Bits | Number of registers |
|----------------------|----------------|---------------------|
| P_{power}, K, F, E | bn/(b+1) | 4 |
| Subtotal | 4bn/(b+1) bits | |

- Total memory cost: (6b+3)n/(b+1) bits

6.5 Multi-Power RSA modulo $N = P^b Q$

6.5.1 Algorithm

• Key generation:

We have as input a security parameter n, we choose two primes P and Q whose bit length is n/(b+1) and compute the modulus $N = P^bQ$. Then we pick some integer e like in standard RSA and compute d such that $ed = 1 \mod (P-1)(Q-1)$. Finally, we compute $dP = d \mod P - 1$ and $dQ = d \mod Q - 1$. We also pre-compute $(P^2_inv_Q)$ and (e_inv_P) . The two integers $\langle N, e \rangle$ are the public key and $\langle dP, dQ, P, Q, (P^2_inv_Q), (e_inv_P) \rangle$ is the secret key. Please see §3.1 for more information about classic RSA.

• Encryption:

The encryption stage is the same as in classic RSA.

• Decryption:

We compute $M_P = C_P^d \mod P^b$, where $C_P = C \mod P^b$ and $M_P = M \mod P^b$ using Hensel lifting. It allows us to perform an exponentiation of a bn/(b+1)-bit integer for about the cost of an exponentiation of a n/(b+1)-bit integer and some extra operations. We can finally recover the plain text thanks to the Chinese remainder theorem. This method performs a RSA decryption for about the cost of two exponentiations of n/(b+1)-bit integers and some auxiliary operations. Therefore we can expect this algorithm to be about $(b+1)^3/2$ faster than a simple RSA decryption. This is however a first approximation; in fact, the auxiliary operations are not so cheap when b grows up. We also compute $M_Q = C_Q^d \mod Q$ and get the plaintext with Garner's algorithm.

6.5.2 Description of the decryption stage

Algorithm 25 generalizes the decryption algorithm to the case $N = P^b Q$. The principle is the same as when $N = P^2 Q$: first we compute the exponentiation modulo Q and get the plaintext modulo Q, then recover the plaintext modulo P^2 thanks to Hensel lifting and finally get the plaintext back with Garner's algorithm. Algorithm 25: RSA decryption modulo $N = P^b Q$

UNIT: RSA_PbQ; INPUT: $C, N, P, Q, d_P, d_Q, (P^b_inv_Q), (e_inv_P);$ OUTPUT: M;

1. $Pb \leftarrow P$; 2. for *i* from 2 to *b* do $Pb \leftarrow Pb * P$; 3. $C_P \leftarrow C \mod Pb$; 4. $C_Q \leftarrow C \mod Q$; 5. $M_P \leftarrow \text{HENSEL}_{-} \text{GEN}(C_P, d_P, P, (e_-inv_-P))$; 6. $M_Q \leftarrow C_Q^{d_Q} \mod Q$; 7. $V \leftarrow M_Q - M_P \mod Q$; 8. $V \leftarrow V * (P^b_-inv_-Q) \mod Q$; 9. $M \leftarrow V * Pb \mod N$; 10. $M \leftarrow M + M_P \mod N$; 11. return(M);

6.5.3 Performances of the decryption stage

The critical parts of the algorithm are the Hensel lifting at step 5 and the exponentiation at step 6. However, we here compute with moduli which are much smaller than the initial modulus. Thus, we can expect a theoretical speed-up of about $(b + 1)^3/2$.

- Notations: n is bit length of N and we assume that the bit length of P and Q is n/(b+1)
- Cost Estimation:

Each reduction at steps 3 and 3 costs $n^2 * b/(b+1) + o(n^2)$. Hensel lifting at step 5 costs:

$$\frac{3n^3}{(b+1)^3} + \frac{n^2}{(b+1)^2} * \left(5b^3/3 + 2b^2 - 2b/3 + k * \left(b^3 + 3b^2/2 + b/2 - 3\right)\right) + o(n^2)$$

whereas the exponentiation at step 6 costs $3n^3/(b+1)^3 + n^2/(b+1)$.

At Garner's step, we compute 1 multiplication of n/(b+1)-bit integers and 1 multiplication of *n*-bit integers.

Asymptotic behavior:

$$\frac{6n^3}{(b+1)^3} + \frac{n^2}{(b+1)^2} * \left(8b^3/3 + 5b^213b/3 + 3 + k * \left(b^3 + 3b^2/2 + b/2 - 3\right)\right) + o(n^2)$$

- Memory Cost:
 - System Parameters

| Register names | Bits | Number of registers |
|------------------------------|---------|---------------------|
| C, M, N | n | 3 |
| dP, dQ | n/(b+1) | 2 |
| P,Q | n/(b+1) | 2 |
| $(P^2_inv_Q), (e_inv_P)$ | n/(b+1) | 2 |
| Subtotal | n(3 + | -6b/(b+1)) bits |

- Accumulators

| Register names | Bits | Number of registers |
|----------------|--------------------|---------------------|
| Pb | bn/(b+1) | 1 |
| V | n/(b+1) | 1 |
| C_P, M_P | bn/(b+1) | 2 |
| C_Q, M_Q | n/(b+1) | 2 |
| HENSEL | 4bn/(b+1) | 1 |
| Subtotal | (7b+3)n/(b+1) bits | |

- Total memory cost: n(3 + (13b + 3)/(b + 1)) bits

CHAPTER 6. MULTI-POWER RSA

Chapter 7

Batch RSA

A complete different approach consists of grouping different decryptions under certain conditions. For example, a SSL server performs a lot of consecutive RSA decryptions and can be quickly overloaded. But if it waits for more than one RSA decryption request and performs one big computation for all decryptions, it can spare a lot of running time capacity [Fia89], being able to perform more SSL handshakes.

Contents

| 7.1 | Main idea | 65 |
|-----|--|-----------|
| 7.2 | How to increase efficiency? | 69 |
| 7.3 | Improving batch RSA: Montgomery's trick | 70 |
| 7.4 | Improving batch RSA: Shamir's Trick | 72 |
| 7.5 | Batch RSA using Shamir's and Montgomery's Tricks | 74 |
| 7.6 | Batch RSA using the Chinese remainder theorem | 77 |
| 7.7 | Comparison with classic RSA | 78 |

7.1 Main idea

Notation

We will use the notation $M = C^{1/e} \mod N$, which is equivalent to $M = C^d$, where $d = e^{-1} \mod \phi(N)$.

7.1.1 Batching two RSA decryptions

Assuming that we have two messages to encrypt M_1 and M_2 respectively with the public keys $\langle N, 3 \rangle$ and $\langle N, 5 \rangle$, Fiat showed that it is possible to decrypt the ciphertexts $C_1 = M_1^3 \mod N$ and $C_2 = M_2^5 \mod N$ for approximately the price of a single RSA decryption [BS02, p. 3].

Let $A = (C_1^5 * C_2^3)^{1/15}$, then we can compute:

$$\frac{A^{10}}{C_1^3 * C_2^2} = \frac{C_1^{10/3} * C_2^2}{C_1^3 * C_2^2} = M_1 = C_1^{1/3}$$
$$\frac{A^6}{C_1^2 * C_2} = \frac{C_1^2 * C_2^{6/5}}{C_1^2 * C_2} = M_2 = C_2^{1/5}$$

For the cost of a single 15th-root computation (which is equivalent to a single RSA decryption), we can compute two RSA decryptions. However, the public exponents e_1 and e_2 (here 5 and 3) have to be chosen small because the auxiliary exponentiations must be cheap.

7.1.2 General case: batching p RSA decryptions

We can generalize this batching algorithm: assuming that we have p pairwise relatively prime exponents e_i sharing the same modulus N and p ciphertexts C_i respectively encrypted with the exponents e_i , we want to batch the necessary computations to get the plaintexts $M_i = C^{1/e_i} \mod N$. We first set:

$$e = \prod_{i=1}^{p} e_i$$

and

$$A_0 = \prod_{i=1}^p C_i^{e/e_i} \mod N$$

Then we compute $A = A_0^{1/e} \mod N$. Each plaintext M_i can be recovered with the following formula:

$$M_{i} = \frac{A^{\alpha_{i}}}{C_{i}^{(\alpha_{i}-1)/e_{i}} * \prod_{j=1, j \neq i}^{p} C_{j}^{\alpha_{i}/e_{j}}} \bmod N$$

with $\alpha_i = 1 \mod e_i$ and $\alpha_i = 0 \mod e_j$ for $j \neq i$. Algorithm 26 shows an implementation of Batch RSA.

7.1.3 Correctness of Batch RSA

We have the conditions $\alpha_i = 1 \mod e_i$ and $\alpha_i = 0 \mod e_j$; in other words, there are integers k_i and $k_{i,j}$ such that:

$$\alpha_i = 1 + k_i * e_i$$

$$\alpha_i = k_{i,j} * e_j$$

Applying this result to the formula, we get:

$$\frac{A^{\alpha_i}}{C_i^{(\alpha_i-1)/e_i} * \prod_{j=1, j \neq i}^p C_j^{\alpha_i/e_j}} = \frac{A^{\alpha_i}}{C_i^{k_i} * \prod_{j=1, j \neq i}^p C_j^{k_{i,j}}} \bmod N$$

Besides,

$$A^{\alpha_i} = \left(\prod_{j=1}^p C_j^{1/e_j}\right)^{\alpha_i} \mod N$$

66

Algorithm 26: Batch RSA

UNIT: BATCH_RSA; INPUT: $C_1, ..., C_p, e_1, ..., e_p, d_1, ..., d_p, N;$ OUTPUT: $M_1, ..., M_p$ verifying $C_i = M_i^{e_i} \mod N;$

```
1. e \leftarrow e_1; d \leftarrow d_1; A \leftarrow 1;
2. for i from 2 to p
      (a) e \leftarrow e * e_i; d \leftarrow d * d_i;
3. for i from 1 to p
      (a) E \leftarrow e/e_i;
      (b) F \leftarrow E^{-1} \mod e_i;
      (c) a_i \leftarrow E * F \mod N;
(d) F \leftarrow C_i^{e_i} \mod N;
      (e) A \leftarrow A * F \mod N;
4. A \leftarrow A^d \mod N;
5. for i from 1 to p
      (a) M_i \leftarrow A^{a_i} \mod N;
     (b) F \leftarrow a_i - 1 \mod N;
      (c) F \leftarrow F/e_i;
(d) T \leftarrow C_i^F \mod N;
      (e) for j from 1 to p, j \neq i
             i. E \leftarrow a_i/e_j;

ii. F \leftarrow C_j^E \mod N;

iii. T \leftarrow T * F \mod N;
      (f) T \leftarrow T^{-1} \mod N;
      (g) M_i \leftarrow M_i * T \mod N;
6. return(M_1, \ldots, M_p);
```

$$A^{\alpha_i} = C_i^{\alpha_i/e_i} * \prod_{j=1, j \neq i}^p C_j^{\alpha_i/e_j} \mod N$$

Using once again the conditions of α_i , we get $\alpha_i/e_i = 1/e_i + k_i$ and $\alpha_i/e_j = k_{i,j}$ Therefore:

$$A^{\alpha_i} = C_i^{1/e_i} * C_i^{k_i} \prod_{j=1, j \neq i}^p C_j^{k_{i,j}} \mod N$$

Finally:

$$\frac{A^{\alpha_i}}{C_i^{(\alpha_i-1)/e_i} * \prod_{j=1, j \neq i}^p C_j^{\alpha_i/e_j}} = \frac{C_i^{1/e_i} * C_i^{k_i} \prod_{j=1, j \neq i}^p C_j^{k_{i,j}}}{C_i^{k_i} * \prod_{j=1, j \neq i}^p C_j^{k_{i,j}}} \mod N$$
$$\frac{A^{\alpha_i}}{C_i^{(\alpha_i-1)/e_i} * \prod_{j=1, j \neq i}^p C_j^{\alpha_i/e_j}} = M_i \mod N$$

7.1.4 Performance analysis of batch RSA

• Notations:

n is the bit length of the public modulus N, p the number of batched decryptions and k the bit length of the bigger of e_i .

- Cost Estimation: Because k and p are very small compared to n, we can ignore many operations in the speed estimation: if the speed cost of a given operation does not involve at least n^2 , we neglect it.
 - In the first loop at step 2, 2(p-1) multiplications in \mathbb{Z} are computed: it costs $2(p-1)n^2$.
 - The exponent at step 3d is e_i ; its bit length is k therefore this exponentiation costs $(3k-2)n^2+o(n^2)$. The inversion at step 3b is computed modulo e_i , hence it is equivalent to 20 multiplications of k-bit integers: we can neglect this operation since it does not involve n^2 . We have also two multiplication of n-bit integers. The whole loop at step 3 costs $p(2+3k)n^2 + o(n^2)$.
 - At step 4, the exponent has a bit length of n, therefore the exponentiation costs $3n^3 + n^2 + o(n^2)$.
 - The exponent at step 5a is $\prod_{j=1}^{p} e_j/e_i * F$, where $F = E_i^{-1} \mod e_i$; its bit length is p * k, therefore the exponentiation at step 5a costs $(3p * k 2)n^2 + p(n^2)$. Both of the exponentiations at steps 5d and 5(e)ii cost $(3(p-1)k-2)n^2 + o(n^2)$. We have also an inversion at step 5f which is equivalent to 20 multiplications of *n*-bit integers and costs $40n^2 + o(n^2)$.

Asymptotic behavior: $3n^3 + n^2 (42p - 1 + k * (3p^3 + 3p)) + o(n^2)$

• Memory Cost:

| _ | System | Parame | ters |
|---|--------|--------|-----------------------|
| | | | |

| Register names | Bits | Number of registers |
|----------------|------|---------------------|
| C_i | n | p |
| e_i | k | p |
| d_i | n | p |
| N | n | 1 |
| M_i | n | p |
| Subtotal | n | (3p+1) + pk bits |

68

- Accumulators

| Register names | Bits | Number of registers |
|----------------|-----------|-----------------------|
| e | p * k | 1 |
| a_i | p * k | p |
| E | (p-1) * k | 1 |
| F, T | n | 2 |
| Subtotal | 2n+k (| $(p^2 + 2p - 1)$ bits |

- Total memory cost: $n(3p+3) + k(p^2 + 3p - 1)$ bits

7.1.5 Comparison with classic RSA

Here we will compare the number of multiplications of n bit integers for batch RSA and p classic RSA decryptions, using a bit length k = 16 for public exponents e_i .

• Cost Estimation:

| | p=2 | p=4 | p=8 |
|-----------------------|-----------------|------------------|-------------------|
| RSA without CRT | $6n^3 + 2n^2$ | $12n^3 + 4n^2$ | $24n^3 + 8n^2$ |
| Speed-up for $n=1024$ | 1.0 | 1.0 | 1.0 |
| RSA with CRT | $3n^3/2 + 7n^2$ | $3n^3 + 14n^2$ | $6n^3 + 28n^2$ |
| Speed-up for $n=1024$ | 3.98 | 3.98 | 3.98 |
| Batch RSA | $3n^3 + 563n^2$ | $3n^3 + 3431n^2$ | $3n^3 + 25295n^2$ |
| Speed-up for $n=1024$ | 1.69 | 1.89 | 0.87 |

• Memory:

| | Total Memory | System Parameters | Accumulators |
|-----------------|----------------|-------------------|---------------|
| RSA without CRT | 3n bits | 3n bits | |
| n=1024 | 3072 bits | 3072 bits | |
| RSA with CRT | 8n bits | 11n/2 bits | 5n/2 bits |
| n=1024 | 8192 bits | 5632 bits | 2560 bits |
| Batch RSA (p=2) | 7n + 96 bits | 5n + 32 bits | 2n + 64 bits |
| n=1024, $k=16$ | 9360 bits | 7200 bits | 2160 bits |
| Batch RSA (p=4) | 11n + 256 bits | 9n + 64 bits | 2n + 192 bits |
| n=1024, $k=16$ | 15792 bits | 13376 bits | 2416 bits |
| Batch RSA (p=8) | 19n + 768 bits | 17n + 128 bits | 2n + 640 bits |
| n=1024 , $k=16$ | 29040 bits | 25728 bits | 3312 bits |

We see here that batch RSA has no advantages over RSA using the Chinese remainder theorem with a bit length of 1024. Besides the memory requirements are very high compared to p successive RSA decryptions. However it is possible to make use of some tricks in order to speed-up batch RSA.

7.2 How to increase efficiency?

If we want $A_0 = \prod_{i=1}^p C_i^{e/e_i} \mod N$ to really be the critical part of the algorithm, we have to keep the auxiliary computations cheap. Therefore, the exponents e_i have to be small. Nevertheless, it is possible to increase the speed of many computations by using special algorithms.

7.2.1Chinese remainder theorem

Like in other RSA-based algorithms, one can use the Chinese remainder theorem in order to divide by approximatively four the computations. We can implement it with Garner's algorithm.

7.2.2Computing multiple inverses

When we compute M_i back for i from 1 to p, we have to also compute an inverse:

$$M_{i} = \frac{A^{\alpha_{i}}}{C_{i}^{(\alpha_{i}-1)/e_{i}} * \prod_{j=1, j \neq i}^{p} C_{j}^{\alpha_{i}/e_{j}}} \bmod N$$

Let $T_i = C_i^{(\alpha_i - 1)/e_i} * \prod_{j=1, j \neq i}^p C_j^{\alpha_i/e_j}$. Instead of computing the T_i^{-1} independently, we can group the computations thanks to a trick due to Montgomery and increase the speed of batch RSA. Instead of computing p inverses, we just compute a single inverse and 3(p-1) multiplications.

7.2.3Computing multiple exponentiations

In many cases, we must evaluate the product of single exponentiations, for example:

$$A_0 = \prod_{i=1}^p C_i^{e/e_i} \bmod N$$

Or:

$$D_i = C_i^{(\alpha_i - 1)/e_i} * \prod_{j=1, j \neq i}^p C_j^{\alpha_i/e_j} \mod N$$

Instead of computing individually each exponentiation and then multiply them, we can group the computations again ([MOV97, p. 618]). Although we need a pre-computation stage which has an exponential asymptotic behavior, this is particularly cheap for the case of the D_i , because the basis stay the same, only the exponents change. Thus, we just have one pre-computation stage for all computations of the D_i .

Improving batch RSA: Montgomery's trick 7.3

This method allows to compute n inversions for the cost of a single inversion and some auxiliary multiplications. It is known as Montgomery's trick.

Description of Montgomery's Trick for multiple inversions 7.3.1

Algorithm 27 takes the integers to invert and the modulus as input, and computes the product of the inverted integers.

Algorithm 27: Montgomery's Trick for inversions

UNIT: MONTGOMERY; INPUT: A_1, \ldots, A_p, N ; OUTPUT: $Z_1 = A_1^{-1}, \ldots, Z_p = A_p^{-1} \mod N$;

1. $X_1 \leftarrow A_1$; 2. for *i* from 2 to *p* (a) $X_i \leftarrow X_{i-1} * A_i \mod N$; 3. $Z_1 \leftarrow X_p^{-1} \mod N$; 4. for *i* from *p* down to 2 (a) $Z_i \leftarrow X_{i-1} * Z_1 \mod N$; (b) $Z_1 \leftarrow Z_1 * A_i \mod N$; 5. return (Z_1, \ldots, Z_p)

7.3.2 Correctness of Montgomery's Trick

The values X_i computed at step 2a are the product of the input values A_j from 1 to *i*:

$$X_i = \prod_{j=1}^i A_j \bmod N$$

We compute then Z_1 at step 3 which is the product of all inverses:

$$Z_1 = \prod_{j=1}^p A_j^{-1} \bmod N$$

We multiply Z_1 with A_i at each iteration of the loop 4. Thus, at the *i*-th iterations, the value of Z_1 is:

$$Z_{1,i} = \prod_{j=1}^{i} A_j^{-1} \bmod N$$

This value is multiplied with $X_{i-1} = \prod_{j=1}^{i-1} A_j$ at step 4a. Therefore, we get:

$$Z_{i} = \prod_{j=1}^{i-1} A_{j} * \prod_{j=1}^{i} A_{j}^{-1} \mod N$$

Finally, $Z_i = A_i^{-1}$.

7.3.3 Performance analysis of Montgomery's Trick

We just compute a single inversion at step 3 and 3(p-1) multiplications (one multiplication pro iteration at step 2a and two multiplications pro iteration at steps 4a and 4b. Thus we compute an equivalent of 3(p-1) + 20 multiplications instead of an equivalent of 20p multiplications (p successive inversions).

- Notations: Let *n* be the bit length of N
- Cost estimation:
 - -p-1 multiplications for the first loop at step 2a
 - -2(p-1) multiplications for the second loop at steps 4a and 4b
 - 1 inversion (equivalent to 20 multiplications) at step 3.
 - Asymptotic behavior: $(6(p-1)+40)n^2 + o(n^2)$
- Memory Cost:

| _ | System Parame | ters | |
|---|----------------|-----------------------|---------------------|
| | Register names | Bits | Number of registers |
| | A_i | n | p |
| | Z_i | n | p |
| | Subtotal | | 2pn bits |
| _ | Accumulators | | |
| | Register names | Bits | Number of registers |
| | X_i | n | p |
| | Subtotal | | pn bits |

- Total memory cost: 3pn bits

7.4 Improving batch RSA: Shamir's Trick

7.4.1 Description of Shamir's Trick for multiple exponentiations

Just like the fast exponentiation method, the fast computation of multiple exponentiation orders the different exponents regarding their binary representation: the same two-powers are computed in the same time. This method is known as Shamir's Trick. As input we take p basis M_1, \ldots, M_p , p exponents e_1, \ldots, e_p and the modulus N. The binary representation of each exponent is known: $e_i = (e_{i,k-1} \ldots e_{i,0})_2$. We compute $C = M_1^{e_1} * \ldots * M_p^{e_p}$ faster than if we simply perform consecutive exponentiations.

7.4.2 Explanations

At the pre-computation stage, we compute all possible products of the M_i . In fact, we establish a bijection b defined by:

$$b: i = (i_{p-1} \dots i_0)_2 \longmapsto \prod_{j=0}^{p-1} M_j^{i_j} \mod N$$

where $i_j \in \{0, 1\}$.

At the evaluation stage, we proceed like in the left-to-right exponentiation method, but instead of multiplying with a single basis M, we do have many possibilities and must must choose the right basis. We want to compute:

$$C = \prod_{i=1}^{k} M_i^{e_i} \bmod N$$

72

Algorithm 28: Shamir's Trick for multiple exponentiations

UNIT: SHAMIR; INPUT: $M_1, \ldots, M_p, e_1, \ldots, e_p, N$; OUTPUT: $C = M_1^{e_1} * \ldots * M_p^{e_p}$;

1. for *i* from 0 to $(2^p - 1)$ (a) $G_i \leftarrow M_1^{i_1} * \ldots * M_p^{i_p} \mod N$ where $i = (i_p \ldots i_1)_2$; 2. $A \leftarrow 1$; 3. for *i* from (k - 1) down to 0 (a) $C \leftarrow C * C \mod N$; (b) $C \leftarrow C * G_{b_i} \mod N$ where $b_i = (e_{p,i} \ldots e_{1,i})_2$; 4. return(C);

Let's use the binary representation of the exponents e_i :

$$C = \prod_{i=1}^{k} M_{i}^{\sum_{j=0}^{p-1} e_{i,j} * 2^{j}} \mod N$$

Now we can transform the sum inside the exponents into a product:

$$C = \prod_{i=1}^{k} \prod_{j=0}^{p-1} M_i^{e_{i,j} * 2^j} \mod N$$

And we can swap the two products:

$$C = \prod_{j=0}^{p-1} \left(\prod_{i=1}^{k} M_i^{e_{i,j}} \right)^{2^j} \mod N$$

The term inside the parenthesis is the basis we must choose at each step; this products have already been computed in the pre-computation step. They are simply:

$$C = \prod_{j=0}^{p-1} b\left(E\right)^{2^j} \bmod N$$

where $E = (e_{1,j} \dots e_{k,j})_2$. The two-power exponentiation is simply done by successive squaring at each step.

7.4.3 Performance analysis of Shamir's Trick

• Notations:

Let k be the bit length of $max(e_i)$ and n the bit length of N.

- Cost Estimation:
 - at step 1a, there are j integers i having i times the bit one in their binary representation. Therefore the whole pre-computation stage costs $\sum_{j=0}^{p} {p \choose j} * j = p * 2^{p-1}$ multiplications of n-bit integers.

- (k-1) square computations of *n*-bit integers are computed at step 3a: at the beginning, A = 1 so we practically don't have any squaring when i = k 1.
- $-k * (1 1/2^p)$ multiplications of *n*-bit integers are computed at step 3b: the probability of having $G_{b_i} = 1$ is $1/2^p$. In this case, we don't have to compute any multiplication. Thus, we compute on average $1 - 1/2^p$ multiplications at each step of the loop.

Asymptotic behavior: $\underbrace{p*2^{p}*n^{2}}_{pre-computation} + \underbrace{\left(k(4-1/2^{p-1})-2\right)n^{2}}_{evaluation} + o(n^{2})$

If k = n, then we get:

$$\underbrace{p*2^{p}*n^{2}}_{pre-computation} + \underbrace{n^{3}*\left(4-1/2^{p-1}\right)-2*n^{2}}_{evaluation} + o(n^{2})$$

• Memory Cost:

– System Parameters

| System 1 an anno | 0010 | |
|------------------|------|---------------------|
| Register names | Bits | Number of registers |
| M_i | n | p |
| e_i | k | p |
| C, N | n | 2 |
| Subtotal | n | k(p+2) + kp bits |
| | n(2 | (p+2) bits if $k=n$ |

- Accumulators

| Register names | Bits | Number of registers |
|----------------|------|---------------------|
| G_i | n | 2^p |
| Subtotal | | $2^p n$ bits |

- Total memory cost: $n(p+2+2^p) + kp$ bits $n(2p+2+2^p)$ bits if k = n

7.5 Batch RSA using Shamir's and Montgomery's Tricks

7.5.1 Description of improved Batch RSA

In the first version of batch RSA, we needed to compute the following formula in order to recover plain texts:

$$M_{i} = \frac{\left(\prod_{j=1}^{p} C_{j}^{e/e_{j}}\right)^{\alpha_{i}/e}}{C_{i}^{(\alpha_{i}-1)/e_{i}} * \prod_{j=1, j \neq i}^{p} C_{j}^{\alpha_{i}/e_{j}}} \mod N$$

We can see here that we need to compute two products of p exponentiations and 1 inversions for each M_i . Thanks to the Shamir's trick, we can speed-up the multiple exponentiation process for each M_i , and, if we wait and compute all inversions in the same time, we can make use of Montgomery's trick to do it faster. Please note that we just need one pre-computation for the Shamir's trick, although this unit is called (p + 1) times, because the basis stay always the same. This does not appear in algorithm 29 to make it easier to read.

Algorithm 29: Improved Batch RSA

UNIT: I_BATCH_RSA; INPUT: $C_1, ..., C_p, e_1, ..., e_p, d_1, ..., d_p, N$; OUTPUT: $M_1, ..., M_p$ verifying $C_i = M_i^{e_i} \mod N$;

```
1. e \leftarrow e_1; d \leftarrow d_1;
2. for i from 2 to p
     (a) e \leftarrow e * e_i;
     (b) d \leftarrow d * d_i;
3. for i from 1 to p
     (a) E_i \leftarrow e/e_i;
4. A \leftarrow \text{SHAMIR}(C1, \ldots, C_p, E_1, \ldots, E_p, N);
5. A \leftarrow A^d \mod N;
6. for i from 1 to p
     (a) F \leftarrow E_i^{-1} \mod e_i;
(b) a \leftarrow E_i * F \mod N;
      (c) M_i \leftarrow A^a \mod N;
     (d) g_i \leftarrow a - 1 \mod N;
      (e) g_i \leftarrow g_i/e_i;
      (f) for j from 1 to p
              i. if j \neq i then g_j \leftarrow a/e_j;
     (g) T_i \leftarrow \text{SHAMIR}(C1, \ldots, C_p, g_1, \ldots, g_p, N);
7. T_1, \ldots, T_p \leftarrow \text{MONTGOMERY}(T_1, \ldots, T_p);
8. for i from 1 to p
     (a) M_i \leftarrow M_i * T_i \mod N;
9. return(M_1, \ldots, M_p);
```

7.5.2 Performance analysis of improved Batch RSA

• Notations:

n is the bit length of the public modulus N, p the number of batched decryptions and k the bit length of the greatest e_i .

- Cost Estimation:
 - In the first loop at step 2b, (p-1) multiplications in \mathbb{Z} of *n*-bit integers are computed; the operations at steps 3a and 2a can be neglected since the involved integers have a bit length of at most p * k.
 - The exponents E_i have a bit length of (p-1)k, therefore the Shamir's trick at step 4 costs

$$\underbrace{p*2^{p}*n^{2}}_{pre-computation} + \underbrace{\left((p-1)k(4-1/2^{p-1})-2\right)n^{2}}_{evaluation} + o(n^{2})$$

We don't need any further pre-computation stage when the Shamir's trick is called again.

- At step 5, we compute the batched decryption; this operation should be the critical one for the whole algorithm and costs $3n^3 + n^2$.
- The inversions $E_i^{-1} \mod e_i$ are equivalent to 20 multiplications of k-bit integers, since these operations do not involve n^2 , we can neglect them.
- The bit length of $F = E_i^{-1} \mod e_i$ is k and the bit length of $E_i = \prod_j = 1, j \neq i^p e_i$ is (p-1)k. Therefore, the bit length of $a = F * E_i$ is pk and the exponentiation at step 6c costs $(3pk-2)n^2 + o(n^2)$.
- The exponents g_i have a bit length of (p-1)k and the Shamir's trick at step 6g only costs $((p-1)k(4-1/2^{p-1})-2)n^2+o(n^2)$ because we don't need any pre-computation stage. However, this unit is called p times. Montgomery's trick allows us to compute all p inversions with a cost of $(6(p-1)+40)n^2+o(n^2)$. In the last loop, we compute p multiplications of n-bit integers: it costs $2pn^2 + o(n^2)$.

Asymptotic behavior:

$$3n^{3} + n^{2} \left(32 + p \left(2^{p} + 7\right) + k * \left(-4 + 1/2^{p-1} + p^{2} * \left(7 - 1/2^{p-1}\right)\right)\right) + o(n^{2})$$

• Memory Cost:

| _ | System Parame | eters | |
|---|----------------|-------|---------------------|
| | Register names | Bits | Number of registers |
| | C_i | n | p |
| | e_i | k | p |
| | d_i | n | p |
| | N | n | 1 |
| | M_i | n | p |
| | Subtotal | n | (3p+1) + pk bits |

| – Accumulators | | |
|----------------|-----------|---------------------------------|
| Register names | Bits | Number of registers |
| e | pk | 1 |
| d | n | 1 |
| a | pk | 1 |
| E_i | (p-1)k | p |
| F | k | 1 |
| g_i | pk | p |
| T_i | n | p |
| SHAMIR | $n * 2^p$ | 1 |
| MONTGOMERY | pn | 1 |
| Subtotal | n(2p+1- | $(+2^p) + k(2p^2 + p + 1)$ bits |

- Total memory cost: $n(5p + 2 + 2^p) + k(2p^2 + p + 1)$ bits

7.6 Batch RSA using the Chinese remainder theorem

7.6.1 Description of Batch RSA using the Chinese remainder theorem

We can also use the Chinese remainder theorem in order to speed up batch RSA: instead of decrypting the messages modulo N, we can decrypt them modulo P and Q with algorithm 29, assuming that N = PQ, and then recover the messages thanks to Garner's algorithm.

Algorithm 30: Batch RSA using the Chinese remainder theorem

UNIT: BATCH_RSA_CRT; INPUT: $C_1, \ldots, C_p, e_1, \ldots, e_p, d_1, \ldots, d_p, N, P, Q, (P_inv_Q);$ OUTPUT: M_1, \ldots, M_p verifying $C_i = M_i^{e_i} \mod N;$

1. for *i* from 1 to *p* (a) $C_{i,P} \leftarrow C_i \mod P$; (b) $e_{i,P} \leftarrow e_i \mod P - 1$; (c) $d_{i,P} \leftarrow d_i \mod P - 1$; (d) $C_{i,Q} \leftarrow C_i \mod Q$; (e) $e_{i,Q} \leftarrow e_i \mod Q - 1$; (f) $d_{i,Q} \leftarrow d_i \mod Q - 1$; 2. $M_{1,P}, \dots, M_{p,P} \leftarrow \text{L-BATCH-RSA}(C_{1,P}, \dots, C_{p,P}, e_{1,P}, \dots, e_{p,P}, d_{1,P}, \dots, d_{p,P}, P)$; 3. $M_{1,Q}, \dots, M_{p,Q} \leftarrow \text{L-BATCH-RSA}(C_{1,Q}, \dots, C_{p,Q}, e_{1,Q}, \dots, e_{p,Q}, d_{1,Q}, \dots, d_{p,Q}, Q)$; 4. for *i* from 1 to *p* (a) $M_i \leftarrow \text{GARNER}(M_{i,P}, M_{i,Q}, P, Q, (P_-inv_-Q), N)$; 5. return (M_1, \dots, M_p) ;

7.6.2 Performances of Batch RSA using the Chinese remainder theorem

• Notations: Let *n* be the bit length of N

- Cost estimation:
 - In the first loop (step 1), we compute 6 modular reductions. The reductions of e_i can be neglected, because they cost less than n^2 . The other reductions cost each $n^2/4 + o(n^2)$. Therefore, the whole loop costs $p * n^2 + o(n^2)$.
 - Step 2 and 3 both call the improved batch RSA unit with moduli whose bit length is n/2. Therefore we have here a total cost of:

$$3n^{3}/4 + n^{2}/2\left(32 + p\left(2^{p} + 7\right) + k * \left(-4 + 1/2^{p-1} + p^{2} * \left(7 - 1/2^{p-1}\right)\right)\right) + o(n^{2})$$

- At each of the p steps 4a, we execute Garner's algorithm which costs $5n^2/2 + o(n^2)$.
- Asymptotic behavior:

$$3n^{3}/4 + n^{2}/2\left(32 + p\left(2^{p} + 21/2\right) + k * \left(-4 + 1/2^{p-1} + p^{2} * \left(7 - 1/2^{p-1}\right)\right)\right) + o(n^{2})$$

• Memory Cost:

– System Parameters

| v | | |
|----------------|------|---------------------|
| Register names | Bits | Number of registers |
| C_i | n | p |
| e_i | k | p |
| d_i | n | p |
| N | n | 1 |
| M_i | n | p |
| Subtotal | n | (3p+1) + pk bits |

– Accumulators

| recumulations | | |
|--------------------|--------------------------------|---------------------|
| Register names | Bits | Number of registers |
| $C_{i,P}, C_{i,Q}$ | n/2 | 2p |
| $e_{i,P}, e_{i,Q}$ | k | 2p |
| $d_{i,P}, d_{i,Q}$ | n/2 | 2p |
| I_BATCH_RSA | $n(p+1/2+2^{p-1})+k(2p^2+p+1)$ | 1 |
| GARNER | n/2 | 1 |
| Subtotal | $n(3p+1/2+2^{p-1})+k(2p^2-$ | +3p+1) bits |

- Total memory cost: $n(6p+3/2+2^{p-1}) + k(2p^2+4p+1)$ bits

7.7 Comparison with classic RSA

| | p=2 | p=4 | p=8 |
|-----------------------|---------------------|-------------------|------------------------|
| RSA without CRT | $6n^3 + 6n^2$ | $12n^3 + 12n^2$ | $24n^3 + 24n^2$ |
| Speed-up for $n=1024$ | 1.0 | 1.0 | 1.0 |
| RSA without CRT | $3n^3/2 + 9n^2$ | $3n^3 + 18n^2$ | $6n^3 + 36n^2$ |
| Speed-up for $n=1024$ | 3.98 | 3.98 | 3.98 |
| Batch RSA | $3n^3/4 + 421n^2/2$ | $3n^3/4 + 918n^2$ | $3n^3/4 + 74081n^2/16$ |
| Speed-up for $n=1024$ | 6.28 | 7.29 | 4.55 |

• Cost Estimation, k=16 bits:

Batch RSA appears to be faster than RSA using the Chinese remainder theorem, this is particularly clear for p = 4, but the speed-up becomes smaller when p is too big. We used

7.7. COMPARISON WITH CLASSIC RSA

a bit length k = 16 for the public exponents e_i , but [BS02] recommends smaller values for k, hence achieving a greater speed-up. Nevertheless, in a real situation, the decryption device would have to wait for decryption requests, which was not taken into account in our estimation and would generate idle cycles. The batching strategy is very important: it appears here that batching four decryptions twice is faster than batching eight decryptions once. Depending on the decryption requests frequency and the decryption speed, we have to make a good batching choice.

• Memory:

| | Total Memory | System Parameters | Accumulators |
|-----------------|--------------------|-------------------|--------------------|
| RSA without CRT | 4n bits | 4n bits | |
| n=1024 | 4096 bits | 4096 bits | |
| RSA with CRT | 8n bits | 11n/2 bits | 5n/2 bits |
| n=1024 | 8192 bits | 5632 bits | 2560 bits |
| Batch RSA (p=2) | 31n/2 + 272 bits | 7n + 32 bits | 17n/2 + 240 bits |
| n=1024, $k=16$ | 16144 bits | 7200 bits | 8944 bits |
| Batch RSA (p=4) | 67n/2 + 784 bits | 13n + 64 bits | 41n/2 + 720 bits |
| n=1024, $k=16$ | 35088 bits | 13376 bits | 21712 bits |
| Batch RSA (p=8) | 355n/2 + 2576 bits | 25n + 128 bits | 305n/2 + 2448 bits |
| n=1024 , $k=16$ | 184336 bits | 25728 bits | 158608 bits |

Using some tricks, batching RSA decryptions can be faster than p successive RSA decryptions, at the expense of memory consumption. Therefore this algorithm is well designed for environments having a huge memory capacity, like SSL servers.

As shown in [SB01], it is possible to modify the architecture of an Apache web server in order to make use of batch RSA, hence improving performances of SSL handshakes.

Chapter 8

Conclusion

We compare the different decryption schemes that have been described in terms of speed and memory. However, some algorithms have very specific application domains, therefore we must not forget what the characteristics, downsides and applications of each algorithm are.

Contents

| 8.1 | Speed comparisons | 81 |
|-----|--------------------|----|
| 8.2 | Memory comparisons | 82 |

8.1 Speed comparisons

In the following, we compare the different algorithms of the paper, namely RSA without and with the Chinese remainder theorem, rebalanced RSA, Multi-Prime RSA, Multi-Power RSA and batch RSA. Please note that these algorithms have different application domains, and one must consider the downsides of each of the algorithms. Although rebalanced RSA seems to be the fastest algorithm, its encryption stage is much slower, therefore it is only designed for signing or decrypting purposes. Batch RSA is not as fast as Multi-Prime and Multi-Power, but in fact, it can be combined with these algorithms, enhancing their speed at the expense of the memory. RSA modulo P^2Q is not only fast, but also does not have any important downside.

| | Speed | Speed-up for $n=1024$ |
|------------------------------|----------------------------|-----------------------|
| RSA without CRT | $3n^3 + n^2$ | 1.0 |
| RSA with CRT | $3n^3/4 + 7n^2/2 + o(n^2)$ | 3.98 |
| Batch RSA, $p = 4, k = 16$ | $3n^3/4 + 918n^2 + o(n^2)$ | 7.29 |
| RSA modulo PQR | $n^3/3 + 19n^2/3 + o(n^2)$ | 8.84 |
| RSA modulo P^2Q , $k = 16$ | | 12.06 |
| Rebalanced RSA, $k = 160$ | $n^2(3k/2+2) + o(n^2)$ | 12.70 |

| | Total Memory | System Parameters | Accumulators |
|-------------------|------------------|-------------------|------------------|
| RSA without CRT | 4n bits | 4n bits | 0 bit |
| n=1024 bits | 4096 bits | 4096 bits | 0 bits |
| RSA with CRT | 8n bits | 11n/2 bits | 5n/2 bits |
| n=1024 bits | 8192 bits | 5632 bits | 2560 bits |
| Rebalanced RSA | 7n+2k bits | 9n/2 + 2k bits | 5n/2 bits |
| n=1024, k=160 | 7200 bits | 4640 bits | 2560 bits |
| RSA modulo PQR | 26n/3 bits | 17n/3 bits | 3n bits |
| n=1024 bits | 8875 bits | 5803 bits | 3072 bits |
| RSA modulo P^2Q | 25n/3 bits | 5n bits | 10n/3 bits |
| n=1024 bits | 8534 bits | 5120 bits | 3414 bits |
| Batch RSA (p=4) | 67n/2 + 784 bits | 13n + 64 bits | 41n/2 + 720 bits |
| n=1024, $k=16$ | 35088 bits | 13376 bits | 21712 bits |

8.2 Memory comparisons

Although the rebalanced RSA decryption scheme requires less memory than the other algorithms, we must not forget that its encryption scheme is very slow. For 4 batched decryptions, batch RSA requires about three times as much memory as the other decryption schemes. Its huge memory requirement make it only useful in environments without memory constraints, like SSL servers. Multi-Prime and Multi-Power RSA have about the same memory requirements, Multi-Power RSA being slightly better.

Bibliography

- [BG00] D. Boneh and G.Durfee. Cryptoanalysis of RSA with private key d less than $n^{0.292}$. *IEEE Trans. on Info. Th.*, 46(4):1339–1349, Jul. 2000.
- [BS02] D. Boneh and H. Shacham. Fast variants of RSA. *Cryptobytes*, 5(1):1–9, 2002.
- [Buc01] J. Buchmann. Introduction to Cryptography. Springer-Verlag, 2001.
- [CDL⁺00] S. Cavallar, B. Dodson, A. K. Lensra, W. Lioen, P. Montgomery, B. Murphy, H. Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffet, C. Putnam, and P. Zimmermann. Factorization of a 512-bit RSA-modulus. In *Proceedings of Eurocrypt 2000*, volume 1807, pages 1–11. Springer-Verlag, 2000.
- [CHLS97] T. Collins, D. Hopkins, S. Langford, and M. Sabin. Public Key Cryptographic Apparatus and Method. US Patent #5, 1997.
- [Fia89] A. Fiat. Batch RSA. In Proceedings of Crypto '89, volume 435, pages 175–185. Springer-Verlag, 1989.
- [Lab02] RSA Laboratories. PKCS#1 v. 2.1: RSA cryptography standard. June 2002.
- [MOV97] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook for applied cryptography*. CRC Press, 1997.
- [RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communication of the ACM*, 21(2):120–126, Feb. 1978.
- [SB01] H. Shacham and D. Boneh. Improving SSL handshake performance via batching. In ed. D. Naccache, editor, *Proceedings of RSA 2001*, volume 2020, pages 28–43. Springer-Verlag, 2001.
- [SSW93] R. Silverman and Jr. S. Wagstaff. A practical analysis of the elliptic curve factoring algorithm. Math. Comp., 61(203):445–462, Jul. 1993.
- [Tak98] T. Takagi. Fast RSA-type cryptosystem modulo p^kq . In Advances in cryptology, volume 1462, pages 318–326. Springer-Verlag, 1998.
- [Wie90] M. Wiener. Cryptanalysis of short RSA secret exponents. IEEE Trans. on Info. Th., 36(3):553–558, 1990.