

► **Agenda:** explore **RSA** [8] via

1. an “in theory”, i.e., design-oriented perspective, and
2. an “in practice”, i.e., implementation-oriented perspective,
 - 2.1 multi-precision (or “big”) integer arithmetic,
 - 2.2 exponentiation,
 - 2.3 modular multiplication.

► **Caveat!**

~ 2 hours $\Rightarrow$ introductory, and (very) selective (versus definitive) coverage.

Part 1: in theory (1)

► Recall: the RSA primitive can be used as

► an asymmetric encryption scheme

1. generation of a public and private key pair: given a security parameter λ

$$\text{RSA.KeyGen}(\lambda) = \left\{ \begin{array}{l} 1 \text{ select random } \frac{\lambda}{2}\text{-bit primes } p \text{ and } q \\ 2 \text{ compute } N = p \cdot q \\ 3 \text{ compute } \Phi(N) = (p-1) \cdot (q-1) \\ 4 \text{ select random } e \in \mathbb{Z}_N^* \text{ such that } \gcd(e, \Phi(N)) = 1 \\ 5 \text{ compute } d = e^{-1} \pmod{\Phi(N)} \\ 6 \text{ return public key } (N, e) \text{ and private key } (N, d) \end{array} \right.$$

2. encryption of a plaintext $m \in \mathbb{Z}_N^*$:

$$c = \text{RSA.Enc}((N, e), m) = m^e \pmod{N}$$

3. decryption of a ciphertext $c \in \mathbb{Z}_N^*$:

$$m = \text{RSA.Dec}((N, d), c) = c^d \pmod{N}$$

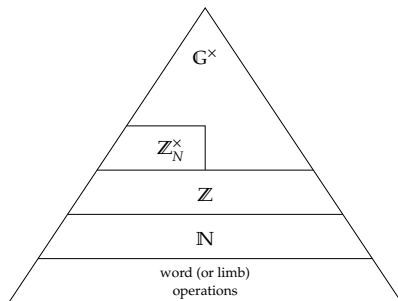
► a digital signature scheme: (very) roughly, we just set

$$\begin{array}{lll} \text{RSA.Sig} & \simeq & \text{RSA.Dec} \\ \text{RSA.Ver} & \simeq & \text{RSA.Enc} \end{array}$$

Part 2: in practice (1)

► Challenge:

- given a functionality “stack”, i.e.,



bridge gap between what we have (bottom) and want (top),

- an **implementation strategy** for doing so must consider many

- goals : parameter set, functionality, ...
- metrics : latency, throughput, memory footprint, ...
- constraints : hardware versus software, data-path width, ...
-

Part 2.1: in practice (1)

► Problem:

- we want to represent and perform operations on integers, i.e., elements of

$$\mathbb{Z} = \{\dots, -3, -2, -1, 0, +1, +2, +3, \dots\},$$

- a micro-processor equipped with a w -bit word size approximates $\mathbb{Z}$ using an appropriate data type; e.g., in C we get

$$\begin{array}{llllll} w & = & 8 & \rightsquigarrow & \text{char} & \simeq \text{int8_t} \mapsto \{ -2^7, \dots, 0, \dots, +2^7 - 1 \} \\ w & = & 16 & \rightsquigarrow & \text{short} & \simeq \text{int16_t} \mapsto \{ -2^{15}, \dots, 0, \dots, +2^{15} - 1 \} \\ w & = & 32 & \rightsquigarrow & \text{int} & \simeq \text{int32_t} \mapsto \{ -2^{31}, \dots, 0, \dots, +2^{31} - 1 \} \end{array}$$

- the magnitude of some $x \in \mathbb{Z}$ can be much larger than a w -bit word.

► Solution:

1. a data structure to represent x , and
2. algorithms to operate on instances of said data structure.

Part 2.1: in practice (2)

► Idea:

1. represent x using a base- b expansion

$$\hat{x} = \langle \hat{x}_0, \hat{x}_1, \dots, \hat{x}_{n-1} \rangle$$

$$\mapsto x$$

$$= \pm \sum_{i=0}^{n-1} \hat{x}_i \cdot b^i$$

where each $\hat{x}_i \in X = \{0, \dots, b-1\}$,

2. select $b = 2^w$ such that

- each $\hat{x}_i$ is termed a **limb**,
- we can represent an $x \gg 2^w$, and
- digits of x can be dealt with conveniently by the processor.

Part 2.1: in practice (3)

► Idea:

1. represent $x \in \{0, 1, \dots, 2^w - 1\}$ using an instance of

```
typedef uint32_t limb_t;
```

2. represent $x \in \mathbb{N}$ using an array

```
limb_t x[ l_x ]
```

or a pointer to such an array, e.g.,

```
limb_t* x
```

plus an associated length l_x .

3. represent $x \in \mathbb{Z}$ using an instance of

Listing

```
1 typedef struct __mpz_t {  
2     limb_t d[ MPZ_LIMB_MAX ];  
3  
4     int     l;  
5     int     s;  
6 } mpz_t;
```

where

- $x.d$ is a fixed-size array of limbs representing the magnitude of x ,
- $x.l$ is the number limbs used within $x.d$, and
- $x.s$ is the sign of x .

Part 2.1: in practice (5)

Algorithms for $x \in \{0, 1, \dots, 2^w - 1\}$

- ▶ **Step #1:** limb-focused operations, e.g., “multiply-accumulate with carry”.

- ▶ the idea is to support computation of

$$r_1 \cdot 2^w + r_0 = t \leftarrow e \cdot f + g + h,$$

- ▶ for $b = 2^w$, the result r has at most $2 \cdot w$ bits because

$$(2^w - 1) \cdot (2^w - 1) + (2^w - 1) + (2^w - 1) = 2^{2w} - 1 < 2^{2 \cdot w},$$

Part 2.1: in practice (5)

Algorithms for $x \in \{0, 1, \dots, 2^w - 1\}$

- **Step #1:** limb-focused operations, e.g., “multiply-accumulate with carry”.

Listing

```
1 #define LIMB_MUL2(r_1,r_0,e,f,g,h) {
2     dlimb_t __t = ( dlimb_t )( e ) *
3                   ( dlimb_t )( f ) +
4                   ( dlimb_t )( g ) +
5                   ( dlimb_t )( h ) ;
6
7     r_0 =          ( limb_t )( __t >> 0 );
8     r_1 =          ( limb_t )( __t >> BITSOF( limb_t ) );
9 }
```

Part 2.1: in practice (5)

Algorithms for $x \in \{0, 1, \dots, 2^w - 1\}$

- **Step #1:** limb-focused operations, e.g., “multiply-accumulate with carry”.

Listing

```
1 #define LIMB_MUL2(r_1, r_0, e, f, g, h) {      \
2     asm( "movl %2,%%eax ; mull %3             \
3         addl %4,%%eax ; adcl $0,%%edx ; \
4         addl %5,%%eax ; adcl $0,%%edx ; \
5         movl %%eax,%0 ; movl %%edx,%1 ;" \
6         : "=&g" (r_0), "=&g" (r_1)          \
7         : "1" (e), "r" (f),                  \
8         "r" (g), "0" (h)                      \
9         : "%eax", "%edx", "cc"                ); \
10 }
11 }
```

Part 2.1: in practice (7)

Algorithms for $x \in \mathbb{N}$

► **Step #2:** $\mathbb{N}$ -focused operations, e.g., “long multiplication”.

► the idea is to support computation of

$$\begin{array}{rcl}
 x & = & 6 2 3 \\
 y & = & 5 6 7 \quad \times \\
 p_0 = 7 \cdot 3 & = & 2 1 \\
 p_1 = 7 \cdot 2 & = & 1 4 \\
 p_2 = 7 \cdot 6 & = & 4 2 \\
 p_3 = 6 \cdot 3 & = & 1 8 \\
 p_4 = 6 \cdot 2 & = & 1 2 \\
 p_5 = 6 \cdot 6 & = & 3 6 \\
 p_6 = 5 \cdot 3 & = & 1 5 \\
 p_7 = 5 \cdot 2 & = & 1 0 \\
 p_8 = 5 \cdot 6 & = & 3 0 \\
 c & = & 0 1 1 1 0 0 \\
 r & = & 3 5 3 2 4 1
 \end{array}$$

► the result r has at most $l_x + l_y$ limbs.

Part 2.1: in practice (7)

Algorithms for $x \in \mathbb{N}$

- **Step #2:** $\mathbb{N}$ -focused operations, e.g., “long multiplication”.

Algorithm ($\mathbb{N}$ -MUL)

Input: Two unsigned, base- b integers x and y

Output: An unsigned, base- b integer $r = x \cdot y$

```
1  $l_x \leftarrow |x|, l_y \leftarrow |y|, l_r \leftarrow l_x + l_y$ 
2  $r \leftarrow 0$ 
3 for  $j = 0$  upto  $l_y - 1$  step  $+1$  do
4    $c \leftarrow 0$ 
5   for  $i = 0$  upto  $l_x - 1$  step  $+1$  do
6      $u \cdot b + v = t \leftarrow y_j \cdot x_i + r_{j+i} + c$ 
7      $r_{j+i} \leftarrow v$ 
8      $c \leftarrow u$ 
9   end
10   $r_{j+l_x} \leftarrow c$ 
11 end
12 return  $r$ 
```

Listing

```
1 void mpn_mul( limb_t* r, const limb_t* x, int l_x,
2                const limb_t* y, int l_y ) {
3
4   int l_r = l_x + l_y;
5
6   limb_t R[ l_r ], c;
7
8   memset( R, 0, l_r * sizeof( limb_t ) );
9
10  for( int j = 0; j < l_y; j++ ) {
11    c = 0;
12
13    for( int i = 0; i < l_x; i++ ) {
14      limb_t d_y = y[ j ];
15      limb_t d_x = x[ i ];
16      limb_t d_R = R[ j + i ];
17
18      LIMB_MUL2( c, R[ j + i ], d_y, d_x, d_R, c );
19    }
20
21    R[ j + l_x ] = c;
22  }
23
24  memcpy( r, R, l_r * sizeof( limb_t ) );
25 }
```

Part 2.1: in practice (9)

Algorithms for $x \in \mathbb{Z}$

► Step #3: $\mathbb{Z}$ -focused operations.

Algorithm ($\mathbb{Z}$ -MUL)

Input: Two signed, base- b integers x and y

Output: A signed, base- b integer $r = x \cdot y$

```
1 if  $x < 0$  and  $y \geq 0$  then
2   |  $r \leftarrow -\mathbb{N}\text{-MUL}(\text{abs}(x), \text{abs}(y))$ 
3 end
4 else if  $x \geq 0$  and  $y < 0$  then
5   |  $r \leftarrow -\mathbb{N}\text{-MUL}(\text{abs}(x), \text{abs}(y))$ 
6 end
7 else if  $x \geq 0$  and  $y \geq 0$  then
8   |  $r \leftarrow +\mathbb{N}\text{-MUL}(\text{abs}(x), \text{abs}(y))$ 
9 end
10 else if  $x < 0$  and  $y < 0$  then
11   |  $r \leftarrow +\mathbb{N}\text{-MUL}(\text{abs}(x), \text{abs}(y))$ 
12 end
13 return  $r$ 
```

Listing

```
1 void mpz_mul( mpz_t* r, const mpz_t* x,
2               const mpz_t* y ) {
3   if( x->s == y->s ) {
4     MPZ_MUL( r, x, y, MPZ_SIGN_POS );
5   }
6   else {
7     MPZ_MUL( r, x, y, MPZ_SIGN_NEG );
8   }
9 }
```

Part 2.1: in practice (10)

Algorithms for $x \in \mathbb{Z}$

► Step #3: $\mathbb{Z}$ -focused operations.

Listing

```
1 #define MPZ_MUL(r,x,y,s_r) { \
2     int l_r = (x)->l + (y)->l; \
3     \
4     mpn_mul( (r)->d, (x)->d, (x)->l, \
5              (y)->d, (y)->l ); \
6     \
7     (r)->l = mpn_lop( (r)->d, l_r ); \
8     (r)->s = s_r; \
9 }
```

Listing

```
1 int mpn_lop( const limb_t* x, int l_x ) {
2     while( ( l_x > 1 ) && ( x[ l_x - 1 ] == 0 ) ) {
3         l_x--;
4     }
5
6     return l_x;
7 }
```

Part 2.2: in practice (1)

► Problem:

- given the **base** $x \in \mathbb{Z}_N^\times$ and the **exponent** $y \in \mathbb{Z}$, we want to compute $r = x^y \pmod{N}$,
- we could do so via

$$r = x^y = \underbrace{x \times x \times \cdots \times x}_{y \text{ terms}} \pmod{N},$$

but this is $O(y)$ and the magnitude of y can be very large.

► Solution:

- algorithms for efficient

multiplicative group $\mathbb{G}^\times$	$\leadsto$	exponentiation	x^y
additive group $\mathbb{G}^+$	$\leadsto$	scalar multiplication	$[y]x$

Part 2.2: in practice (2)

- ▶ **Idea:** consider some $\mathbb{G}^\times$.
 - ▶ exponentiation *means* repeated application of the group operation, i.e.,

$$x^y = \underbrace{x \times x \times \cdots \times x}_{y \text{ terms}},$$

so if $y = 14_{(10)}$ we have

$$x^y = x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x.$$

- ▶ expressing y in base-2, we can rewrite this as

$$\begin{aligned} x^y &= x^{\sum_{i=0}^{n-1} y_i \cdot 2^i} \\ &= x^{y_{n-1} \cdot 2^{n-1} + \cdots + y_1 \cdot 2^1 + y_0 \cdot 2^0} \\ &= x^{y_{n-1} \cdot 2^{n-1}} \times \cdots \times x^{y_1 \cdot 2^1} \times x^{y_0 \cdot 2^0} \end{aligned}$$

Part 2.2: in practice (2)

► **Idea:** consider some $\mathbb{G}^\times$.

► given $y = 14_{(10)} = 1110_{(2)}$ we can see that

$$\begin{aligned} x^y &= x^{y_0 \cdot 2^0} \times x^{y_1 \cdot 2^1} \times x^{y_2 \cdot 2^2} \times x^{y_3 \cdot 2^3} \\ &= x^{0 \cdot 2^0} \times x^{1 \cdot 2^1} \times x^{1 \cdot 2^2} \times x^{1 \cdot 2^3} \\ &= x^0 \times x^2 \times x^4 \times x^8 \\ &= x^{14} \end{aligned}$$

► given $y = 14_{(10)} = 1110_{(2)}$ we can see that

$$\begin{aligned} x^y &= x^{y_0} \times (x^{y_1} \times (x^{y_2} \times (x^{y_3} \times (1)^2)^2)^2)^2 \\ &= x^0 \times (x^1 \times (x^1 \times (x^1 \times (1)^2)^2)^2)^2 \\ &= x^0 \times (x^1 \times (x^1 \times (x^1 \times 1)^2)^2)^2 \\ &= x^0 \times (x^1 \times (x^1 \times x^2)^2)^2 \\ &= x^0 \times (x^1 \times x^6)^2 \\ &= x^0 \times (x^7)^2 \\ &= x^0 \times x^{14} \\ &= x^{14} \end{aligned}$$

via application of **Horner's rule**.

Part 2.2: in practice (3)

- ▶ **Option #1:** left-to-right *binary* exponentiation.
 - ▶ evaluate the Horner expansion step-by-step in an “inside-out” order ,
 - ▶ express y in base-2, i.e., extract a 1-bit digit d from y in each step,
 - ▶ trade-off time in favour of space,
 - ▶ apply

$$r \leftarrow \begin{cases} r^2 & \text{if } d = 0 \\ r^2 \times x & \text{if } d = 1 \end{cases}$$

so as to *accumulate* the result.

- **Option #1:** left-to-right *binary* exponentiation.

Algorithm (1EXP-L2R-BINARY)

Input: A group element $x \in \mathbb{G}^\times$, a base-2 integer $0 \leq y < n$

Output: The group element $r = x^y \in \mathbb{G}^\times$

```
1  $r \leftarrow 1$ 
2 for  $i = |y| - 1$  downto 0 step -1 do
3    $r \leftarrow r^2$ 
4   if  $y_i = 1$  then
5      $r \leftarrow r \times x$ 
6   end
7 end
8 return  $r$ 
```

► Option #2: left-to-right *windowed* exponentiation.

- evaluate the Horner expansion step-by-step in an “inside-out” order ,
- express y in base- m for $m = 2^k$, i.e., extract a k -bit digit d from y in each step,
- trade-off space in favour of time, i.e., perform pre-computation reflected by X ,
- apply

$$r \leftarrow \begin{cases} r^{2^k} & \text{if } d = 0 \\ r^{2^k} \times (X[0] = x^1) & \text{if } d = 1 \\ r^{2^k} \times (X[1] = x^2) & \text{if } d = 2 \\ \vdots & \\ r^{2^k} \times (X[2^k - 2] = x^{2^k-1}) & \text{if } d = 2^k - 1 \end{cases}$$

so as to *accumulate* the result.

► **Option #2:** left-to-right *windowed* exponentiation.

Algorithm (RECODE-BASE- m)

Input: An integer y represented in base-2, an integer $m = 2^k$

Output: A sequence y' which represents y in base- 2^k

```
1  $y' \leftarrow \emptyset, i \leftarrow 0$ 
2 while  $y \neq 0$  do
3    $| y'_i \leftarrow y \wedge (2^k - 1), y \leftarrow y \gg k, i \leftarrow i + 1$ 
4 end
5 return  $y'$ 
```

Algorithm (1EXP-L2R-FIXEDWINDOW)

Input: A group element $x \in \mathbb{G}^\times$, a base-2 integer $0 \leq y < n$, an integer $m = 2^k$

Output: The group element $r = x^y \in \mathbb{G}^\times$

```
1  $y' \leftarrow \text{RECODE-BASE-}m(y, m = 2^k)$ 
2 Pre-compute  $X = [x^i \mid i \in \{1, 2, 3, \dots, m - 1 = 2^k - 1\}]$ 
3  $r \leftarrow 1$ 
4 for  $i = |y'| - 1$  downto 0 step -1 do
5   for  $j = 0$  upto  $k - 1$  step +1 do
6      $| r \leftarrow r^2$ 
7   end
8   if  $y'_i \neq 0$  then
9      $| r \leftarrow r \times X[y'_i - 1]$ 
10  end
11 end
12 return  $r$ 
```

Part 2.3: in practice (1)

► Problem:

- we want to represent and perform operations on integers modulo N , i.e., elements of

$$\mathbb{Z}_N = \{0, 1, 2, \dots, N - 1\},$$

- we *could* implement $\mathbb{Z}_N$ based on $\mathbb{Z}$, but any $x \in \mathbb{Z}_N$ will
 - always be positive, and
 - have a size that is upper-bounded by N .

► Solution:

1. a data structure to represent x , and
2. algorithms to operate on instances of said data structure.

Part 2.3: in practice (1)

► Problem:

- we want to represent and perform operations on integers modulo N , i.e., elements of

$$\mathbb{Z}_N = \{0, 1, 2, \dots, N-1\},$$

- we *could* implement $\mathbb{Z}_N$ based on $\mathbb{Z}$, but any $x \in \mathbb{Z}_N$ will
 - always be positive, and
 - have a size that is upper-bounded by N .

► Solution:

1. use the existing data structure for $\mathbb{N}$, and
2. focus on an algorithm for $x \times y \pmod{N}$ to support $\mathbb{Z}_N^\times$.

► Idea:

1. Use the relationship

$$t \pmod{N} \equiv t - (N \times \left\lfloor \frac{t}{N} \right\rfloor)$$

which

- uses a standard integer representation,
 - requires no pre-computation, but
 - requires an integer division, which is relatively complicated and computationally expensive.
2. Use **Barrett reduction** [4], which
 - uses a standard integer representation,
 - requires some pre-computation, and
 - requires $2 \cdot (l_N + 1) \cdot (l_N + 1)$ limb multiplications for an l_N -limb N .
 3. Use **Montgomery reduction** [7], which
 - uses a non-standard integer representation,
 - requires some pre-computation, and
 - requires $2 \cdot l_N \cdot l_N$ limb multiplications for an l_N -limb N .

Part 2.3: in practice (3)

- ▶ Montgomery-based multiplication $\leadsto$ exponentiation $\leadsto$ RSA:
 - ▶ pre-compute a set of **Montgomery parameters** $\Pi = (N, \rho, \omega)$ where
 1. $\rho = b^k$ for the smallest k such that $b^k > N$, and
 2. $\omega = -N^{-1} \pmod{\rho}$,assuming a base- b representation of N such that $\gcd(N, b) = 1$,

- Montgomery-based multiplication $\leadsto$ exponentiation $\leadsto$ RSA:

- the **Montgomery representation** of an integer

$$0 \leq x < N$$

is then defined as

$$\hat{x} = x \times \rho \pmod{N},$$

- Montgomery-based multiplication $\leadsto$ exponentiation $\leadsto$ RSA:

- using these concepts, we can define **Montgomery multiplication**:

Algorithm (MONTMUL)

Input: A set of Montgomery parameters $\Pi = (N, \rho, \omega)$, integers $0 \leq x, y < N$

Output: $r = x \times y \times \rho^{-1} \pmod{N}$

```
1  $r \leftarrow x \times y$   
2  $r \leftarrow (r + ((r \times \omega) \bmod \rho) \times N) / \rho$   
3 if  $r \geq N$  then  $r \leftarrow r - N$   
4 return  $r$ 
```

where, crucially and by-design, the modular reduction and division by ρ are special-case.

Part 2.3: in practice (3)

► Montgomery-based multiplication $\leadsto$ exponentiation $\leadsto$ RSA:

- given we can convert into

$$\begin{aligned}\text{MONTMUL}((N, \rho, \omega), x, \rho^2 \bmod N) &= x \times \rho^2 \times \rho^{-1} && (\bmod N) \\ &= x \times \rho && (\bmod N) \\ &= \hat{x}\end{aligned}$$

and from

$$\begin{aligned}\text{MONTMUL}((N, \rho, \omega), \hat{x}, 1) &= \hat{x} \times 1 \times \rho^{-1} && (\bmod N) \\ &= (x \times \rho) \times 1 \times \rho^{-1} && (\bmod N) \\ &= x\end{aligned}$$

Montgomery representation,

- we *could* then compute

$$r = x \times y \pmod{N} \quad \leftrightarrow \quad \begin{cases} \hat{x} &= \text{MONTMUL}((N, \rho, \omega), x, \rho^2 \bmod N) \\ \hat{y} &= \text{MONTMUL}((N, \rho, \omega), y, \rho^2 \bmod N) \\ \hat{r} &= \text{MONTMUL}((N, \rho, \omega), \hat{x}, \hat{y}) \\ r &= \text{MONTMUL}((N, \rho, \omega), \hat{r}, 1) \end{cases}$$

but the overhead of conversion is too high ...

- ▶ Montgomery-based multiplication $\leadsto$ exponentiation $\leadsto$ RSA:
 - ▶ ... instead, we adapt an exponentiation algorithm: for example

Algorithm (MONTExp)

Input: A set of Montgomery parameters $\Pi = (N, \rho, \omega)$, integers $0 \leq x, y < N$

Output: $r = x^y \pmod{N}$

```
1  $\hat{r} \leftarrow \text{MONTMUL}(\Pi, 1, \rho^2 \bmod N), \hat{x} \leftarrow \text{MONTMUL}(\Pi, x, \rho^2 \bmod N)$ 
2 for  $i = |y| - 1$  downto 0 step -1 do
3    $\hat{r} \leftarrow \text{MONTMUL}(\Pi, \hat{r}, \hat{r})$ 
4   if  $y_i = 1$  then
5      $\hat{r} \leftarrow \text{MONTMUL}(\Pi, \hat{r}, \hat{x})$ 
6   end
7 end
8 return  $\text{MONTMUL}(\Pi, \hat{r}, 1)$ 
```

i.e.,

- ▶ convert input into Montgomery representation,
- ▶ compute a series of (many) Montgomery multiplications,
- ▶ convert output from Montgomery representation,

and thus amortise the conversion.

- ▶ **Take away points:** you can often simply *use*

$$c = \text{RSA.ENC}((N, e), m) = m^e \pmod{N},$$

but understanding internals of this primitive can be useful and/or important.

- ▶ some historically interesting aspects; some “portable” concepts,
- ▶ close relationship between primitive and underlying Mathematics,
- ▶ wide range of viable implementation strategies,
- ▶ extensive deployment, in various contexts and use-cases.

Additional Reading

- ▶ [Wikipedia: RSA](https://en.wikipedia.org/wiki/RSA). URL: <https://en.wikipedia.org/wiki/RSA>.
- ▶ V. Shoup. “Chapter 3: Computing with large integers”. In: *Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008. URL: <http://shoup.net/ntb/>.
- ▶ A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. “Chapter 14: Efficient implementation”. In: *Handbook of Applied Cryptography*. CRC, 1996. URL: <http://cacr.uwaterloo.ca/hac/about/chap14.pdf>.
- ▶ D.M. Gordon. “A Survey of Fast Exponentiation Methods”. In: *Journal of Algorithms* 27 (1998), pp. 129–146.
- ▶ P.D. Barrett. “Implementing the Rivest, Shamir and Adleman public key encryption algorithm on a standard digital signal processor”. In: *Advances in Cryptology (CRYPTO)*. LNCS 263. Springer-Verlag, 1986, pp. 311–323.
- ▶ P.L. Montgomery. “Modular multiplication without trial division”. In: *Mathematics of Computation* 44.170 (1985), pp. 519–521.
- ▶ Ç.K. Koç, T. Acar, and B.S. Kaliski. “Analyzing and comparing Montgomery multiplication algorithms”. In: *IEEE Micro* 16.3 (1996), pp. 26–33.

References

- [1] [Wikipedia: RSA](https://en.wikipedia.org/wiki/RSA). URL: <https://en.wikipedia.org/wiki/RSA> (see p. 30).
- [2] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. “Chapter 14: Efficient implementation”. In: *Handbook of Applied Cryptography*. CRC, 1996. URL: <http://cacr.uwaterloo.ca/hac/about/chap14.pdf> (see p. 30).
- [3] V. Shoup. “Chapter 3: Computing with large integers”. In: *Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008. URL: <http://shoup.net/ntb/> (see p. 30).
- [4] P.D. Barrett. “Implementing the Rivest, Shamir and Adleman public key encryption algorithm on a standard digital signal processor”. In: *Advances in Cryptology (CRYPTO)*. LNCS 263. Springer-Verlag, 1986, pp. 311–323 (see pp. 23, 30).
- [5] D.M. Gordon. “A Survey of Fast Exponentiation Methods”. In: *Journal of Algorithms* 27 (1998), pp. 129–146 (see p. 30).
- [6] Ç.K. Koç, T. Acar, and B.S. Kaliski. “Analyzing and comparing Montgomery multiplication algorithms”. In: *IEEE Micro* 16.3 (1996), pp. 26–33 (see p. 30).
- [7] P.L. Montgomery. “Modular multiplication without trial division”. In: *Mathematics of Computation* 44.170 (1985), pp. 519–521 (see pp. 23, 30).
- [8] R.L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Communications of the ACM (CACM)* 21.2 (1978), pp. 120–126 (see p. 1).