Applied Cryptology

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

April 24, 2024

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

COMS30048 lecture: week #15

- 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 mulitiplication.
- Caveat!

$$\sim 2 \text{ hours} \quad \Rightarrow \quad \text{introductory, and (very) selective (versus definitive) coverage.}$$

Notes:

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

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

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

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

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

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

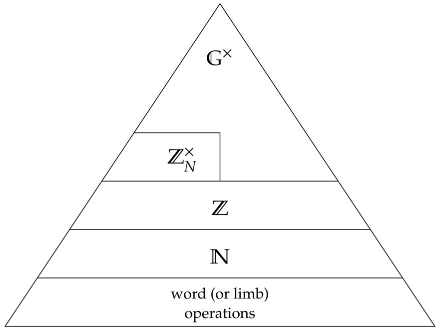
  - a digital signature scheme: (very) roughly, we just set

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

Notes:

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,	...
	$\vdots$		

Notes:

Part 2.1: in practice (1)

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

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

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

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

  - 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.

Notes:

Part 2.1: in practice (2)

- **Idea**:
  1. represent x using a base-b expansion

$$\begin{array}{rcl} \hat{x} & = & \langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{n-1} \rangle \\ & \mapsto & x \\ & = & \pm \sum_{i=0}^{n-1} \hat{x}_i \cdot b^i \end{array}$$

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

Notes:

Part 2.1: in practice (3)

- **Idea**:
  1. represent $x \in \{0, 1, \ldots 2^w - 1\}$ 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

```
typedef struct __mpz_t {
  limb_t d[ MPZ_LIMB_MAX ];

  int    l;
  int    s;
} 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.

Notes:

- Step #1: limb-focused operations, e.g., “add with carry”.
  - the idea is to support computation of

$$r_1 \cdot b + r_0 = t \leftarrow e + f + g$$

    i.e., a software-based version of a hardware-based full-adder cell (where $b = 2$),
  - for $b = 2^w$, the result r has at most $w + 1$ bits because

$$(2^w - 1) + (2^w - 1) + 1 \;=\; 2^{w+1} - 1 \;<\; 2^{w+1}.$$

Notes:

Part 2.1: in practice (4)
Algorithms for $x \in \{0, 1, \ldots, 2^w - 1\}$

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

Listing

```
#define LIMB_ADD1(r_1,r_0,e,f,g) { \
  limb_t __t_0 =      e +      f;    \
  limb_t __t_1 = __t_0 <      e;    \
           r_0 = __t_0 +      g;    \
  limb_t __t_2 =    r_0 < __t_0;    \
           r_1 = __t_1 | __t_2;    \
}
```

© Daniel Page (csdsp@bristol.ac.uk)
Applied Cryptology
University of BRISTOL
git # c8178615 @ 2024-04-24

Notes:

Part 2.1: in practice (4)

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

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

Listing

```
#define LIMB_ADD1(r_1,r_0,e,f,g)   {                  \
  dlimb_t __t  = ( dlimb_t )( e ) +                   \
                 ( dlimb_t )( f ) +                   \
                 ( dlimb_t )( g ) ;                   \
                                                      \
  r_0 =       ( limb_t )( __t >> 0                ); \
  r_1 = 0x1 & ( limb_t )( __t >> BITSOF( limb_t ) ); \
}
```


Notes:

Part 2.1: in practice (4)

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

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

Listing

```
#define LIMB_ADD1(r_1,r_0,e,f,g) { \
  asm( "movl %2,%0 ; movl $0,%1 ;  \
        addl %3,%0 ; adcl $0,%1 ;  \
        addl %4,%0 ; adcl $0,%1 ;" \
                                   \
       : "+&r" (r_0), "+&r" (r_1)  \
       :   "r" (e),     "r" (f),   \
           "r" (g)                 \
       : "cc"                 );   \
}
```

© Daniel Page (csdsp@bristol.ac.uk)
Applied Cryptology
University of BRISTOL
git # c8178615 @ 2024-04-24

Notes:

Part 2.1: in practice (5)

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

- 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

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

Notes:

Part 2.1: in practice (5)

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

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

Listing

```
#define LIMB_MUL2(r_1,r_0,e,f,g,h) {                   \
  dlimb_t __t  = ( dlimb_t )( e ) *                    \
                 ( dlimb_t )( f ) +                    \
                 ( dlimb_t )( g ) +                    \
                 ( dlimb_t )( h ) ;                    \
                                                       \
  r_0 =        ( limb_t )( __t >> 0                ); \
  r_1 =        ( limb_t )( __t >> BITSOF( limb_t ) ); \
}
```

© Daniel Page ⟨csdsp@bristol.ac.uk⟩
Applied Cryptology
University of BRISTOL
git # c8178615 @ 2024-04-24

Notes:

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

Listing

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

Notes:

- Step #2: $\mathbb{N}$-focused operations, e.g., "long addition".
  - the idea is to support computation of

$$
\begin{array}{rclcccccccccccc}
x & = & 623_{(10)} & \mapsto & & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & \\
y & = & 567_{(10)} & \mapsto & & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & + \\
\hline
c & = & & & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \\
r & = & 1190_{(10)} & \mapsto & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & \\
\hline
\end{array}
$$

    i.e., a software-based version of a hardware-based ripple-carry adder (where $b = 2$),
  - the result r has at most $\max(l_x, l_y) + 1$ limbs.

Notes:

- Step #2: $\mathbb{N}$-focused operations, e.g., "long addition".

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

Input: Two unsigned, base-b integers x and y
Output: An unsigned, base-b integer $r = x + y$

1. $l_x \leftarrow |x|$, $l_y \leftarrow |y|$, $l_r \leftarrow \max(l_x, l_y)$
2. $r \leftarrow 0$, $c \leftarrow 0$
3. **for** $i = 0$ **upto** $l_r - 1$ **step** $+1$ **do**
4. $\quad r_i \leftarrow (x_i + y_i + c) \bmod b$
5. $\quad$ **if** $(x_i + y_i + c) < b$ **then** $c \leftarrow 0$ **else** $c \leftarrow 1$
6. **end**
7. **return** r, c

Listing

```
limb_t mpn_add( limb_t* r, const limb_t* x, int l_x,
                           const limb_t* y, int l_y ) {

  int l_r = MAX( l_x, l_y );

  limb_t c = 0;

  for( int i = 0; i < l_r; i++ ) {
    limb_t d_x = ( i < l_x ) ? x[ i ] : 0;
    limb_t d_y = ( i < l_y ) ? y[ i ] : 0;

    LIMB_ADD1( c, r[ i ], d_x, d_y, c );
  }

  return c;
}
```

Notes:

Part 2.1: in practice (6)
Algorithms for $x \in \mathbb{N}$

- Step #2: $\mathbb{N}$-focused operations, e.g., "long addition".

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

Input: Two unsigned, base-b integers x and y
Output: An unsigned, base-b integer $r = x + y$

1. $l_x \leftarrow |x|$, $l_y \leftarrow |y|$, $l_r \leftarrow \max(l_x, l_y)$
2. $r \leftarrow 0$, $c \leftarrow 0$
3. **for** $i = 0$ **upto** $l_r - 1$ **step** $+1$ **do**
4. $\quad r_i \leftarrow (x_i + y_i + c) \bmod b$
5. $\quad$ **if** $(x_i + y_i + c) < b$ **then** $c \leftarrow 0$ **else** $c \leftarrow 1$
6. **end**
7. **return** r, c

Listing

```
limb_t mpn_add( limb_t* r, const limb_t* x, int l_x,
                           const limb_t* y, int l_y ) {

  int l_r = MIN( l_x, l_y ), i = 0;

  limb_t c = 0;

  while( i < l_r ) {
    limb_t d_x = x[ i ];
    limb_t d_y = y[ i ];

    LIMB_ADD1( c, r[ i ], d_x, d_y, c ); i++;
  }
  while( i < l_x ) {
    limb_t d_x = x[ i ];

    LIMB_ADD0( c, r[ i ], d_x,      c ); i++;
  }
  while( i < l_y ) {
    limb_t d_y = y[ i ];

    LIMB_ADD0( c, r[ i ],      d_y, c ); i++;
  }

  return c;
}
```

Notes:

- Step #2: $\mathbb{N}$-focused operations, e.g., "long multiplication".
  - the idea is to support computation of

x	$=$				6	2	3	
y	$=$				5	6	7	$\times$
$p_0 = 7 \cdot 3$	$=$					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	

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

Notes:

- 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 $r = x \cdot y$

$l_x \leftarrow |x|$, $l_y \leftarrow |y|$, $l_r \leftarrow l_x + l_y$
$r \leftarrow 0$
for $j = 0$ **upto** $l_y - 1$ **step** $+1$ **do**
  $c \leftarrow 0$
  for $i = 0$ **upto** $l_x - 1$ **step** $+1$ **do**
    $u \cdot b + v = t \leftarrow y_j \cdot x_i + r_{j+i} + c$
    $r_{j+i} \leftarrow v$
    $c \leftarrow u$
  end
  $r_{j+l_x} \leftarrow c$
end
return r

Listing

```
void mpn_mul( limb_t* r, const limb_t* x, int l_x,
                         const limb_t* y, int l_y ) {

  int l_r = l_x + l_y;

  limb_t R[ l_r ], c;

  memset( R, 0, l_r * SIZEOF( limb_t ) );

  for( int j = 0; j < l_y; j++ ) {
    c = 0;

    for( int i = 0; i < l_x; i++ ) {
      limb_t d_y = y[ j     ];
      limb_t d_x = x[     i ];
      limb_t d_R = R[ j + i ];

      LIMB_MUL2( c, R[ j + i ], d_y, d_x, d_R, c );
    }

    R[ j + l_x ] = c;
  }

  memcpy( r, R, l_r * SIZEOF( limb_t ) );
}
```


Notes:

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

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

Input: Two signed, base-b integers x and y
Output: A signed, base-b integer $r = x + y$

if $x < 0$ **and** $y \geq 0$ **then**
  if abs(x) $\geq$ abs(y) **then**
    $r \leftarrow -\mathbb{N}$-Sub(abs(x), abs(y))
  else
    $r \leftarrow +\mathbb{N}$-Sub(abs(y), abs(x))
  end
end
else if $x \geq 0$ **and** $y < 0$ **then**
  if abs(x) $\geq$ abs(y) **then**
    $r \leftarrow +\mathbb{N}$-Sub(abs(x), abs(y))
  else
    $r \leftarrow -\mathbb{N}$-Sub(abs(y), abs(x))
  end
end
else if $x \geq 0$ **and** $y \geq 0$ **then**
  $r \leftarrow +\mathbb{N}$-Add(abs(x), abs(y))
end
else if $x < 0$ **and** $y < 0$ **then**
  $r \leftarrow -\mathbb{N}$-Add(abs(x), abs(y))
end
return r

Listing

```
void mpz_add( mpz_t* r, const mpz_t* x,
                        const mpz_t* y ) {
  if     ( ( x->s <  0 ) && ( y->s >= 0 ) ) {
    if( mpn_cmp( x->d, x->l, y->d, y->l ) >= 0 ) {
      MPZ_SUB( r, x, y, MPZ_SIGN_NEG );
    }
    else {
      MPZ_SUB( r, y, x, MPZ_SIGN_POS );
    }
  }
  else if( ( x->s >= 0 ) && ( y->s <  0 ) ) {
    if( mpn_cmp( x->d, x->l, y->d, y->l ) >= 0 ) {
      MPZ_SUB( r, x, y, MPZ_SIGN_POS );
    }
    else {
      MPZ_SUB( r, y, x, MPZ_SIGN_NEG );
    }
  }
  else if( ( x->s >= 0 ) && ( y->s >= 0 ) ) {
      MPZ_ADD( r, x, y, MPZ_SIGN_POS );
  }
  else if( ( x->s <  0 ) && ( y->s <  0 ) ) {
      MPZ_ADD( r, x, y, MPZ_SIGN_NEG );
  }
}
```


Notes:

- **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 $r = x \cdot y$

if $x < 0$ **and** $y \geq 0$ **then**
  $r \leftarrow -\mathbb{N}$-Mul(abs(x), abs(y))
end
else if $x \geq 0$ **and** $y < 0$ **then**
  $r \leftarrow -\mathbb{N}$-Mul(abs(x), abs(y))
end
else if $x \geq 0$ **and** $y \geq 0$ **then**
  $r \leftarrow +\mathbb{N}$-Mul(abs(x), abs(y))
end
else if $x < 0$ **and** $y < 0$ **then**
  $r \leftarrow +\mathbb{N}$-Mul(abs(x), abs(y))
end
return r

Listing

```
void mpz_mul( mpz_t* r, const mpz_t* x,
                        const mpz_t* y ) {
  if( x->s == y->s ) {
    MPZ_MUL( r, x, y, MPZ_SIGN_POS );
  }
  else {
    MPZ_MUL( r, x, y, MPZ_SIGN_NEG );
  }
}
```


Notes:

Part 2.1: in practice (10)

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

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

Listing

```
#define MPZ_MUL(r,x,y,s_r) {                                 \
  int l_r = (x)->l + (y)->l;                                 \
                                                             \
                        mpn_mul( (r)->d, (x)->d, (x)->l,     \
                                         (y)->d, (y)->l );   \
                                                             \
  (r)->l              = mpn_lop( (r)->d, l_r );              \
  (r)->s              = s_r;                                 \
}
```

Listing

```
int mpn_lop( const limb_t* x, int l_x ) {
  while( ( l_x > 1 ) && ( x[ l_x - 1 ] == 0 ) ) {
    l_x--;
  }

  return l_x;
}
```

Notes:

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
  $$\begin{array}{rcrl} \text{multiplicative group } \mathbb{G}^\times & \rightsquigarrow & \textbf{exponentiation} & x^y \\ \text{additive group } \mathbb{G}^+ & \rightsquigarrow & \textbf{scalar multiplication} & [y]\,x \end{array}$$

Notes:

- 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

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

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

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

Notes:

Part 2.2: in practice (2)

- Idea: consider some $\mathbb{G}^{\times}$.
  - given $y = 14_{(10)} = 1110_{(2)}$ we can see that

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

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

$$\begin{aligned} x^y &= x^{y_0} \times (x^{y_1} \times (x^{y_2} \times (x^{y_3} \times (1)^2)^2)^2)^2 \\ & x^{0} \times (x^{1} \times (x^{1} \times (x^{1} \times (1)^2)^2)^2)^2 \\ & x^{0} \times (x^{1} \times (x^{1} \times (x^{1} \times 1)^2)^2)^2 \\ & x^{0} \times (x^{1} \times (x^{1} \times (x^{1})^2)^2)^2 \\ & x^{0} \times (x^{1} \times (x^{1} \times x^{2})^2)^2 \\ & x^{0} \times (x^{1} \times (x^{3})^2)^2 \\ & x^{0} \times (x^{1} \times x^{6})^2 \\ & x^{0} \times (x^{7})^2 \\ & x^{0} \times x^{14} \\ & x^{14} \end{aligned}$$

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

Notes:

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.

Notes:

Part 2.2: in practice (3)

- 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 $r = x^y \in \mathbb{G}^{\times}$

$r \leftarrow 1$
for $i = |y| - 1$ **downto** 0 **step** -1 **do**
  $r \leftarrow r^2$
  if $y_i = 1$ **then**
    $r \leftarrow r \times x$
  end
end
return r

Notes:

Part 2.2: in practice (4)

- 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.

Notes:

Part 2.2: in practice (4)

- 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

```
y' ← ∅, i ← 0
while y ≠ 0 do
  | y'_i ← y ∧ (2^k − 1), y ← y ≫ k, i ← i + 1
end
return y'
```

Algorithm (1Exp-L2R-FixedWindow)

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

```
y' ← Recode-Base-m(y, m = 2^k)
Pre-compute X = [x^i | i ∈ {1, 2, 3, ..., m − 1 = 2^k − 1}]
r ← 1
for i = |y'| − 1 downto 0 step −1 do
  | for j = 0 upto k − 1 step +1 do
  |   | r ← r^2
  | end
  | if y'_i ≠ 0 then
  |   | r ← r × X[y'_i − 1]
  | end
end
return r
```

Notes:

Part 2.3: in practice (1)

- **Problem**:
  - we want to represent and perform operations on integers modulo N, i.e., elements of
  $$\mathbb{Z}_N = \{0, 1, 2, \ldots, N-1\},$$
  - 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.

Notes:

Part 2.3: in practice (1)

- **Problem**:
  - we want to represent and perform operations on integers modulo N, i.e., elements of
  $$\mathbb{Z}_N = \{0, 1, 2, \ldots, N-1\},$$
  - 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 $\mathbb{Z}_N^\times$.

Notes:

© Daniel Page ⟨csdsp@bristol.ac.uk⟩
Applied Cryptology
University of BRISTOL
git # c8178615 @ 2024-04-24

Part 2.3: in practice (2)

- 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.

Notes:

Part 2.3: in practice (3)

- Montgomery-based multiplication $\rightsquigarrow$ exponentiation $\rightsquigarrow$ 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 $\gcd(N, b) = 1$,

Notes:

Part 2.3: in practice (3)

- Montgomery-based multiplication $\rightsquigarrow$ exponentiation $\rightsquigarrow$ RSA:
  - the **Montgomery representation** of an integer

$$0 \leq x < N$$

is then defined as

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

Notes:

Part 2.3: in practice (3)

- Montgomery-based multiplication $\rightsquigarrow$ exponentiation $\rightsquigarrow$ RSA:
  - using these concepts, we can define **Montgomery multiplication**:

Algorithm (MontMul)

Input: A set of Montgomery parameters $\Pi = (N, \rho, \omega)$, intgers $0 \leq x, y < N$
Output: $r = x \times y \times \rho^{-1} \pmod{N}$

$r \leftarrow x \times y$
$r \leftarrow (r + ((r \times \omega) \bmod \rho) \times N)/\rho$
if $r >= N$ **then** $r \leftarrow r - N$
return r

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

Notes:

Part 2.3: in practice (3)

- Montgomery-based multiplication $\rightsquigarrow$ exponentiation $\rightsquigarrow$ RSA:
  - given we can convert into

$$
\begin{array}{rcll}
\textsc{MontMul}((N,\rho,\omega),x,\rho^2 \bmod N) & = & x \times \rho^2 \times \rho^{-1} & \pmod{N} \\
 & = & x \times \rho & \pmod{N} \\
 & = & \hat{x} &
\end{array}
$$

and from

$$
\begin{array}{rcll}
\textsc{MontMul}((N,\rho,\omega),\hat{x},1) & = & \hat{x} \times 1 \times \rho^{-1} & \pmod{N} \\
 & = & (x \times \rho) \times 1 \times \rho^{-1} & \pmod{N} \\
 & = & x &
\end{array}
$$

Montgomery representation,
  - we *could* then compute

$$
r = x \times y \pmod{N} \quad \leftrightarrow \quad \left\{ \begin{array}{rcl}
\hat{x} & = & \textsc{MontMul}((N,\rho,\omega),x,\rho^2 \bmod N) \\
\hat{y} & = & \textsc{MontMul}((N,\rho,\omega),y,\rho^2 \bmod N) \\
\hat{r} & = & \textsc{MontMul}((N,\rho,\omega),\hat{x},\hat{y}) \\
r & = & \textsc{MontMul}((N,\rho,\omega),\hat{r},1)
\end{array} \right.
$$

but the overhead of conversion is too high ...

Notes:

Part 2.3: in practice (3)

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

Algorithm (MontExp)

Input: A set of Montgomery parameters $\Pi = (N,\rho,\omega)$, intgers $0 \leq x,y < N$
Output: $r = x^y \pmod{N}$

$\hat{r} \leftarrow \textsc{MontMul}(\Pi,1,\rho^2 \bmod N)$, $\hat{x} \leftarrow \textsc{MontMul}(\Pi,x,\rho^2 \bmod N)$
for $i = |y| - 1$ **downto** 0 **step** -1 **do**
  $\hat{r} \leftarrow \textsc{MontMul}(\Pi,\hat{r},\hat{r})$
  if $y_i = 1$ **then**
    $\hat{r} \leftarrow \textsc{MontMul}(\Pi,\hat{r},\hat{x})$
  end
end
return $\textsc{MontMul}(\Pi,\hat{r},1)$

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

and thus amortise the conversion.

Notes:

Conclusions

- Take away points: you can often simple *use*

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

  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.

Notes:

Additional Reading

- *Wikipedia: 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.

Notes:

References

[1] *Wikipedia: RSA*. URL: https://en.wikipedia.org/wiki/RSA (see p. 79).

[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. 79).

[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. 79).

[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. 65, 79).

[5] D.M. Gordon. "A Survey of Fast Exponentiation Methods". In: *Journal of Algorithms* 27 (1998), pp. 129–146 (see p. 79).

[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. 79).

[7] P.L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of Computation* 44.170 (1985), pp. 519–521 (see pp. 65, 79).

[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. 5).

Notes: