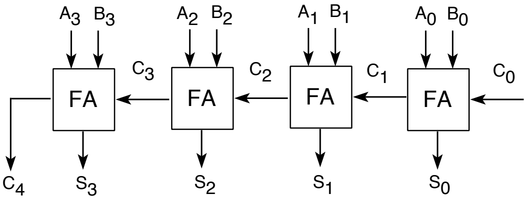
Arithmetic in Integer Rings and Prime Fields



Contents

Integer Rings and Finite Fields in Cryptography

- Several cryptographic algorithms are based on similar mathematical structures built upon finite sets of integers:
  - Rings $\mathcal{Z}_n$ or groups $\mathcal{Z}_n^*$ for a composite n
  - Fields GF(p) or their multiplicative groups for a prime p
- The arithmetic of such structures are often called modular arithmetic
- The arithmetic operations of interest in cryptography are addition, multiplication and inversion mod n or mod p
- The modulus n or p is either composite or prime
- The fact that modulus is prime or composite makes little difference in addition and multiplication algorithms

Integer Addition

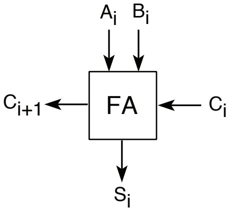
- The computation of two k-bit numbers a and b
- The bits are represented using A_i and B_i

$$
\begin{array}{rcccccc}
 & & A_{k-1} & A_{k-2} & \cdots & A_1 & A_0 \\
+ & & B_{k-1} & B_{k-2} & \cdots & B_1 & B_0 \\
\hline
 & C_k & S_{k-1} & S_{k-2} & \cdots & S_1 & S_0
\end{array}
$$

- Carry propagate adder
- Carry completion sensing adder
- Carry look-ahead adder
- Carry save adder

Carry Propagate Adder: CPA

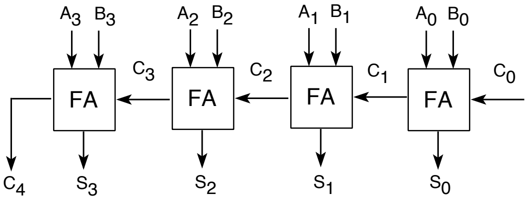
- The full adder box: FA



$$
\begin{aligned}
S_i &= A_i \oplus B_i \oplus C_i \\
C_{i+1} &= A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i
\end{aligned}
$$

$$
\begin{aligned}
\oplus &\rightarrow \text{XOR} \\
\cdot &\rightarrow \text{AND} \\
+ &\rightarrow \text{OR}
\end{aligned}
$$

- Topology:



Properties of CPA

- Total (worst case) delay $= k \times$ FA delay
- The circuit needs consider the worst case scenarios

$$\begin{array}{rr} & 011111111111111111111 \\ + & 000000000000000000001 \\ \hline & 100000000000000000000 \end{array}$$

- Total area $= k \times$ FA area
- Scales up easily for k
- Subtraction is easy: Use 2's complement arithmetic
- Sign detection is easy: MSB gives the sign

Carry Completion Sensing Adder

- While the worst case carry propagation length is k, there will be many cases in which carry propagation length will be a lot less
- The carry completion sensing adder waits only as long as the longest carry, which is less than k
- The carry completion sensing adder is an asynchronous adder which detects the completion of the carry propagation process
- An example of carry propagation processes

$$
\begin{array}{rcccccccccccccccc}
 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
+ & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
\end{array}
$$

$$\xleftarrow{\quad 4 \quad}\xleftarrow{\ 2\ }\qquad\xleftarrow{\qquad 5 \qquad}\quad\xleftarrow{\ 1\ }$$

- Analysis shows that average carry length is bounded by $\log_2(k)$

Carry Completion Sensing Adder

- Carry completion signal is a bit pair (C, N) which is produced from the current input bit pair (A, B)
- The carry completion signals are then applied to a wide AND gate which computes the product of all carry completion signals $C + N$

$$
\begin{array}{lcl}
(A, B) = (0, 0) & \Rightarrow & (C, N) \leftarrow (0, 1) \\
(A, B) = (1, 1) & \Rightarrow & (C, N) \leftarrow (1, 0) \\
(A, B) = (0, 1) & \Rightarrow & (C, N) \leftarrow \text{previous } (C, N) \\
(A, B) = (1, 0) & \Rightarrow & (C, N) \leftarrow \text{previous } (C, N)
\end{array}
$$

- When $C + N$ is determined, it will be 1 and it remains at 1
- Undetermined $C + N$ values are kept at logic 0

Carry Completion Sensing Adder

A	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1
B	1	0	0	1	1	1	0	0	0	0	1	0	1	0	1
C				1		1		0					1	0	1
N				0		0		1					0	1	0
$C+N$				1		1		1					1	1	1
C			1	1	1	1	0	0				1	1	0	1
N			0	0	0	0	1	1				0	0	1	0
$C+N$			1	1	1	1	1	1				1	1	1	1
C		1	1	1	1	1	0	0			1	1	1	0	1
N		0	0	0	0	0	1	1			0	0	0	1	0
$C+N$		1	1	1	1	1	1	1			1	1	1	1	1
C	1	1	1	1	1	1	0	0		1	1	1	1	0	1
N	0	0	0	0	0	0	1	1		0	0	0	0	1	0
$C+N$	1	1	1	1	1	1	1	1		1	1	1	1	1	1
C	1	1	1	1	1	1	0	0	1	1	1	1	1	0	1
N	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0
$C+N$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Carry Look-Ahead Adder

- Compute C_is in advance using more logic
- Then, use C_is to compute S_is in parallel
- Let $G_i = A_i B_i$ and $P_i = A_i + B_i$
- C_{i+1} is a function of C_0 and $G_0, G_1, \ldots, G_i$ and $P_0, P_1, \ldots, P_i$

$$
\begin{aligned}
C_1 &= A_0 B_0 + C_0(A_0 + B_0) \\
&= G_0 + C_0 P_0 \\
C_2 &= G_1 + C_1 P_1 = G_1 + G_0 P_1 + C_0 P_0 P_1 \\
C_3 &= G_2 + C_2 P_2 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2 \\
C_4 &= G_3 + C_3 P_3 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3
\end{aligned}
$$

Properties of CLA

- The total delay is $O(\log k)$
- The total area is essentially $O(k)$ using parallel prefix circuits (See: Ladner & Fischer, Brent & Kung)
- A complete CLA is not cost-effective for large k (> 256)
- By grouping G and P functions, larger CLAs can be designed
- Even with grouping, design of a 1024-bit adder may not be feasible or cost-effective

Carry Save Adder

- Input: 3 k-bit numbers a, b, and c

$$\begin{aligned} a &= (A_{k-1}A_{k-2}\cdots A_1A_0) \\ b &= (B_{k-1}B_{k-2}\cdots B_1B_0) \\ c &= (C_{k-1}C_{k-2}\cdots C_1C_0) \end{aligned}$$

- Output: 2 k-bit numbers c' and s such that $c' + s = a + b + c$

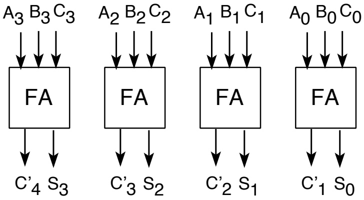
$$\begin{aligned} s &= (S_kS_{k-1}\cdots S_1S_0) \\ c' &= (C'_kC'_{k-1}\cdots C'_2C'_1) \end{aligned}$$

- The individual bits of s and c' are computed as

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i \\ C'_{i+1} &= A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i \end{aligned}$$

Carry Save Adder

- Topology:



- An example: $40 + 25 + 20 \to 48 + 37$

$$
\begin{array}{lcrr}
A & = & 40 & 101000 \\
B & = & 25 & 011001 \\
C & = & 20 & 010100 \\
\hline
S & = & 37 & 100101 \\
C' & = & 48 & 011000
\end{array}
$$

Properties of Carry Save Adder

- The total delay is $O(1)$ (a single FA delay)
- The total area is $k \times$ FA area
- Scales up easily for large k
- Subtraction is easy: Use 2's complement arithmetic
- Sign detection is "complicated"

Sign Detection Problem for Carry Save Adders

- Numbers are represented in Carry and Sum pairs $x = (c', s)$
- The actual value of the number is $x = c' + s$
- Unless the addition is performed in full length, the correct sign may never be determined
- Example: $a = -18$, $b = 19$, and $c = 6$ are given
- We compute their sum using the CSA

a	$=$	-18	101110	
b	$=$	19	010011	
c	$=$	6	000110	
s	$=$	-5	111011	
c'	$=$	12	000110	
			1	(1 MSB)
			11	(2 MSB)
			000	(3 MSB)
			0001	(4 MSB)
			00011	(5 MSB)

A Sign Estimation Algorithm for CSA

- We add the most significant t bits of c' and s to estimate the sign of $x = c' + s$, represented as $\text{esign}_t(c', s)$

$$\begin{aligned} c' &= 011110 \\ s &= 001010 \\ \text{esign}_1(c', s) &= \mathbf{0} \\ \text{esign}_2(c', s) &= \mathbf{01} \\ \text{esign}_3(c', s) &= \mathbf{1}00 \\ \text{esign}_4(c', s) &= \mathbf{1}001 \\ \text{esign}_5(c', s) &= \mathbf{1}0100 \end{aligned}$$

- It is shown: if $\text{esign}_t(c', s)$ is used for mod n reduction, then:

$$C' + S < n + 2^{k-t}$$

where n is the modulus and k is its length in bits

Addition and Subtraction mod n

- The computation of $s = a + b \bmod n$
- Add and Reduce:
  Given $a, b < n$
  Compute $s' = a + b$
  Compute $s'' = s' - n$
  If $s'' \geq 0$
  then $s = s''$
  else $s = s'$
- Requires fast sign detection: Is $s'' \geq 0$?

Incomplete (Lazy) Reduction

- Given $a, b < 2^k$ $\quad$ (a, b can be $\geq n$)
- Compute $s' = a + b$
  If there is a carry out of the k-bit register
  then $s = s' + m$
  else $s = s'$
- Correction factor: $m = 2^k - n$ (precomputed)

Incomplete Reduction

- Carry out of the k-bit register implies

$$(s') = a + b \geq 2^k$$

- Thus, if the carry is discarded, we essentially compute

$$s' = a + b - 2^k$$

- The result is then corrected by adding m to s'

$$\begin{aligned} s &= s' + m \\ &= a + b - 2^k + m \\ &= a + b - n \end{aligned}$$

- A temporary value may be larger than n, but it is always less than 2^k
- Whenever it exceeds 2^k, we discard the carry, and perform a correction

Incomplete Reduction Example

- $n = 39$, thus $m = 64 - 39 = 25 = (011001)$

$$\begin{array}{lcrll} a & = & 23 = & (010111) & \\ b & = & 26 = & (011010) & \\ s' & = & a+b = & 0(110001) & \text{no carry out} \\ s & = & s' = & (110001) & \end{array}$$

$$\begin{array}{lcrll} a & = & 40 = & (101000) & \\ b & = & 30 = & (011110) & \\ s' & = & s+b = & 1(000110) & \text{carry out} \\ m & = & & (011001) & \\ s & = & s'+m = & (011111) & \text{correction} \end{array}$$

Final Correction Phase

- After all additions are completed, a final result that is out of range can be corrected by adding m:

$$
\begin{aligned}
s &= & & (110001) \\
m &= & & (011001) \\
s &= & s+m = & \; 1(001010) \\
s &= & & (001010)
\end{aligned}
$$

Modular Multiplication

- Given $a, b < n$, compute $p = a \cdot b \bmod n$
- Methods:
  - Multiply and reduce:
    Multiply: $p' = a \cdot b$ ($2k$-bit number)
    Reduce: $p = p' \bmod n$ (k-bit number)
  - Interleave multiply and reduce steps
  - The Montgomery multiplication

Interleaving Multiply and Reduce

- The product $p' = a \cdot b$ can be written as

$$p' = a \cdot \sum_{i=0}^{k-1} B_i 2^i \;=\; a \cdot (B_0 + B_1 2^1 + B_2 2^2 + \cdots + B_{k-1} 2^{k-1})$$

- We can apply Horner's rule to this formulation of p'
- The initial value $p' = 0$ and the loop starts with B_{k-1} and moves down with $B_{k-2}, B_{k-3}, \ldots$

$$\begin{aligned}
p' &\leftarrow 2 \cdot p' + a \cdot B_{k-1} \\
&= a \cdot B_{k-1} \\
p' &\leftarrow 2 \cdot p' + a \cdot B_{k-2} \\
&= 2 \cdot a \cdot B_{k-1} + a \cdot B_{k-2} \\
&\vdots \\
p' &\leftarrow 2 \cdot p' + a \cdot B_i
\end{aligned}$$

Interleaving Multiply and Reduce

- This formulation yields the shift-add multiplication algorithm

1: $p' \leftarrow 0$
2: **for** $i = k - 1$ **downto** 0
2a: $\quad p' \leftarrow 2 \cdot p' + a \cdot B_i$
3: **return** p'

- We can also reduce the partial product mod n at each step:

1: $p \leftarrow 0$
2: **for** $i = k - 1$ **downto** 0
2a: $\quad p \leftarrow 2 \cdot p + a \cdot B_i$
2b: $\quad p \leftarrow p \bmod n$
3: **return** p

Interleaving Multiply and Reduce

- Assuming that $a, b, p < n$, we have

$$\begin{aligned} p &\leftarrow 2 \cdot p + a \cdot B_j \\ &\leq 2(n-1) + (n-1) = 3n - 3 \end{aligned}$$

- Thus, at most two subtractions are needed to reduce p to the range $0 \leq p < n$
- We can use

$$p' \leftarrow p - n \; ; \; \textbf{if} \;\; p' \geq 0 \;\; \textbf{then} \;\; p \leftarrow p'$$
$$p' \leftarrow p - n \; ; \; \textbf{if} \;\; p' \geq 0 \;\; \textbf{then} \;\; p \leftarrow p'$$

- Addition and subtraction steps need to be performed faster

Interleaving Multiply and Reduce

- Carry propagate adder gives $O(k)$ delay
- Incomplete reduction can be used to avoid unnecessary subtractions:
  2a. $p \leftarrow 2p$
  2b. **if** carry-out **then** $p \leftarrow p + m$
  2c. $p \leftarrow p + a \cdot B_j$
  2d. **if** carry-out **then** $p \leftarrow p + m$
- Carry save adder gives $O(1)$ delay; fast sign detection is needed to decide if the partial product needs to be reduced modulo n
  2a. $(c, s) \leftarrow 2c + 2s + a \cdot B_i$
  2b. $(c', s') \leftarrow c + s - n$
  2c. **if** $\text{esign}_t(c', s') \geq 0$ **then** $(c, s) \leftarrow (c', s')$
- Function $\text{esign}_t(c', s')$ estimates the sign of $c' + s'$

Montgomery Multiplication

- The Montgomery multiplication algorithm replaces division by n operation with division by $r = 2^k$
- If n is a k-bit odd integer, i.e., $2^{k-1} < n < 2^k$, we assign $r = 2^k$
- We map the integers $a \in [0, n-1]$ to the integers $\bar{a} \in [0, n-1]$ using

$$\bar{a} = a \cdot r \pmod{n}$$

- For example, for $n = 11$ and $r = 16$ the mapping is

a :	0	1	2	3	4	5	6	7	8	9	10
$\bar{a}$:	0	5	10	4	9	3	8	2	7	1	6

Definition of Montgomery Product

- The Montgomery product of $a, b \in [0, n-1]$ is defined as

$$\text{MonPro}(a, b) = a \cdot b \cdot r^{-1} \pmod{n}$$

- Here r^{-1} is the multiplicative inverse of r modulo n
- The inverse of $r = 2^k$ exists if the modulus n is odd
- Interestingly the Montgomery product of two integers actually involves two multiplications, instead of one
- Furthermore, we need $r^{-1} \pmod{n}$, but it can be precomputed

Properties of the Montgomery Product

- **Property 1:** If $c = a \cdot b \pmod{n}$, then $\bar{c} = \text{MonPro}(\bar{a}, \bar{b})$

$$
\begin{aligned}
\text{MonPro}(\bar{a}, \bar{b}) &= \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \\
&= (a \cdot r) \cdot (b \cdot r) \cdot r^{-1} \pmod{n} \\
&= a \cdot b \cdot r \pmod{n} \\
&= c \cdot r \pmod{n} \\
&= \bar{c}
\end{aligned}
$$

Properties of the Montgomery Product

- **Property 2:** $\bar{a} = \text{MonPro}(a, r^2)$

$$\begin{aligned} \text{MonPro}(a, r^2) &= a \cdot r^2 \cdot r^{-1} \pmod{n} \\ &= a \cdot r \pmod{n} \\ &= \bar{a} \end{aligned}$$

- **Property 3:** $c = \text{MonPro}(\bar{c}, 1)$

$$\begin{aligned} \text{MonPro}(\bar{c}, 1) &= \bar{c} \cdot 1 \cdot r^{-1} \pmod{n} \\ &= (c \cdot r) \cdot 1 \cdot r^{-1} \pmod{n} \\ &= c \end{aligned}$$

Classical Montgomery Algorithm

- Peter Montgomery introduced his original algorithm in 1985
- The function MonPro(a, b) computes $a \cdot b \cdot r^{-1} \pmod n$
- Interestingly the algorithm does not need $r^{-1} \pmod n$
- However, it requires another quantity n' which is related to it

function MonPro(a, b)
Input: a, b, n, n'
Output: $u = a \cdot b \cdot r^{-1} \bmod n$
1: $t \leftarrow a \cdot b$
2: $m \leftarrow t \cdot n' \pmod r$
3: $u \leftarrow (t + m \cdot n)/r$
4: **if** $u \geq n$ **then** $u \leftarrow u - n$
5: **return** u

Computation of n'

- The quantity n' appears in the computation of $r^{-1} \pmod{n}$ using the extended Euclidean algorithm
- The EEA computes r^{-1} and n' using

$$(s,t) \leftarrow \text{EEA}(r,n) \quad \Rightarrow \quad s \cdot r + t \cdot n = 1$$

- Here we have $r^{-1} = s \pmod{n}$ and n' is defined to be $n' = -t$
- While $r^{-1} \pmod{n}$ is not needed, the Montgomery function requires n' which is also computed using the EEA
- Furthermore, they are related as

$$r^{-1} \cdot r + (-n') \cdot n = 1 \quad \Rightarrow \quad n' = \frac{-1 + r \cdot r^{-1}}{n}$$

Properties of the Montgomery Algorithm

- Steps 2 and 3 of the Montgomery algorithm seem complicated, as they are modular multiplication and division operations
- However, the modular reduction and division operations involve the modulus and divisor as r which is a power of 2
- **Step 2:** The Montgomery function performs **modular** multiplication $m \leftarrow t \cdot n' \pmod{r}$, however, the modulus is $r = 2^k$, which means the reduction by r is accomplished by taking the least significant k bits of the product
- Example: Given $273 = (10101011\underline{0111})$, we reduce it mod $16 = 2^4$ by taking its least significant 4 bits: $(0111) = 7$
- Indeed $273 = 7 \pmod{16}$

Properties of the Montgomery Algorithm

- **Step 3:** The Montgomery function first performs $u \leftarrow (t + m \cdot n)$, and then divides u by $r = 2^k$, which implies a k-bit right shift $u \leftarrow (t + m \cdot n)/2^k$, i.e., discarding the least significant k bits
- Example: Given $208 = (\underline{1101}0000)$, we divide it by $16 = 2^4$ by discarding its least significant 4 bits and obtain $(1101) = 13$
- Indeed $208/16 = 13$

- Thus, we conclude that the modular reduction by r in Step 2 and the division by r in Step 3 are simple operations on a digital computer
- They are easily accomplished:
  Reduction by $r = 2^k$: "taking least significant k bits"
  Division by $r = 2^k$: "discarding least significant k bits"

Properties of the Montgomery Algorithm

- To compute $a \cdot b \cdot r^{-1} \pmod{n}$ for a k-bit odd $n < r$ and $r = 2^k$, the MonPro function performs only multiplications in Steps 1, 2, and 3
- Multiplication operations require $O(k^2)$ bit operations if the standard algorithms are being utilized
- The modular reduction by r operation in Step 2 and the division by r operation in Step 3 require only $O(k)$ bit operations
- Similarly, the subtraction in Step 4 is also $O(k)$

- The power of the Montgomery algorithm is that it requires no division or reduction by n which is an arbitrary k-bit integer
- However, it requires computation of n' using the EEA
- It also requires 3 integer multiplications (Steps 1, 2, and 3)

Correctness of the Montgomery Algorithm

- For proof, we use two facts
  - $n' = (-1 + r \cdot r^{-1})/n$ implies $1 + n' \cdot n = r \cdot r^{-1}$
  - $m = t \cdot n' \pmod{r}$ implies $m = t \cdot n' + N \cdot r$ for some N
- MonPro computes

$$
\begin{aligned}
u &= (t + m \cdot n)/r \\
&= (t + [t \cdot n' + N \cdot r] \cdot n)/r \\
&= (t \cdot [1 + n' \cdot n] + N \cdot r \cdot n)/r \\
&= (t \cdot r \cdot r^{-1} + N \cdot r \cdot n)/r \\
&= t \cdot r^{-1} + N \cdot n \\
&= a \cdot b \cdot r^{-1} + N \cdot n \\
&= a \cdot b \cdot r^{-1} \pmod{n}
\end{aligned}
$$

Montgomery Exponentiation

- MonPro function is not suitable for a single modular multiplication $c = a \cdot b \pmod{n}$ since it has significant overhead
- Compute n' using the EEA

$$(s, t) \leftarrow \text{EEA}(r, n) \quad \Rightarrow \quad n' = -t$$

- Convert a and b to bar notation

$$\begin{aligned} \bar{a} &\leftarrow \text{MonPro}(a, r^2) \\ \bar{b} &\leftarrow \text{MonPro}(b, r^2) \end{aligned}$$

- Perform the Montgomery product: $\bar{c} \leftarrow \text{MonPro}(\bar{a}, \bar{b})$
- Convert $\bar{c}$ to unbar notation: $c \leftarrow \text{MonPro}(\bar{c}, 1)$

Montgomery Exponentiation

- However, MonPro function is very suitable for several modular multiplications with the same modulus: Montgomery Exponentiation

function MonExp(m, d, n)
Input: m, d, n
Output: $s = m^d \bmod n$
1: $\bar{m} \leftarrow$ MonPro(m, r^2)
2: $\bar{s} \leftarrow$ MonPro($1, r^2$)
3: **for** $i = k - 1$ **downto** 0
4a: $\bar{s} \leftarrow$ MonPro($\bar{s}, \bar{s}$)
4b: **if** $d_i = 1$ **then** $\bar{s} \leftarrow$ MonPro($\bar{s}, \bar{m}$)
5: $s \leftarrow$ MonPro($\bar{s}, 1$)
3: **return** s

Montgomery Exponentiation Example

- Computation of $\text{MonExp}(3, 50, 55) = 3^{50} \pmod{55}$
- Since $n = 55$, we can take r is the next power of 2 as $r = 64$
- Using the EEA we compute

$$\text{EEA}(r, n) = \text{EEA}(64, 55) \Rightarrow (r^{-1}, -n') = (49, -57)$$

- Thus, we obtain $r^{-1} = 49$ and $n' = 57$
- We start with $m = 3$ and $s = 1$
- $\bar{m} \leftarrow \text{MonPro}(m, r^2) = \text{MonPro}(3, 64^2)$ which gives $\bar{m} = 27$
- $\bar{s} \leftarrow \text{MonPro}(s, r^2) = \text{MonPro}(1, 64^2)$ which gives $\bar{s} = 9$

Montgomery Exponentiation Example

- $e = 50 = (110010)_2$

e_i	Step 5	Step 6
1	$\text{MonPro}(9, 9) = 9$	$\text{MonPro}(9, 27) = 27$
1	$\text{MonPro}(27, 27) = 26$	$\text{MonPro}(26, 27) = 23$
0	$\text{MonPro}(23, 23) = 16$	
0	$\text{MonPro}(16, 16) = 4$	
1	$\text{MonPro}(4, 4) = 14$	$\text{MonPro}(14, 27) = 42$
0	$\text{MonPro}(42, 42) = 31$	

- $s = \text{MonPro}(31, 1) = 34$

The Montgomery Exponentiation Example

- Computation of MonPro$(27, 27)$:

$$\begin{aligned}
t &\leftarrow 27 \cdot 27 \\
&= 729 \\
m &\leftarrow 729 \cdot 57 \pmod{64} \\
&\leftarrow 41553 \pmod{64} \\
&\leftarrow (1010001001\ \underline{010001}) \\
&= 17 \\
u &\leftarrow (729 + 17 \cdot 55)/64 \\
&\leftarrow 1664/64 \\
&\leftarrow (\underline{11010}\ 000000) \\
&= 26
\end{aligned}$$

The Montgomery Exponentiation Example

- Computation of MonPro$(31, 1)$:

$$
\begin{aligned}
t &\leftarrow 31 \cdot 1 \\
&= 31 \\
m &\leftarrow 31 \cdot 57 \pmod{64} \\
&\leftarrow 1767 \pmod{64} \\
&\leftarrow (11011\ \underline{100111}) \\
&= 39 \\
u &\leftarrow (31 + 39 \cdot 55)/64 \\
&\leftarrow 2176/64 \\
&\leftarrow (100010\ \underline{000000}) \\
&= 34
\end{aligned}
$$

Derivation of the CIOS Algorithm

- CIOS stands for Coarsely Integrated Operand Scanning
- CIOS performs the MonPro function
- It is more efficient than the classical Montgomery algorithm
- CIOS requires 1.5 integer multiplications instead of 3
- Furthermore, the binary CIOS algorithm does not require the computation of n_0' because it is always equal to 1
- The word-level CIOS requires the computation of only the least significant word (the least significant w bits) of n_0'

Derivation of the binary CIOS

- Consider the Montgomery function which computes

$$\text{MonPro}(a, b) = a \cdot b \cdot r^{-1} \pmod{n}$$

- Since $r = 2^k$, we can write it as

$$\begin{aligned} u &= a \cdot b \cdot 2^{-k} \pmod{n} \\ &= \left(\sum_{i=0}^{k-1} A_i 2^i \right) \cdot b \cdot 2^{-k} \pmod{n} \\ &= \left(\sum_{i=0}^{k-1} A_i 2^{i-k} \right) \cdot b \pmod{n} \end{aligned}$$

- Thus, we obtain

$$u = (A_0 2^{-k} + A_1 2^{-k+1} + \cdots + A_{k-1} 2^{-1}) \cdot b \pmod{n}$$

Derivation of the binary CIOS

- We can apply Horner's rule to this formulation of u

$$u = (A_0 2^{-k} + A_1 2^{-k+1} + \cdots + A_{k-1} 2^{-1}) \cdot b \pmod{n}$$

- The initial value of the sum $u = 0$ and the innermost loop starts with A_0 and moves up with $A_1, A_2, \ldots$

$$\begin{aligned}
u &\leftarrow (u + A_0 \cdot b) \cdot 2^{-1} \pmod{n} \\
&= A_0 \cdot b \cdot 2^{-1} \pmod{n} \\
u &\leftarrow (u + A_1 \cdot b) \cdot 2^{-1} \pmod{n} \\
&= A_0 \cdot b \cdot 2^{-2} + A_1 \cdot b \cdot 2^{-1} \pmod{n} \\
u &\leftarrow (u + A_2 \cdot b) \cdot 2^{-1} \pmod{n} \\
&= A_0 \cdot b \cdot 2^{-3} + A_1 \cdot b \cdot 2^{-2} + A_2 \cdot b \cdot 2^{-1} \pmod{n} \\
&\vdots \\
u &\leftarrow (u + A_i \cdot b) \cdot 2^{-1} \pmod{n}
\end{aligned}$$

Derivation of the binary CIOS

- This formulation gives the following code for the binary CIOS:

1: $u \leftarrow 0$
2: **for** $i = 0$ **to** $k-1$
3: $\quad u \leftarrow (u + A_i \cdot b) \cdot 2^{-1} \pmod{n}$
4: **return** u

- We can separate Step 3 into two steps

1: $u \leftarrow 0$
2: **for** $i = 0$ **to** $k-1$
3a: $\quad u \leftarrow u + A_i \cdot b$
3b: $\quad u \leftarrow u \cdot 2^{-1} \pmod{n}$
4: **return** u

Derivation of the binary CIOS

- The computation of $u \cdot 2^{-1} \pmod{n}$ for a given u can be performed without explicitly computing $2^{-1} \pmod{n}$
- If u in Step 3a is **even**, such that $u = 2v$, then Step 3b gives

$$
\begin{array}{rcll}
u & = & 2v & \{ \text{ Step 3a } \} \\
u & \leftarrow & (2v) \cdot 2^{-1} & \pmod{n} \\
 & = & v &
\end{array}
$$

- If u in Step 3a is **odd**, then we know $u + n$ will be even since n is always odd, therefore, Step 3 gives

$$
\begin{array}{rcll}
u & = & \text{odd} & \{ \text{ Step 3a } \} \\
u + n & = & 2v & \\
u & \leftarrow & (2v) \cdot 2^{-1} & \pmod{n} \\
 & = & v &
\end{array}
$$

Derivation of the binary CIOS

- Furthermore, v is easily computed for an even u as $v \leftarrow u/2$
- Therefore, we revise Step 3b into two steps, the first step is the if-then statement checking if the number is odd, while the second step performs division by 2

1: $u \leftarrow 0$
2: **for** $i = 0$ **to** $k-1$
3a: $\quad u \leftarrow u + A_i \cdot b$
3b: $\quad$ **if** u is odd **then** $u \leftarrow u + n$
3c: $\quad u \leftarrow u/2$
4: **return** u

- The resulting algorithm has several nice properties

The Binary CIOS Algorithm

- However, we should also add the subtraction step
- Furthermore, checking if u is odd can be made more efficient
- Given u, the LSB $U_0 = 1$ implies that u is odd

1: $u \leftarrow 0$
2: **for** $i = 0$ **to** $k - 1$
3a: $u \leftarrow u + A_i \cdot b$
3b: $u \leftarrow u + U_0 \cdot n$
3c: $u \leftarrow u/2$
4: **if** $u > n$ **then** $u \leftarrow u - n$
5: **return** u

An Example of the Binary CIOS Algorithm

- Consider the computation of MonPro$(27, 27)$ using the binary CIOS algorithm for $n = 55$ and $k = 6$
- We have $a = 27 = (011011)$ and $b = 27$

i	a_i	Step 3a (u)	u_0	Step 3b (u)	Step 3c (u)
0	1	$0 + 1 \cdot 27 = 27$	1	$27 + 1 \cdot 55 = 82$	$82/2 = 41$
1	1	$41 + 1 \cdot 27 = 68$	0	68	$68/2 = 34$
2	0	$34 + 0 \cdot 27 = 34$	0	34	$34/2 = 17$
3	1	$17 + 1 \cdot 27 = 44$	0	44	$44/2 = 22$
4	1	$22 + 1 \cdot 27 = 49$	1	$49 + 1 \cdot 55 = 104$	$104/2 = 52$
5	0	$52 + 0 \cdot 27 = 52$	0	52	$52/2 = 26$

- The subtraction (Step 4) is not needed since $u < n$
- The result is found as MonPro$(27, 27) = 26$

Properties of the Binary CIOS Algorithm

- The binary CIOS algorithm performs 2 multiplications with k-bit numbers in Steps 3a and 3b, requiring $O(k^2)$ operations
- Step 3c is a simple bit-level shift operation, requiring at most $O(k)$
- However, Step 3b does not perform a multiplication when $u_0 = 0$
- Assuming U_0 will be 1 or 0 with uniform probability, we can deduce that, in the average, half of the time Step 3b will be skipped
- Thus, the binary CIOS algorithm requires 1.5 multiplications with k-bit numbers in the average

The Word-Level CIOS Algorithm

- The word-level algorithm scans the words of a, n, and u

$$\begin{aligned} a &= (A_{s-1}A_{s-2}\cdots A_1A_0) \\ n &= (N_{s-1}N_{s-2}\cdots N_1N_0) \\ u &= (U_{s-1}U_{s-2}\cdots U_1U_0) \end{aligned}$$

  for $sw = k$ where w is the word size in bits

- The least significant w bits, in other words, the least significant words (LSW) of a, n, u are A_0, N_0, U_0

- Now we consider the steps of the binary CIOS algorithm, and extend them to word level

- Step 3a is easily extended as $u \leftarrow u + A_i \cdot b$

The Word-Level CIOS Algorithm

- Since Step 3c performs a w-bit right shift, Step 3b should update u so that its LSW is zero
- Let M be a 1-word integer such that the LSW of $u + M \cdot n$ is zero

$$u + M \cdot n = 0 \pmod{2^w} \quad \Rightarrow \quad M = -u \cdot n^{-1} \pmod{2^w}$$

- Since the modulus is 2^w, we only need the LSW of u and n

$$M = U_0 \cdot (-N_0^{-1}) \pmod{2^w}$$

- Interestingly, the identity $r \cdot r^{-1} + (-n') \cdot n = 1$ implies

$$(-n') \cdot n = 1 \pmod{2^w} \quad \Rightarrow \quad -N_0^{-1} = N'_0 \pmod{2^w}$$

- Therefore, $-N_0^{-1}$ is actually equal to N'_0, the LSW of N'

The Word-Level CIOS Algorithm

1: $u \leftarrow 0$
2: **for** $i = 0$ **to** $s - 1$
3a: $u \leftarrow u + A_i \cdot b$
3b: $M \leftarrow U_0 \cdot (-N_0^{-1}) \pmod{2^w}$
3c: $u \leftarrow u + M \cdot n$
3d: $u \leftarrow u/2^w$
4: **if** $u > n$ **then** $u \leftarrow u - n$
5: **return** u

An Example of the Word-Level CIOS Algorithm

- Consider the computation of MonPro$(27, 27)$ using the word-level CIOS algorithm for $n = 55$, $k = 6$, and $w = 3$
- We have $a = 27 = (011011)_2 = (33)_8$ and $b = 27$
- We also have $n = 55 = (110111)_2 = (67)_8$
- Furthermore, $N_0 = 7$ an $N_0' = -N_0^{-1} = -7^{-1} = 1 \pmod 8$

i	A_i	Step 3a (u)	U_0	Step 3b (M)	Step 3c (u)	Step 3d (u)
0	$(3)_8$	$0 + 3 \cdot 27 = 81$	$(1)_8$	$1 \cdot 1 = 1$	$81 + 1 \cdot 55 = 136$	$136/8 = 17$
1	$(3)_8$	$17 + 3 \cdot 27 = 98$	$(2)_8$	$1 \cdot 2 = 2$	$98 + 2 \cdot 55 = 208$	$208/8 = 26$

- The subtraction (Step 4) is not needed since $u < n$
- The result is found as MonPro$(27, 27) = 26$

Properties of the Word-Level CIOS Algorithm

- The word-level CIOS performs 2 multiplications with s word numbers in Steps 3a and 3c, requiring $O(s^2)$ operations
- Step 3b requires a 1-word operation: $O(1)$
- Step 3d is a simple word-level shift operation, requiring at most $O(s)$
- The word-level CIOS algorithm is more efficient than the classical Montgomery algorithm because:
  - It does not require the k-bit (s-word) number n', instead, it only requires computation of the w-bit (1-word) number N_0'
  - The classical Montgomery algorithm requires the k-bit number n'
  - It requires 2 multiplications with k-bit (s-word) numbers
  - The classical Montgomery algorithm requires 3 multiplications with k-bit numbers

Fast Computation of N_0'

- There is an efficient algorithm for computing the one-word integer

$$N_0' = -N_0^{-1} \pmod{2^w}$$

- It is based on a specialized version of the extended Euclidean algorithm for computing the inverse
- The following algorithm computes $x^{-1} \pmod{2^w}$ for an odd x

function ModInverse$(x, 2^w)$

1: $y \leftarrow 1$

2: **for** $i = 2$ **to** w

3: **if** $2^{i-1} < x \cdot y \pmod{2^i}$ **then** $y \leftarrow y + 2^{i-1}$

4: **return** y

An Example Computation of n_0'

- As an example, we compute $23^{-1} \pmod{2^6}$
- Here we have $x = 23$ and $w = 6$.
- We start with $y = 1$

i	2^{i-1}	2^i	y	$x \cdot y \pmod{2^i}$		y
2	2	4	1	$23 \cdot 1 = 3 \pmod 4$	$3 > 2$	$1 + 2 = 3$
3	4	8	3	$23 \cdot 3 = 5 \pmod 8$	$5 > 4$	$3 + 4 = 7$
4	8	16	7	$23 \cdot 7 = 1 \pmod{16}$	$7 \not> 8$	7
5	16	32	7	$23 \cdot 7 = 1 \pmod{32}$	$7 \not> 14$	7
6	32	64	7	$23 \cdot 7 = 33 \pmod{64}$	$33 > 32$	$7 + 32 = 39$

- Thus, we find $y = 39$, implying $23^{-1} = 39 \pmod{64}$
- This is true, since $23 \cdot 39 = 897 = 1 \pmod{64}$

Arithmetic with Special Primes

- Until now we considered modular arithmetic with arbitrary composite or prime numbers
- However, elliptic cryptographic algorithms often use special primes
- For example, the NIST elliptic curves over GF(p) use these primes

Curve	Field prime p
P-192	$2^{192} - 2^{64} - 1$
P-224	$2^{224} - 2^{96} + 1$
P-256	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
P-384	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
P-521	$2^{521} - 1$

Arithmetic with Special Primes

- Similarly, SECG elliptic curves over GF(p) use these primes

Curve	Field prime p
secp192k1	$2^{192} - 2^{32} - 2^{12} - 2^{8} - 2^{6} - 2^{6} - 2^{3} - 1$
secp192r1	$2^{192} - 2^{64} - 1$
secp224k1	$2^{224} - 2^{32} - 2^{12} - 2^{11} - 2^{9} - 2^{7} - 2^{4} - 2 - 1$
secp224r1	$2^{224} - 2^{96} + 1$
secp256k1	$2^{256} - 2^{32} - 2^{9} - 2^{8} - 2^{7} - 2^{6} - 2^{4} - 1$
secp256r1	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
secp384r1	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
secp521r1	$2^{521} - 1$

- Furthermore, Curve25519 also uses a special prime $p = 2^{255} - 19$

Arithmetic with Special Primes

- We can use the existing modular addition and multiplication algorithms, including the classical Montgomery algorithm and the binary and word-level versions of the CIOS algorithm for arithmetic with these special primes
- However, these algorithms are designed for arbitrary primes
- They will not be as efficient as the algorithms that are designed and optimized for particular primes
- Several algorithms proposed for such special primes
- These algorithms however work only for the type of primes or for the primes for which they are designed

Arithmetic with NIST Prime P-192

- Consider the NIST prime P-192 which is $p = 2^{192} - 2^{64} - 1$
- Assume $w = 64$ and represent a word using A_i
- Assume we are performing reduction modulo this prime
- For example, we can take a number that is the twice the length of p and reduce it mod p
- Such number may appear in computations after a multiplication, and thus, it will need to be reduced mod p

Arithmetic with NIST Prime P-192

- Since p is k bits, we can take a $2k$-bit integer a and reduce it mod p
- Every $2k$-bit integer a can be represented using 6 words

$$a = A_5 2^{320} + A_4 2^{256} + A_3 2^{192} + A_2 2^{128} + A_1 2^{64} + A_0$$

- We can also use the compact notation $a = (A_5 A_4 A_3 A_2 A_1 A_0)$
- After the reduction mod $p = 2^{192} - 2^{64} - 1$, the result will be 3 words
- $b = a \pmod{p}$ implies $b = (B_2 B_1 B_0)$

Reduction with NIST Prime P-192

- In order to obtain the reduced number b, we first compute

$$\begin{aligned} T &= (A_2 A_1 A_0) \\ S_1 &= (0 \; A_3 A_3) \\ S_2 &= (A_4 A_4 \; 0) \\ S_3 &= (A_5 A_5 A_5) \end{aligned}$$

- Then, we compute $b = a \pmod{p}$ using modular addition

$$b = (B_2 B_1 B_0) = T + S_1 + S_2 + S_3 \pmod{p}$$

- This is a special reduction algorithm that works only for this p

Reduction with NIST Prime P-192

- How can we prove that this reduction is correct?

- For $r = 2^{64}$, we can express the prime as $p = r^3 - r - 1$
- Given $a = A_5 r^5 + A_4 r^4 + A_3 r^3 + A_2 r^2 + A_1 r + A_0$, the reduction operation can be expressed as a polynomial reduction

$$b = A_5 r^5 + A_4 r^4 + A_3 r^3 + A_2 r^2 + A_1 r + A_0 \pmod{r^3 - r - 1}$$

- Using a computer algebra tool, we obtain b as

$$b = (A_5 + A_4 + A_2) r^2 + (A_5 + A_4 + A_3 + A_1) r + (A_5 + A_3 + A_0)$$

Reduction with NIST Prime P-192

- By rearranging the terms, we can write

$$\begin{array}{c|cccc}
r^2 & A_2 & 0 & A_4 & A_5 \\
r & A_1 & A_3 & A_4 & A_5 \\
1 & A_0 & A_3 & 0 & A_5 \\
\hline
 & T & S_1 & S_2 & S_3
\end{array}$$

- Therefore, we obtain

$$\begin{aligned}
T &= (A_2 A_1 A_0) \\
S_1 &= (0\ A_3 A_3) \\
S_2 &= (A_4 A_4\ 0) \\
S_3 &= (A_5 A_5 A_5)
\end{aligned}$$

- Finally:

$$b = T + S_1 + S_2 + S_3 \pmod{p}$$

Reduction with NIST Prime P-521

- These specialized algorithms require fewer terms to be computed and added if the binary expansion of the prime contains fewer 1s, i.e., the power of 2 terms in its expression
- Therefore, primes with few 1s or few power of 2 terms are preferred
- The NIST P-521 is one such prime: $p = 2^{521} - 1$
- Primes of this form are called the Mersenne primes
- Reduction with such primes is significantly simpler

Reduction with NIST Prime P-521

- Assume A_1 and A_0 are 521-bit integers
- Every integer less than p^2 can be represented

$$a = A_1 \cdot 2^{521} + A_0$$

- The compact representation $a = (A_1 A_0)$
- Consider the reduction operation $b = a \pmod{p}$
- The expression for b is simply given as

$$b = A_1 + A_0 \pmod{p}$$

- A single modular addition suffices to obtain $b = a \pmod{p}$

Reduction with NIST Prime P-521

- The expression for b is also easily proven by assigning $r = 2^{521}$
- Therefore, $p = r - 1$ and $a = A_1 r + A_0$
- Now we can compute b using polynomial reduction

$$\begin{aligned} b &= A_1 r + A_0 \pmod{r-1} \\ &= A_1(r-1) + A_1 + A_0 \pmod{r-1} \\ &= A_1 + A_0 \pmod{r-1} \end{aligned}$$

- Therefore, $b = A_1 + A_0 \pmod{p}$
- In other words, the reduction requires a single modular addition

Reduction with Primes of the Form $2^k - c$

- The prime for Curve25519 is given as $p = 2^{255} - 19$
- Primes of the form $p = 2^k - c$ are commonly used in cryptography, where c is a 1-word integer,
- If we assign $r = 2^k$, we get $p = r - c$
- Consider the $2k$-bit number $a = A_1 r + A_0$
- To compute $b = a \pmod{p}$, we perform polynomial reduction

$$\begin{aligned} b &= A_1 r + A_0 \pmod{r - c} \\ &= A_1(r - c) + c\,A_1 + A_0 \pmod{r - c} \\ &= c\,A_1 + A_0 \pmod{r - c} \end{aligned}$$

- The final reduced value is computed as $b = c\,A_1 + A_0 \pmod{p}$

Generalized Mersenne Numbers

- The reduction algorithm for prime $p = 2^{192} - 2^{64} - 1$ is invented by Jerome Solinas, who developed several specialized reduction algorithms for the NIST primes
- For example, consider $p = 2^{224} - 2^{96} + 1$
- Given the 32-bit numbers A_i, the mod p reduction of the $2k$-bit number $a = (A_{13}A_{12} \cdots A_1 A_0)$ is accomplished as

$$
\begin{aligned}
T &= (A_6 A_5 A_4 A_3 A_2 A_1 A_0) \\
S_1 &= (A_{10} A_9 A_8 A_7 \; 0 \; 0 \; 0) \\
S_2 &= (0 \; A_{13} A_{12} A_{11} \; 0 \; 0 \; 0) \\
D_1 &= (A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7) \\
D_2 &= (0 \; 0 \; 0 \; 0 \; A_{13} A_{12} A_{11}) \\
b &= T + S_1 + A_2 - D_1 - D_2 \pmod{p}
\end{aligned}
$$