Efficient implementation of ECC in software

+

IBE in Sensor Networks

Ricardo Dahab
rdahab@ic.unicamp.br
Institute of Computing, UNICAMP

SCSW4/IPAM/UCLA - Dec 4-8, 2006

Crypto in Brazil

- Small, active group with international presence and collaboration: Barreto (USP), Dahab (UNICAMP), van de Graaf (UFMG), López (UNICAMP), Nakahara (UNISantos), Terada (USP).
- Efficient implementations, pairings, quantum crypto, symmetric algorithms.
- Reasonable amount of cooperation.
  - Yearly Workshop on Crypto Algorithms and Protocols (WCAP) - next coming up in August 2007.
  - National PKI working groups.
  - Co-supervision.

Outline

- Improvements for software multiplication in $\mathbb{F}_{2^m}$, with timings for the MMX platform.
- Pairings in wireless sensor networks

Part I - Software Multiplication in $\mathbb{F}_{2^m}$

(with J. López and others)

1. Motivation for finite field arithmetic
2. Definition of $\mathbb{F}_{2^m}$
3. Bases (Polynomial and normal bases)
4. Multiplication in $\mathbb{F}_{2^m}$
   - Squaring
   - Basic methods
   - The comb. (López-Dahab) method
   - An improvement of the comb. method

Motivation

- Efficient implementation of binary field arithmetic is fundamental to the performance of common cryptographic mechanisms such as those based on elliptic curves.
- Of particular importance is the multiplication operation since it is a major building block in cryptographic applications.
- Elliptic curve cryptography can be implemented using binary fields (NIST recommended five binary fields and ten curves).
  $\mathbb{F}_{2^m}, m = 163, 233, 283, 409, 571.$
- The speed of the multiplication algorithm depends greatly on the particular basis used to represent field elements. The two most common choices of bases for $\mathbb{F}_{2^m}$ are normal and polynomial.
- For software multiplication, polynomial bases present some computational advantages over normal bases.

The binary field $\mathbb{F}_2 = GF(2) = \{0, 1\}$

- $(\mathbb{F}_2, +)$ is an abelian group (0)
- $(\mathbb{F}_2^*, \cdot)$ is an abelian group (1)
- For each $a, b, c \in \mathbb{F}_2$, $a \cdot (b + c) = a \cdot b + a \cdot c$
- $a + b = a$ `xor` b = $a + b \mod 2$

$$
\begin{array}{ccccc}
0 & + & 0 & = & 0 \\
1 & + & 0 & = & 1 \\
0 & + & 1 & = & 1 \\
1 & + & 1 & = & 0
\end{array}
$$

The binary field $\mathbb{F}_{2^m} = GF(2^m)$

- The finite field $\mathbb{F}_{2^m}$ is a vector space of dimension m on $\mathbb{F}_2 = \{0, 1\}$ called *binary field* or *characteristic-two finite field*.
- Let $\{\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{m-1}\}$ be a basis for $\mathbb{F}_{2^m}$ on $\mathbb{F}_2$.
  Then, any element $a \in \mathbb{F}_{2^m}$ can be represented as $a = \sum_{i=0}^{m-1} a_i \alpha_i$, where $a_i \in \mathbb{F}_2$.
- The binary vector associated to the element a is $(a_0 a_1 a_2 \ldots a_{m-1})$.
- Therefore, the binary field $\mathbb{F}_{2^m}$ can be viewed as the set of 2^m binary strings of length m:

$$\mathbb{F}_{2^m} = \{(a_0 a_1 \ldots a_{m-1}) : a_i \in \{0, 1\}\}$$

Example: $\mathbb{F}_{2^8}$

- Let $f(x) = x^8 + x^4 + x^3 + x + 1$
  be an irreducible polynomial of degree 8

  $\mathbb{F}_{2^8} = \{ (b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0) | b_i \in \{0, 1\} \}$

  Given $a, b \in \mathbb{F}_{2^8}$
  $a = x^7 + x^3 + 1 = (10001001) = \texttt{0x89}$
  $b = x^6 + x^5 + x^2 = (01100100) = \texttt{0x64}$
- Addition: $a + b$ = ? xor

  $x^7 + x^6 + x^5 + x^3 + x^2 + 1 = (11101101) = \texttt{0xED}$
- Multiplication: $a \times b$ = ?

  $x^{13} + x^{12} + x^8 + x^6 + x^2$ mod $f(x)$=
  $x^7 + x^5 + x^4 + x^3 + x^2 + 1 = (10111101) = \texttt{0xBD}$

The binary field $\mathbb{F}_{2^m}$

- For $a, b \in \mathbb{F}_{2^m}$, $a + b = a \oplus b$.
- $\Rightarrow a \in \mathbb{F}_{2^m}$, $2a = 0$ (characteristic 2)
- For each $a, b \in \mathbb{F}_{2^m}$, $(a + b)^2 = a^2 + b^2$

$\mathbb{F}_{2^m}$ is a finite field where:

- = +

$$\sqrt{a + b} = \sqrt{a} + \sqrt{b}$$

Bases for $\mathbb{F}_{2^m}$

- Polynomial bases: $\{x^{m-1}, x^{m-2}, \cdots, x, 1\}$,
  $x^i = (0, \cdots, 0, 1, 0, \cdots, 0)$
  Arithmetic is efficient (hardware/software).
- Normal Bases: $\{\beta, \beta^{2^1}, \beta^{2^2}, \ldots, \beta^{2^{m-1}}\}$, $\beta \in \mathbb{F}_{2^m}$.
  Squaring is fast, but multiplication is, in general, a complex operation.

  - Optimal normal bases
  - Gaussian normal bases

  Multiplication is based on permutations (good for hardware).

Squaring in Normal bases

- Let $a = (a_0 a_1 \ldots a_{m-2} a_{m-1})$ be an element in $\mathbb{F}_{2^m}$. Then,

$$\begin{aligned} a^2 &= (a_0\beta + a_1\beta^{2^1} + \ldots + a_{m-1}\beta^{2^{m-1}})^2 \\ &= a_0\beta^{2^1} + a_1\beta^{2^2} + \ldots + a_{m-1}\beta^{2^m} \ (\beta^{2^m} = \beta) \\ &= a_{m-1}\beta + a_0\beta^{2^1} + \ldots + a_{m-2}\beta^{2^{m-2}} \\ &= (a_{m-1} a_0 a_1 \ldots a_{m-2}) \end{aligned}$$

Squaring in Normal bases

- Let $a = (a_0 a_1 \ldots a_{m-2} a_{m-1})$ be an element in $\mathbb{F}_{2^m}$. Then,

$$\begin{aligned} a^2 &= (a_0\beta + a_1\beta^{2^1} + \ldots + a_{m-1}\beta^{2^{m-1}})^2 \\ &= a_0\beta^{2^1} + a_1\beta^{2^2} + \ldots + a_{m-1}\beta^{2^m} \ (\beta^{2^m} = \beta) \\ &= a_{m-1}\beta + a_0\beta^{2^1} + \ldots + a_{m-2}\beta^{2^{m-2}} \\ &= (a_{m-1} a_0 a_1 \ldots a_{m-2}) \end{aligned}$$

  Squaring a field element is a simple right rotation of its binary representation. (fast operation in hardware)

Multiplication in Normal Bases

- *Optimal normal bases* and *Gaussian normal bases* were introduced in order to reduce the hardware complexity of the field multiplication operation.
- Normal optimal bases: these bases use one or two permutations for multiplication, but they exist for some values of m.
- Gaussian normal bases: these bases use two or more permutations and are defined for some prime values of m. The NIST recommended five binary fields $\mathbb{F}_{2^m}$ for $m = 163, 233, 283, 409$, which can be implemented using Gaussian normal bases.
- Software Multiplication using Gaussian Normal Bases. R. Dahab, D.Hankerson, F. Hu, M. Long, J. López and A. Menezes. *IEEE Transactions on Computers*, vol. 55 No. 8, August 2006.

Polynomial Bases

- Let $f(x)$ be an irreducible binary polynomial of degree m, then:

$$\mathbb{F}_{2^m} = \mathbb{F}_2[x]/ < f(x) >$$

$$\text{Basis : } \{x^{m-1}, x^{m-2}, \cdots, x, 1\}$$

- Let $a \in \mathbb{F}_{2^m}$. Then, $a = \sum_{i=0}^{m-1} a_i x^i = (a_{m-1} a_{m-2} \cdots a_1 a_0)$
- NIST (National Institute of Standards and Technology), 2000.
  - $\mathbb{F}_{2^{163}}$, $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$
  - $\mathbb{F}_{2^{233}}$, $f(x) = x^{233} + x^{74} + 1$
  - $\mathbb{F}_{2^{283}}$, $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$
  - $\mathbb{F}_{2^{409}}$, $f(x) = x^{409} + x^{87} + 1$
  - $\mathbb{F}_{2^{571}}$, $f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$

Software Multiplication in $\mathbb{F}_{2^m}$

$$p = \sum_{i=0}^{m-1} a_i x_i, \quad p(x) = A_{t-1} \cdots A_1 A_0$$

- A_i : w-bits; $W = 8, 16, 32, 64, 128$, $t = \lceil m/W \rceil$.
- $p =$ 
- $A_i << j$, `word<< j`
- $A_i >> j$, `word>> j`
- $A_i \oplus B_j$, `word1 ^ word2` (XOR)
- $A_i \odot B_j$, `word1 & word2` (AND)
- $A_i \vee B_j$, `word1 || word2` (OR)

Squaring in $\mathbb{F}_{2^m}$

- Let a be an element in $\mathbb{F}_{2^m}$
  $a = \sum_{i=0}^{m-1} a_i x^i$
  $a^2 = \sum_{i=0}^{m-1} a_i x^{2i}$
- Example:
  $a = x^7 + x^5 + x^3 + 1 = (10101001)$
  $a^2 = x^{14} + x^{10} + x^6 + 1 =$ (0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1)
- Table look-up: $T : 2^8 \rightarrow 2^{16}$
  For each $b = (b_7 b_6 \cdots b_0)$ compute $T[b] = (0b_7 0b_6 \cdots 0b_1 0b_0)$

Squaring in $\mathbb{F}_{2^m}$ ...

Input: $a \in \mathbb{F}_{2^m},\ a = (A_{t-1} \cdots A_0),\ t = \lceil m/32 \rceil$.

Output: $a^2 \bmod f$.

1. Precomputation: for each $b = (b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0)$ compute

$$T[b] = (0b_7 0b_6 0b_5 0b_4 0b_3 0b_2 0b_1 0b_0)$$

2. **for** $i = 0$ **to** $t - 1$ **do**

   2.1 $A_i \leftarrow (b_3|b_2|b_1|b_0),\ b_j$ a byte

   2.2 $C_{2i} \leftarrow (T[b_1]T[b_0]),\ C_{2i+1} \leftarrow (T[b_3]T[b_2])$

3. $c(x) \leftarrow (C_{2t-1} \cdots C_1 C_0) \bmod f$

4. **return** (c)

Modular Reduction

- $\mathbb{F}_{2^{163}}$, $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, w = 32 bits
- $f(x) = 0x \boxed{8}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{c9}$

$$c(x) = a(x) \cdot b(x) \bmod f(x)$$

- $c = c_{324}x^{324} + \cdots + c_1 x + c_0$
- $c = \boxed{C_{10}}\, C_9 C_8 C_7 C_6\ C_5 C_4 C_3 C_2 C_1 C_0$
- $C_{10} = c_{351}x^{351} + c_{350}x^{350} + \cdots + c_{320}x^{320}$
- $m_{10} := c_{351}x^{188} + c_{350}x^{187} + \cdots + c_{320}x^{157}$
- $c' = c + m_{10} \cdot f(x) \equiv c \bmod f(x)$
- $c \equiv c' \equiv \boxed{0}\, C_9 C_8 C_7 C'_6\ C'_5 C'_4 C_3 C_2 C_1 C_0 \bmod f(x)$

$\vdots$

- $c \equiv C''_5 C''_4 C''_3 C''_2\ C''_1 C''_0 \bmod f(x)$

Fast Reduction Modulo

$$f(x) = x^{163} + x^7 + x^6 + x^3 + 1$$

Input: A binary polynomial c of degree at most ≤ 324.
Output: $c \bmod f$.

1. **for** i **from** 10 **downto** 6 **do**
   - 1.1 $T \leftarrow C[i]$
   - 1.2 $C[i-6] \leftarrow C[i-6] \oplus (T \ll 29)$
   - 1.3 $C[i-5] \leftarrow C[i-5] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$
   - 1.4 $C[i-4] \leftarrow C[i-4] \oplus (T \gg 28) \oplus (T \gg 29)$
2. $T \leftarrow C[5] \gg 3$
3. $C[0] \leftarrow C[0] \oplus (T \ll 7) \oplus (T \ll 6) \oplus (T \ll 3) \oplus T$
4. $C[1] \leftarrow C[1] \oplus (T \gg 25) \oplus (T \gg 26)$
5. $C[5] \leftarrow C[5] \&$ `0x7`
6. **return** $(C[5], C[4], C[3], C[2], C[1], C[0])$

Polynomial Multiplication in $\mathbb{F}_{2^m}$

- The *shift-and-add* method (binary method), 1995
- The comb method (López-Dahab method, 2000)
- Karatsuba-Ofman method
- Montgomery method ($c = a \cdot b \cdot r^{-1}$)
  Koç-Acar, 1998

Multiplication in $\mathbb{F}_{2^m}$

Given $a, b \in \mathbb{F}_{2^m}$, the product $c = a \cdot b \bmod \mathrm{f}$ can be computed as:

$$\begin{aligned} a \cdot b &= \sum_{i=0}^{m-1} a_i x^i \cdot \sum_{i=0}^{m-1} b_i x^i \pmod{f(x)} \\ &= \sum_{i=0}^{2m-2} r_i \, x^i \pmod{f(x)} \\ &= (r_{2m-2} r_{2m-1} \cdots r_1 r_0) \pmod{f(x)} \\ &= (c_{m-1} c_{m-2} \cdots c_1 c_0) \end{aligned}$$

$$r_i = \begin{cases} \sum_{j=0}^{i} a_j b_{i-j}, & 0 \le i \le m-1, \\ \sum_{j=i-m+1}^{m-1} a_j b_{i-j}, & m \le i \le 2m-2. \end{cases}$$

Example: multiplication in $\mathbb{F}_{2^4}$

Bit-level:

$$
\begin{array}{lcccccccc}
r_6 & = & & & & & & & a_3b_3 \\
r_5 & = & & & & & a_2b_3 & + & a_3b_2 \\
r_4 & = & & & a_1b_3 & + & a_2b_2 & + & a_3b_1 \\
r_3 & = & a_0b_3 & + & a_1b_2 & + & a_2b_1 & + & a_3b_0 \\
r_2 & = & a_0b_2 & + & a_1b_1 & + & a_2b_0 & & \\
r_1 & = & a_0b_1 & + & a_1b_0 & & & & \\
r_0 & = & a_0b_0 & & & & & &
\end{array}
$$

Example: multiplication in $\mathbb{F}_{2^4}$

Bit-level:

$$\begin{array}{lcccccccc}
r_6 & = & & & & & & & a_3b_3 \\
r_5 & = & & & & & a_2b_3 & + & a_3b_2 \\
r_4 & = & & & a_1b_3 & + & a_2b_2 & + & a_3b_1 \\
r_3 & = & a_0b_3 & + & a_1b_2 & + & a_2b_1 & + & a_3b_0 \\
r_2 & = & a_0b_2 & + & a_1b_1 & + & a_2b_0 & & \\
r_1 & = & a_0b_1 & + & a_1b_0 & & & & \\
r_0 & = & a_0b_0 & & & & & &
\end{array}$$

Vector-level:

$$r = a_0[b] \oplus a_1[b << 1] \oplus a_2[b << 2] \oplus a_3[b << 3]$$

The shift-and-add method
Left-to-right

$$
\begin{aligned}
a \cdot b &= \sum_{i=0}^{m-1} a_i x^i b(x) \bmod f(x) \\
&= x \cdot [\cdots x \cdot [x \cdot a_{m-1} b(x)]_f + a_{m-2} b(x)\ ]_f + \cdots]_f \\
&+ a_0 b(x)
\end{aligned}
$$

$$
a = \begin{array}{|c|c|c|c|c|c|c|} \hline a_{m-1} & a_{m-2} & a_{m-3} & \cdots & \cdots & a_1 & a_0 \\ \hline \end{array}
$$

The shift-and-add method
Left-to-right

$$
\begin{aligned}
a \cdot b &= \sum_{i=0}^{m-1} a_i x^i b(x) \bmod f(x) \\
&= x \cdot [\cdots x \cdot [x \cdot a_{m-1} b(x)]_f + a_{m-2} b(x)\ ]_f + \cdots]_f \\
&+ a_0 b(x)
\end{aligned}
$$

$$
a = \boxed{a_{m-1}} \boxed{a_{m-2}} \boxed{a_{m-3}} \boxed{\cdots} \boxed{\cdots} \boxed{a_1} \boxed{a_0}
$$

$c \leftarrow 0$
for i **from** $m-1$ **to** 0 **do**
    $c \leftarrow x \cdot c \bmod f(x)$ (left-shift)
    if $a_i = 1$ **then** $c \leftarrow c + b$
endfor

The shift-and-add method
Right-to-left

$$
\begin{aligned}
a \cdot b &= \sum_{i=0}^{m-1} a_i x^i b(x) \bmod f(x) \\
&= a_0 b(x) + a_1[x \cdot b(x)] + \ldots + a_{m-1}[x \cdot x^{m-2} b(x)]
\end{aligned}
$$

$$
a = \begin{array}{|c|c|c|c|c|c|c|} \hline a_{m-1} & a_{m-2} & a_{m-3} & \cdots & \cdots & a_1 & a_0 \\ \hline \end{array}
$$

The shift-and-add method
Right-to-left

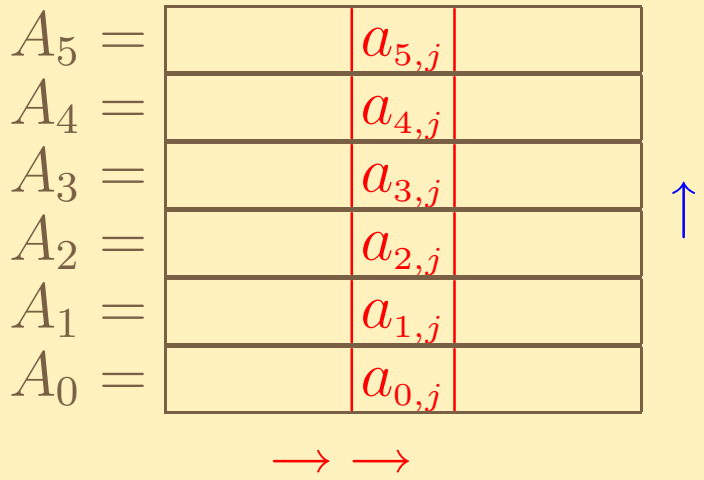
$$
\begin{aligned}
a \cdot b &= \sum_{i=0}^{m-1} a_i x^i b(x) \bmod f(x) \\
&= a_0 b(x) + a_1[x \cdot b(x)] + \ldots + a_{m-1}[x \cdot x^{m-2} b(x)]
\end{aligned}
$$

$$
a = \boxed{a_{m-1}} \boxed{a_{m-2}} \boxed{a_{m-3}} \boxed{\cdots} \boxed{\cdots} \boxed{a_1} \boxed{a_0}
$$

if $a_0 = 1$ **then** $c \leftarrow b$; **else** $c \leftarrow 0$
for i **from** 1 **to** $m-1$ **do**
    $b \leftarrow x \cdot b \bmod f(x)$ ($m-1$ left-shifts)
    if $a_i = 1$ **then** $c \leftarrow c + b$
endfor

The Algorithm of López-Dahab, 2000

$$c(x) = a(x) \cdot b(x)$$



$$p_{c_j}(x) = \sum_{i=0}^{5} a_{i,j} \boxed{x^{32i} b(x)} \text{ (free)}$$

$$a(x) \cdot b(x) = \sum_{j=0}^{31} x^j p_{c_j}(x)$$

The algorithm of López-Dahab left-to-right ...

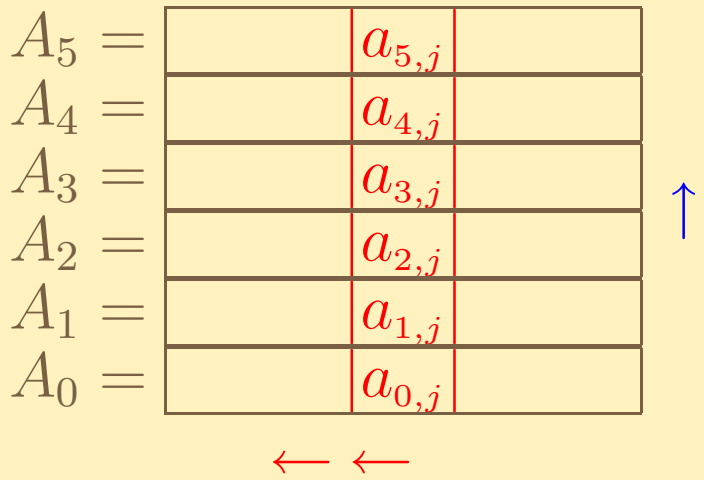
Input: $a, b \in \mathbb{F}_{2^m}$
Output: $c = ab \bmod \mathrm{f}$

1. $t_c \leftarrow 0$
2. **for** $i = 31$ **downto** 1 **do**
   $t_c \leftarrow t_c \oplus p_{c_j}$
   $t_c \leftarrow x \cdot t_c$ (31 left-shifts) $|t_c| \approx 2m$ bits
3. $t_c \leftarrow t_c \oplus p_{c_0}$
4. $c \leftarrow t_c \bmod \mathrm{f}$
5. **return**(c)

$$a(x) \cdot b(x) = \sum_{j=0}^{31} x^j p_{c_j}(x)$$

Improvement I
right-to-left

$$c(x) = a(x) \cdot b(x)$$



$$p_{c_j}(x) = \sum_{i=0}^{5} a_{i,j} \boxed{x^{32i}[x^j b(x)]} \text{ (free)}$$

$$a(x) \cdot b(x) = \sum_{j=0}^{31} p_{c_j}(x)$$

LD Algorithm
right-to-left

Input: $a, b \in \mathbb{F}_{2^m}$
Output: $c = ab \bmod \mathrm{f}$

1. $t_b \leftarrow b,\ t_c \leftarrow 0$
2. **for** $j = 0$ **to** 30 **do**
   $t_c \leftarrow t_c \oplus p_{c_j}(t_b)$
   $t_b \leftarrow x \cdot t_b$ (31 left-shifts) $|t_c| \approx m + 32$ bits
3. $t_c \leftarrow t_c \oplus p_{c_{31}}(t_b)$
4. $c \leftarrow t_c \bmod \mathrm{f}$
5. **return**(c)

Improvement II using precomputation ($w = 4$)

$A_5 =$		$a_{5,j}$	$a_{5,j+1}$	$a_{5,j+2}$	$a_{5,j+3}$	
$A_4 =$		$a_{4,j}$	$a_{4,j+1}$	$a_{4,j+2}$	$a_{4,j+3}$	
$A_3 =$		$a_{3,j}$	$a_{3,j+1}$	$a_{3,j+2}$	$a_{3,j+3}$	
$A_2 =$		$a_{2,j}$	$a_{2,j+1}$	$a_{2,j+2}$	$a_{2,j+3}$	
$A_1 =$		$a_{1,j}$	$a_{1,j+1}$	$a_{1,j+2}$	$a_{1,j+3}$	
$A_0 =$		$a_{0,j}$	$a_{0,j+1}$	$a_{0,j+2}$	$a_{0,j+3}$	

Precomputation : for each u, $0 \leq \mathrm{u} \leq 15$, compute

$$T[u](x) := (u_3x^3 + u_2x^2 + u_1x + u_0)b(x)$$

$$a(x)b(x) = \sum_{j=0}^{7} x^{4j}\{\sum_{i=0}^{5} x^{32i}T[a_{i,j}...a_{i,j+3}](x)\}$$

LD Algorithm with Precomputation

Input: $a, b \in \mathbb{F}_{2^m}$ $[m = 163, W = 32]$
Output: $c = ab \bmod \mathrm{f}$

1. *Precomputation:* for each $u = (u_3u_2u_1u_0)_2$, $0 \leq u \leq 15$,
   $T[u] \leftarrow (u_3x^3 + u_2x^2 + u_1x + u_0)b$
2. $t_c \leftarrow 0$
3. **for** $i = 7$ **downto** 0 **do**
   for $j = 0$ **to** 5 **do**
   $u \leftarrow [A_j >> 4i]_{16} = [u_3u_2u_1u_0]$
   $t_c \leftarrow t_c \oplus T[u]x^{32j}$
   endfor
   if $i \neq 0$ **then** $t \leftarrow t_cx^4$ (7 left-shifts)
4. $c \leftarrow t_c \bmod \mathrm{f}$
5. **return**(c)

Improvement III
shifting 8 bits

$$A_j = \boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}$$

The LD algorithm requires seven left shifts of 4 bits ($t << 4$)

New idea: we modify the order of processing A_j

$$A_j = \boxed{rrrr}\boxed{bbbb}\boxed{rrrr}\boxed{bbbb}\boxed{rrrr}\boxed{bbbb}\boxed{rrrr}\boxed{bbbb}$$

Improvement III
shifting 8 bits

$$A_j = \boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}\boxed{rrrr}$$

The LD algorithm requires seven left shifts of 4 bits ($t << 4$)

New idea: we modify the order of processing A_j

$$A_j = \boxed{rrrr}\boxed{bbbb}\boxed{rrrr}\boxed{bbbb}\boxed{rrrr}\boxed{bbbb}\boxed{rrrr}\boxed{bbbb}$$

The new algorithm requires six left shifts of 8 bits ($t << 8$)
and one left shift of 4 bits

$$ab = (a_1x^4 + a_0)b = a_1bx^4 + a_0b$$

parallel method

Improvement on LD Algorithm with precomputation $(w = 4)$...

1. *Precomputation:* for each $u = (u_3u_2u_1u_0)_2$, $0 \leq u \leq 15$,
   $T[u] \leftarrow (u_3x^3 + u_2x^2 + u_1x + u_0)b$
2. $t_c \leftarrow 0, t_d \leftarrow 0$

3. **for** $i = 3$ **downto** 0 **do**	4. **for** $i = 3$ **downto** 0 **do**
for $j = 0$ **to** 5 **do**	**for** $j = 0$ **to** 5 **do**
$u \leftarrow [A_j >> 8i + 4]_{16}$	$u \leftarrow [A_j >> 8i]_{16}$
$t_c \leftarrow t_c \oplus T[u]x^{32j}$	$t_d \leftarrow t_d \oplus T[u]x^{32j}$
if $i \neq 0$ **then** $t_c \leftarrow t_cx^8$	**if** $i \neq 0$ **then** $t_d \leftarrow t_dx^8$

5. $t_c \leftarrow t_cx^4 \oplus t_d$
6. $c \leftarrow t_c \bmod \mathrm{f}$
7. **return**(c)

Using MMX Instructions on a Pentium 4

Given $a \in \mathbb{F}_{2^{163}}$

$A \ll 8$ is faster than $A \ll 4$

`m1, m0`: registers of 128 bits, a = `m1||m0`

$a \ll 4$:

$\texttt{t1} = \texttt{m1} \gg 60$

$\texttt{m1} = \texttt{m1} \ll 4$

$\texttt{m1} = \texttt{m1} \oplus (\texttt{t1} \ll 64)$

$\texttt{t1} = \texttt{m0} \gg 60$

$\texttt{m1} = \texttt{m1} \oplus (\texttt{t1} \gg 64)$

$\texttt{m0} = \texttt{m0} \ll 4$

$\texttt{m0} = \texttt{m0} \oplus (\texttt{t1} \ll 64)$

$a \ll 8$:

$\texttt{t1} = \texttt{m0} \gg 120$

$\texttt{m1} = \texttt{m1} \ll 8$

$\texttt{m1} = \texttt{m1} \oplus \texttt{tv1}$

$\texttt{m0} = \texttt{m0} \ll 8$

Timings on a Pentium 4 (MMX-128)

work in progress

Timings (in μs) using MMX instructions, with compiler icc, 1.5 GHz.

m	C 32 bits	MMX-64 64 bits	MMX-128 128 bits
163	0.70	0.58	0.49
233	1.21	0.81	0.63
283	1.72	1.16	0.96
409	3.21	1.78	1.49
571	5.02	3.02	2.38

Timings on a Pentium 4 HT (MMX-128) work in progress

Timings (in μs) using MMX instructions, with compiler icc, 3.0 GHz.

m	C 32 bits	MMX-64 64 bits	MMX-128 128 bits
163	0.40	0.25	0.23
233	0.61	0.37	0.29
283	0.82	0.49	0.48
409	1.89	0.92	0.71
571	3.28	1.50	1.20

Timings on a Xscale Processor (wMMX-64) work in progress

Timings (in μs) on an Xscale PAX-270 (550 MHz) using wMMX instructions, with compiler gcc.

Operations	C	ASM	wMMX
Multiplication	9.80	4.89	3.78
Multiplication by constant	8.56	3.26	2.44

Timings for ECDSA

Timings (in μs) on an Xscale Intel PAX-270 (550 MHz) for binary curves over $\mathbb{F}_{2^{163}}$, using gcc:

Binary curves	C	assembly	wMMX-64
Key generation	4074	2202	1737
Signature generation	4257	2399	1926
Signature verification	117432	7408	5997

Koblitz curves	C	assembly	wMMx-64
Key generation	2984	-	-
Signature generation	3153	-	-
Signature verification	7565	-	-

Timings for ECDSA

Timings (in μs) on an Xscale Intel PAX-270 (550 MHz) for NIST prime curves over $\mathbb{F}_{p_{192}}$ and $\mathbb{F}_{p_{256}}$:

Prime curves P-192	GMP	assembly	wMMX-64
Key generation	4363	1302	1298
Signature generation	4555	1515	1509
Signature verification	14751	4553	4539

Prime curves P-256	GMP	assembly	wMMX-64
Key generation	12603	3381	3384
Signature generation	12836	3734	3736
Signature verification	42069	11664	11681

Timings for ECDSA

Timings (in μs) on a Pentium 4 (3.0 GHz) for binary curves over $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$, icc, in C language:

Operations	B-163	K-163
Key generation	177	131
Signature generation	191	144
Signature verification	641	342

Operations	B-233	K-233
Key generation	411	274
Signature generation	428	291
Signature verification	1556	674

Timings for ECDSA

Timings (in μs) on a Pentium 4 (3.0 GHz) for NIST prime curves over $\mathbb{F}_{p_{192}}$ and $\mathbb{F}_{p_{256}}$:(GMP), icc:

Operations	P-192	P-256
Key generation	327	1000
Signature generation	341	1020
Signature verification	1117	3346

Conclusions

- We have presented a new technique for speeding up the López-Dahab method for some special computational platforms (MMX, wMMX).
- On the Pentium 4, the improved method for multiplication (using MMX-128) is about 30%-60% faster than the C (32 bits) version.
- On the Xscale processor, the improved method for multiplication (using wMMX-64 and $m = 163$) is about 54% faster than the C (32 bits) version.
- The running times of our software implementation show that a **multiplication by a constant** is about 50% faster than a full multiplication, given a large look-up table (256 finite field elements).
- We have introduced a parallel method for field multiplication.

Part II - Pairings in sensor networks

(with Leonardo B. Oliveira and others)

- Wireless sensor networks.
- Why pairings in WSNs?
- A protocol for key establishment using pairings.
- Preliminary timing results.

Wireless sensor networks

- Ad hoc networks of small sensor nodes with limited resources and one or more base stations (BSs), which collects data from sensors.
- Used for monitoring purposes.
- Usually disposable. Once deployed, they are left in the environment until battery runs out.
- Data fusion to avoid too much transmission, a very costly operation.
- Classification:
  - Planar $\times$ hierarchical (clustered around cluster-heads);
  - Homogeneous $\times$ heterogeneous.

Vulnerabilities in Wireless sensor networks

Security solutions are crucial but also a challenging task.

- Wireless communication;
- Restricted resources;
- Lack of physical protection;
- Often deployed in hostile environments;
- Prone to routing attacks as well as direct physical tampering.

Energy concerns, then space considerations, are much more relevant than speed.

Security of WSNs

- Best known proposals rely on symmetric-key solutions, but
  - the network does not exist until nodes are deployed;
  - a node has to find its neighboring nodes using as few as possible (secure) broadcasts;
  - global keys are a tempting but very bad idea: compromising one node compromises all.
- usually assume base station is secure against physical attack (but not against impersonation).

Security of WSNs

- Key distribution (after deployment) is a problem:
  too many key to store $\times$ which keys to store.
- $\Rightarrow$ Pre-distribution is preferable (e.g. LEAP), but
  LEAP needs a predistributed key shared among all nodes assumed secure during the t initial time units of the network operation;
  LEAP assumes that once erased, key cannot be recovered from memory.
- Resource constraints prevent use of RSA/DSA-based solutions.

Public-key Crypto in WSNs

Relevant questions

- Is it desirable?
- In what form?
- Is it possible?
- If possible, is it practical?

Public-key Crypto in WSNs

- Key distribution would be, of course, greatly simplified but,
  - public-keys cannot all be expected to be preloaded into the nodes;
  - authentication of public-keys cannot rely on PKIs.
- This seems to favour an identity-based approach, with the base station (not really) in the role of the Trusted Authority.
- David Malan's pioneer implementation of ECC (binary fields, polynomial bases), and other many recent works seem to indicate that there is an interest in PKC for WSNs.
- See

  K. Hoeper and G. Gong, *Bootstrapping Security in Mobile Ad Hoc Networks Using Identity-Based Schemes with Key Revocation*, CACR Tech Report 2006-04,
  for a broader discussion of IBC applied to MANETS.

A protocol for key establishment in WSNs using IBE

IDs being broadcast by nodes (e.g. A and B):

1. $A \Rightarrow \mathcal{G}_A : \quad id_A, \mathsf{nonce}$
   $B \Rightarrow \mathcal{G}_B : \quad id_B, \mathsf{nonce}$

Neighboring nodes (e.g., M from $\mathcal{G}_A$ and N from $\mathcal{G}_B$) use received IDs to generate public keys (e.g. P_A and P_B) and distribute secret keys:

2. $M \rightarrow A : \quad id_A, \mathsf{enc}_{P_A}(\mathsf{sign}_{S_M}(id_M \mid id_A \mid k_{M,A} \mid \mathsf{nonce}))$
   $N \rightarrow B : \quad id_B, \mathsf{enc}_{P_B}(\mathsf{sign}_{S_N}(id_N \mid id_B \mid k_{N,B} \mid \mathsf{nonce}))$

Secure exchange of information between neighboring nodes (e.g., A and M, and N and B)

3. $A \rightarrow M : \quad id_A, id_M, m, \mathsf{mac}_{k_{M,A}}(id_A \mid id_M \mid m \mid \mathsf{nonce}')$
   $N \rightarrow B : \quad id_N, id_B, m, \mathsf{mac}_{k_{N,B}}(id_N \mid id_B \mid m \mid \mathsf{nonce}')$

Pairing computation for WSNs

Ongoing work for TinyTate:

- MICAz node (ATmega128 microcontroller, 8-bit/7.38 MHz processor, 4KB SRAM, 128KB flash memory) running TinyOS;
- Tate pairing over $E/\mathbb{F}_q : y^2 = x^3 + x$;
- q is a 256-bit prime;
- embedding degree $k = 2$

Pairing computation for WSNs

(Preliminary cost results)

TinyTate Pairing		
Time (seconds)	**RAM (bytes)**	**ROM (bytes)**
30.21	1,831	18,384

- Space measures encompass the entire software system (TinyOS + TinyECC + TinyTate).
- There is some room left for protocols!

Furher crypto-related work at UNICAMP

Cooperation with code-optimization group.

- Márcio Juliato, Guido Araujo, Julio López and Ricardo Dahab. *A Custom Instruction Approach for Hardware and Software Implementations of Finite Field Arithmetic over* $\mathbb{F}_{2^{163}}$ *using Gaussian Normal Bases*. Journal of VLSI Signal Processing Systems, Accepted for publication, 2006.
  Also in IEEE 2005 Conf. on Field-Programmable Technology (FPT' 05), Singapore, December 2005.