

---

# *Generic Efficient Arithmetic Algorithms for PAFFs (Processor Adequate Finite Fields) and Related Algebraic Structures*

*ROBERTO AVANZI  
(joint work with PREDĂ MIHĂILESCU)*

*IEM – University of Duisburg–Essen*

*partially supported by the E.U. via the AREHCC Project  
<http://www.arihcc.org> and <http://www.arihcc.com>*

# Outline of Talk

---

*In full screen mode, click on titles to go to corresponding slide.  
On each slide click on [ $\Leftarrow$ ] to return here.*

- Groups for Discrete Logarithm Systems
- Finite Extension Fields
  - Exponentiation Algorithms
  - Modular and Polynomial Reduction
- An Implementation
- Conclusions
- Any questions?
- Appendices

# Groups for Discrete Logarithm Systems

---

Some groups ( $G$ ) used

- Multiplicative group of a finite field.
  - Prime field  $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ .
  - Binary field  $\mathbb{F}_{2^n}$ .
  - Extension field  $\mathbb{F}_{p^m}$  (e.g. XTR).
- Additive group of an Abelian Variety.
  - Elliptic Curve.
  - Jacobian of Hyperelliptic Curve.

In these groups we have to compute  $g^n$  for an element  $g$  and an integer  $n$  (or  $[n]g$  for Additive groups).

# *Finite Extension Fields: Problem*

---

*Which level of performance can be  
attained from finite fields  
(while being as general as possible)?*

# Finite Extension Fields: Concepts

- $p = \text{odd prime}$ .  $m \geq 1$ . Exactly one field  $\mathbb{F}_{p^m}$ .
- $f(X) \in \mathbb{F}_p[X]$  irreducible polynomial  $\text{deg } m$   
 $\Rightarrow \mathbb{F}_{p^m} \cong \frac{\mathbb{F}_p[X]}{(f(X))}$ .
- *model* of  $\mathbb{F}_{p^m}$   $\Leftarrow$  a choice  $f(X)$  and  
a *basis* of  $\mathbb{F}_{p^m}$  as an  $\mathbb{F}_p$ -vector space.  
 $\Rightarrow \{1, X, X^2, \dots, X^{m-1}\}$ .
- *Frobenius automorphism*:  $\varphi : \mathbb{F}_{p^m} \rightarrow \mathbb{F}_{p^m}, x \mapsto x^p$ .  
 $\mathbb{F}_p$ -linear map  $\Leftrightarrow m \times m$  matrix over  $\mathbb{F}_p$ .
- Elements of  $\mathbb{F}_{p^m}$   $\Leftrightarrow$  polys in  $X$  over  $\mathbb{F}_p$  of  $\text{deg} < m$ .

# Finite Extension Fields: *Basic Operations*

---

An element of  $\mathbb{F}_{p^m}$  is represented by a polynomial of degree less than  $m$  over  $\mathbb{F}_p$  – or as a vector of length  $m$ .

Set  $x = X \bmod f(X)$ :  $\mathbb{F}_{p^m}$  has a power basis in  $x$ , i.e. any  $g \in \mathbb{F}_{p^m}$  can be written as  $g = \sum_{i=0}^{m-1} g_i x^i$  with  $g_i \in \mathbb{F}_p$ .

- Addition and subtraction: componentwise (no-brainer).
- Multiplication: split into multiplication of polynomials and polynomial reduction modulo  $f(X)$ . We also need to multiply elements of  $\mathbb{F}_p$ .
- Exponentiation: different algorithms.

# *Finite Extension Fields: Model*

---



Critical for performance:

# *Finite Extension Fields: Model*

---



Critical for performance:

A. Reduction modulo  $f(X)$ .

# Finite Extension Fields: *Model*

---



Critical for performance:

- A. Reduction modulo  $f(X)$ .
- B. The Frobenius.

# Finite Extension Fields: *Model*

---



Critical for performance:

- A. Reduction modulo  $f(X)$ .
- B. The Frobenius.
- C. Multiplication in the base field (modular reduction).

# Finite Extension Fields: *Model*

---



Critical for performance:

- A. Reduction modulo  $f(X)$ .
- B. The Frobenius.
- C. Multiplication in the base field (modular reduction).

An ideal *model* should allow fast implementation of the three above operations.

# Finite Extension Fields: *Model*

---



Critical for performance:

- A. Reduction modulo  $f(X)$ .
- B. The Frobenius.
- C. Multiplication in the base field (modular reduction).

An ideal *model* should allow fast implementation of the three above operations.



Additionally, need fast exponentiation **exploiting the relative speed ratios of the basic operations.**

# Finite Extension Fields: *History*

- **1997** Mihăilescu – FSE4, Haifa, Israel, January 22.  
Take  $f(X) = X^m - b$ .  
Frobenius and reduction mod  $f(X)$  easy.
- **1997** A. Lenstra:  $f(X) = \text{cyclotomic polynomial} = \frac{X^{m+1} - 1}{X - 1}$   
of degree  $m$  with  $m + 1$  a prime, get ONB of type I. Excellent performance, little choice of fields.  
Frobenius free.
- **1998, 2001** Bailey and Paar (after Mihăilescu) take  
 $f(X) = X^m - b$ . B&P choose  *$p$  quasi Mersenne*, i.e.  $2^k - c$  for  
 $c \ll 2^{k/2}$  – implementation on 8-bit microcontrollers.
- **1999**: Kobayashi *et. al.* – Subfield Elliptic Curves.
- **2001–2003**: We.

# Exponentiation Algorithms

Want to compute  $g^n$ . Write  $n$  in base  $p$ :

$$n = \sum_{i=0}^{m-1} n_i p^i \quad \text{with} \quad 0 \leq n_i < p .$$

Then

$$\begin{aligned} g^n &= \prod_{i=0}^{m-1} (g^{n_i})^{p^i} = \prod_{i=0}^{m-1} \varphi^i(g^{n_i}) \\ &= \varphi(\cdots \varphi(\varphi(g^{n_{m-1}}) \cdot g^{n_{m-2}}) \cdots g^{n_1}) \cdot g^{n_0} \end{aligned}$$

$\Rightarrow$  compute the powers  $g^{n_i}$ , then use a Horner scheme.

# Exponentiation Algorithms

---

▶ ▶ In Lenstra's cyclotomic fields paper.

2 Algorithms:

- ▶ One simple (but efficient).
- ▶ One based on Yao's multi-exponentiation algorithm.

The second algorithm offers a speedup of 20% over first method for 500–2000 bit fields .

# Exponentiation Algorithms

---

▶ ▶ In our paper.

▶ Algorithm 1: “Baby-Windows Giant-Windows”.

To be used when Frobenius not fast.

Based on aggressive precomputations.

Related to BGMW, actually a particular case of Pippenger’s multi-exponentiation algorithm.

▶ Algorithm 2: “Abusing the Frobenius”.

To be used when Frobenius is fast or negligible.

Used a very high number of Frobenius operations and reduces squarings to a minimum.

Can be seen as “interleaved” exponentiation (see GLV).

# Comparing 1

Operation counts for one exponentiation in  $\mathbb{F}_{4086122041^m}$

m	Ratios		Simple		Algorithm 1				Algorithm 2			
	$\frac{\tau_S}{\tau_M}$	$\frac{\tau_\varphi}{\tau_M}$	M/S/ $\varphi$	Cost	$\ell$	#S	M/S/ $\varphi$	Cost	$\ell$	#S	M/S/ $\varphi$	Cost
4	.83	.42	60	87.95	1	31	63	90.95	4	15	33.4	69.70
			31				31				32	
			3				3				20.5	
6	.76	.25	90	114.83	2	47	87	111.83	4	15	47	80.30
			31				31				32	
			5				5				34.2	
8	.80	.24	120	146.48	2	47	111	137.48	5	31	60.2	95.53
			31				31				32	
			7				7				40.4	
16	.68	.15	240	264.30	3	72	182	213.34	5	31	106.4	143.27
			31				41				32	
			15				15				86.7	
32	.57	.12	480	501.07	4	119	295	334.88	6	63	190.7	223.75
			31				63				32	
			31				31				155.6	
64	.63	.06	960	983.67	5	188	510	576.25	7	127	347	384.00
			31				97				32	
			63				63				280.5	
128	.65	.04	1920	1943.15	6	317	881	985.86	7	127	632	668.43
			31				161				32	
			127				127				565.5	

# Comparing 2

$\mathbb{F}_{4086122041}^m$ : Comparison with other exponentiation algorithms.

$m$ ( $u = 32$ )	4	6	8	16	32	64	128
Square and Multiply	169.41	241.16	332.00	603.48	1095.11	2313.61	4709.75
Sliding Windows	144.73 ( $\ell = 4$ )	197.24 ( $\ell = 4$ )	269.40 ( $\ell = 4$ )	459.36 ( $\ell = 5$ )	778.23 ( $\ell = 5$ )	1633.77 ( $\ell = 6$ )	3278.05 ( $\ell = 7$ )
Lenstra's Method	87.95	114.83	146.48	264.30	501.07	983.67	1943.15
Algorithm 1	90.95 ( $\ell = 1$ )	111.83 ( $\ell = 2$ )	137.48 ( $\ell = 2$ )	213.34 ( $\ell = 3$ )	334.88 ( $\ell = 4$ )	576.25 ( $\ell = 5$ )	985.86 ( $\ell = 6$ )
Algorithm 2	69.70 ( $\ell = 4$ )	80.30 ( $\ell = 4$ )	95.53 ( $\ell = 5$ )	143.27 ( $\ell = 5$ )	223.75 ( $\ell = 6$ )	384.00 ( $\ell = 7$ )	668.43 ( $\ell = 7$ )

# Modular reduction(s)

---

- Polynomial reduction modulo  $f(X)$ .
- Reduction of integers modulo  $p$ .

For the first, we take  $f(X) = X^m - b$  (Algorithm 2!)

In a product, a lot of sums  $\sum a_i b_i$ .

Two ways for speeding-up:

- Speed up modulo  $p$  reduction.
- Reduce modulo  $p$  less often.

For (i) we suggest either Montgomery's reduction or a variant of Barrett to have *generality and speed*.

# *Incomplete reduction*

---

**Problem:** compute  $\sum a_i b_i \bmod p$   
with  $p$ ,  $a_i$  and  $b_i$  all  $\leq w$  bits long.

# *Incomplete reduction*

---

**Problem:** compute  $\sum a_i b_i \bmod p$   
with  $p$ ,  $a_i$  and  $b_i$  all  $\leq w$  bits long.

**Fact:** Reduction algorithms efficient only if  $p$  is restricted (Bailey–Paar) or if the input is less than  $p \cdot 2^w$ .

# Incomplete reduction

---

**Problem:** compute  $\sum a_i b_i \bmod p$   
with  $p$ ,  $a_i$  and  $b_i$  all  $\leq w$  bits long.

**Fact:** Reduction algorithms efficient only if  $p$  is restricted (Bailey–Paar) or if the input is less than  $p \cdot 2^w$ .

**Idea #1:** add all the  $a_i b_i$ , and reduce only at end (Lim–Hwang).

# Incomplete reduction

---

**Problem:** compute  $\sum a_i b_i \bmod p$   
with  $p$ ,  $a_i$  and  $b_i$  all  $\leq w$  bits long.

**Fact:** Reduction algorithms efficient only if  $p$  is restricted (Bailey–Paar) or if the input is less than  $p \cdot 2^w$ .

**Idea #1:** add all the  $a_i b_i$ , and reduce only at end (Lim–Hwang).

**Problem:** must make  $p$  smaller to avoid intermediate results larger than  $p \cdot 2^w$ .

# Incomplete reduction

---

**Problem:** compute  $\sum a_i b_i \bmod p$   
with  $p$ ,  $a_i$  and  $b_i$  all  $\leq w$  bits long.

**Fact:** Reduction algorithms efficient only if  $p$  is restricted (Bailey–Paar) or if the input is less than  $p \cdot 2^w$ .

**Idea #1:** add all the  $a_i b_i$ , and reduce only at end (Lim–Hwang).

**Problem:** must make  $p$  smaller to avoid intermediate results larger than  $p \cdot 2^w$ .

**Idea #2:** Reduce when overflow.

# Incomplete reduction

---

**Problem:** compute  $\sum a_i b_i \bmod p$   
with  $p$ ,  $a_i$  and  $b_i$  all  $\leq w$  bits long.

**Fact:** Reduction algorithms efficient only if  $p$  is restricted (Bailey–Paar) or if the input is less than  $p \cdot 2^w$ .

**Idea #1:** add all the  $a_i b_i$ , and reduce only at end (Lim–Hwang).

**Problem:** must make  $p$  smaller to avoid intermediate results larger than  $p \cdot 2^w$ .

**Idea #2:** Reduce when overflow.

**Problem:** Reduction still too frequent if  $p$  close to  $2^w$ .  
Montgomery's representation does not allow adding reduced and unreduced elements.

# *Incomplete reduction*

---

**Solution:** replace a result with one  $\equiv \text{mod } p$ .

Keep most significant digits (i.e. the bits from the  $w$ -th upwards) always  $< p$ .

This is compatible with all reduction methods.

# Incomplete reduction

**Solution:** replace a result with one  $\equiv \pmod{p}$ .

Keep most significant digits (i.e. the bits from the  $w$ -th upwards) always  $< p$ .

This is compatible with all reduction methods.

► **Algorithm 3:** *Incomplete reduction*

*Compute:*  $x$  with  $x \equiv \sum_1^t a_i b_i \pmod{p2^w}$  and  $0 \leq x \leq p2^w$

*Notation:*  $x = (x_{\text{hi}}, x_{\text{lo}})_{2^w}$ .

1. Initialise  $x \leftarrow a_0 b_0$   
for  $i = 1$  to  $t$  do {
2.  $x \leftarrow x + a_i b_i$
3. if overflow or  $x_{\text{hi}} \geq p$  then  $x_{\text{hi}} \leftarrow x_{\text{hi}} - p$  }

# *Incomplete reduction*

---

A deceptively simple idea, but...

- ▶ No restrictions on prime  $p$  apart from length!
- ▶ We can pick the size of the prime to make it fit the characteristics of the processor on which we work.
- ▶ Moreover, 32 bits is good for many environments.

So, what happens if we implement extension fields with  $f(X) = X^m - b$  and incomplete reduction?

## *With respect to prime fields...*

---

We have  $1/5$  to  $1/3$  of the multiplications. And the multiplications themselves are faster because

- The time spent reducing is much less (time spent in modular reduction for PAFFs of fixed characteristic grows linearly with size, quadratically for prime fields of the size of our interest).
- There are less carries to propagate.

# An implementation

*Some Timings.*

Bits (B)	gmp-4.0		MGFez	
	without MMX	with MMX	large prime	small prime
128	0.5		0.11	
192	1.75		0.37	
256	3.5		0.52	
512	22	7	2.8	2.1
1024	160	45	13.2	9.4
2048	1158	340	68.2	49.4

Milliseconds on a 400 Mhz Pentium III.

large prime = about 32 bits

small prime = about 29 bits, avoids incomplete red.

# Conclusions

---

- Comparison of different exponentiation algorithms for finite extension fields allows reduction of number of field operations.
- For finite extension fields “better” algorithms than for prime fields.
- Field operations are faster. PAFF are a very good special choice, which is still rather general.
- Implementation results show impressive performance.
- Analogues for curves with efficient endomorphisms, trace-zero varieties... joint work with Tanja Lange (Bochum). But this is another story.

*Any questions?*



# Appendix: Exponentiation Algorithms

---

► From Lenstra's cyclotomic fields paper.

Let  $u = \text{bit-length of } p$ .

1.  $y \leftarrow g$ . Set  $x_i \leftarrow 1$  for  $i = 0, \dots, m - 1$ .
2. For  $j = 0, \dots, u - 1$  do
  - (a) For  $i = 0, \dots, m - 1$  do
  - (b) If  $j$ -th bit of  $n_i$  is  $= 1$  then  $x_i \leftarrow x_i y$  (i.e  $x_i \leftarrow x_i g^{2^j}$ ).
  - (c)  $y \leftarrow y^2$
3. Now each  $x_i = g^{n_i}$ . Then  $g^n = \prod_{i=0}^{m-1} \varphi^i(g^{n_i})$ .

This deceptively simple algorithm performs very well.

# Appendix: Exponentiation Algorithms

► Lenstra's suggestion of using Yao.

**Example:** Let  $n = (320312)_4$  in radix 4 (coeffs 0, 1, 2 and 3).

Want  $g^n$ . Observe that

$$(320312)_4 = (10)_4 + 2 \times (10001)_4 + 3 \times (100100)_4.$$

1st, compute  $\pi(1) = g^{(10)_4}$ ,  $\pi(2) = g^{(10001)_4}$ ,  $\pi(3) = g^{(100100)_4}$ .

Then  $g^n = \pi(1) \cdot (\pi(2))^2 \cdot (\pi(3))^3$ .

If some  $g^{4^j}$  precomputed, several powers of  $g$  can be computed quickly. For example  $g^{n_0}, g^{n_1}, \dots$  and then use  $g^n = \varphi(\dots \varphi(\varphi(g^{n_{m-1}}) \cdot g^{n_{m-2}}) \dots g^{n_1}) \cdot g^{n_0}$

Speedup 20% over other method for 500–2000 bit fields .

# Appendix: Exponentiation Algorithms

## ► Algorithm 1: Baby-Windows Giant-Windows.

Goal: Computation of  $g^n$  in  $\mathbb{F}_{p^m}^\times$ ,  $0 \leq n < p^m$ .

Hypothesis: Frobenius  $\varphi : x \mapsto x^p$  fast.

1. Write  $n = \sum_{i=0}^{m-1} n_i p^i$  with  $0 \leq n_i < p$  and  $u = \lceil \log_2 p \rceil$ .
2. Choose  $\ell$  and put  $r = \lceil u/\ell \rceil$ .
3. Write  $n_i = \sum_{k=0}^{r-1} n_{i,k} (2^\ell)^k \quad \forall i$  with  $0 \leq n_{i,k} < 2^\ell$ .
4. [BS] Compute  $g^{j 2^{\ell k}}$  for  $0 \leq k < r$  and  $1 \leq j < 2^\ell$ .
5. [GS]  $g^n = \varphi(\cdots \varphi(\varphi(g^{n_{m-1}}) \cdot g^{n_{m-2}}) \cdots g^{n_1}) \cdot g^{n_0}$  where  
 $g^{n_i} = \prod_{k=0}^{r-1} g^{n_{i,k} (2^\ell)^k}$  with  $\leq r - 1$  Muls.

# Appendix: Exponentiation Algorithms

## ► Algorithm 2: Abusing the Frobenius.

Required: very fast (negligible) Frobenius.

Write  $n = \sum_{i=0}^{m-1} n_i p^i$ , with  $0 \leq n_i < p$ , and

$n_i = \sum_{j=0}^{K'-1} n_{ij} 2^{j\ell}$ , for some  $\ell \ll u = \lceil \log_2 p \rceil$ ,  $K = \lceil u/\ell \rceil$ .

Change the order of the operations

(collect terms with same exponent  $2^{j\ell}$ )

$$g^n = \prod_{i=0}^{m-1} \varphi^i(g^{n_i}) = \prod_{i=0}^{m-1} \prod_{j=0}^{K-1} \varphi^i(g^{n_{ij} 2^{j\ell}}) = \prod_{j=0}^{K-1} \left( \prod_{i=0}^{m-1} \varphi^i(g^{n_{ij}}) \right)^{2^{j\ell}}.$$

*Products in (...) computed one after the other,  
a square-and-multiply loop yields  $g^n$ .*

# Diffie-Hellman Protocol

Alice

1. secretly picks

$$a < \#\langle g \rangle$$

2. computes  $h_1 = g^a$

3. transmits  $h_1$

4. computes

$$h_2^a$$

Bob

1. secretly picks

$$b < \#\langle g \rangle$$

2. computes  $h_2 = g^b$

3. transmits  $h_2$

4. computes

$$h_1^b$$

The diagram consists of two horizontal lines representing Alice and Bob. Two arrows cross each other: one from Alice to Bob and one from Bob to Alice. Below the crossing, the equation  $= g^{ab} =$  is written, indicating that both parties arrive at the same common key.

**Common Key:** the group element  $k = g^{(ab)} \in \langle g \rangle \subseteq G$