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

Roberto Maria Avanzi[1]* and Preda Mihăilescu[2]

[1] Institut für Experimentelle Mathematik, University of Duisburg-Essen
Ellernstrasse 29, D-45326 Essen
`mocenigo@exp-math.uni-essen.de`
[2] Universität Paderborn
FB 17, D-33095 Paderborn.
`preda@upb.de`

Abstract. In the past years several authors have considered finite fields extensions of odd characteristic optimised for a given architecture to obtain performance gains. The considered fields were however very specific. We define a *Processor Adequate Finite Field* (PAFF) as a field of odd characteristic $p < 2^w$ where w is a CPU related *word length*. PAFFs have several attractive properties for cryptography. In this paper we concentrate on arithmetic aspects. We present some algorithms usually providing better performance in PAFFs than in prime fields and in previously proposed instances of extension fields of comparable size.

Keywords: Finite extension fields, exponentiation algorithms, modular reduction, discrete logarithm systems.

1 Introduction

Most public key cryptosystems are based on the difficulty of the discrete logarithm (DL) problem: *If g is a generator of a group G and $h \in G$, find an integer n with $g^n = h$.* Commonly used groups are: the *multiplicative group* of a finite field, the rational point group of an elliptic curve [13, 23] over a finite field or the group of rational divisor classes of an hyperelliptic curve [14]. *Binary fields*, i.e. fields of characteristic 2, and *prime fields*, i.e. $\mathbb{F}_p \cong \mathbb{Z}/p\mathbb{Z}$, are the most commonly used finite fields, but in principle *any* finite field can be used. Hence, one may choose the best field to optimize the *performance* on a target architecture, provided that the security prerequisites are satisfied.

In a cryptosystem designed around the multiplicative group of a finite field, the fundamental computation is the *exponentiation*: We present several algorithms to perform it in rather general extension fields.

* Partially supported by the Commission of the European Communities through the Fifth Framework Programme under Contract IST-2001-32613.

One of our motivations for doing the present work was the lack of a comprehensive study of exponentiation algorithms for the type and size of fields we are interested in. Some of the methods which we present are much faster than those which are usually deployed. The techniques can be applied *mutatis mutandis* to compute scalar products in elliptic curves, in Jacobians of hyperelliptic curves having efficiently computable endomorphisms, or trace zero varieties [17, 3]: see § 3.5.

The DL problem in large enough finite fields seems hard [1, 25, 30] (see also [35, § 5]). DL based cryptographic schemes have been recently standardized also over generic finite fields. Fields other than prime and binary are already being considered by the industrial community: For example the XTR cryptosystem [35], which uses an extension of degree 6 of a prime field. The usage of more general extension fields $\mathbb{F}_{p^{6m}}$ is mentioned in [35], and investigated in [19], where however the structure of the extension field $\mathbb{F}_{p^{6m}}$ is not exploited to speed up the operations over the original system: the prime field $\mathbb{F}_p$ is merely replaced by $\mathbb{F}_{p^m}$. The methods presented here, together with the approach taken in [33], can be used to deliver fast XTR-like systems over the fields $\mathbb{F}_{p^{6m}}$. With the present contribution we hope to stimulate further research.

The structure of the paper is the following. In the next section we give a general introduction to finite extension field arithmetic and review some previous work. The exponentiation techniques are presented in Section 3. Section 4 is devoted to modular reduction. In Section 5, implementation data and benchmarks show that PAFFs can deliver up to 20 times better performance than prime fields of comparable size.

2 General Setting

Let p be an odd prime. It is known that for any $m \geq 1$, there is up to isomorphism exactly one finite field $\mathbb{F}_{p^m}$ with p^m elements. If $f(X) \in \mathbb{F}_p[X]$ is an irreducible polynomial of degree m, then $\mathbb{F}_{p^m} \cong \mathbb{F}_p[X]/(f(X))$. A *model* of $\mathbb{F}_{p^m}$ is given by a choice of $f(X)$ together with a *basis* of $\mathbb{F}_{p^m}$ as an $\mathbb{F}_p$-vector space. The *Frobenius automorphism* $\varphi : \mathbb{F}_{p^m} \to \mathbb{F}_{p^m}$ given by $x \mapsto x^p$ is an $\mathbb{F}_p$-linear map. With respect to the chosen basis φ may be represented by a $m \times m$ matrix with entries in $\mathbb{F}_p$.

Let $x = X \bmod f(X)$ and assume that $\mathbb{F}_{p^m}$ has a power basis, i.e. any element $g \in \mathbb{F}_{p^m}$ can be written as $g = \sum_{i=0}^{m-1} g_i x^i$ with the $g_i \in \mathbb{F}_p$. Let $n \in \mathbb{N}$ be an exponent of the size of p^m and

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

be its p-adic expansion. The exponentiation can be rewritten as

$$g^n = \prod_{i=0}^{m-1} \, (g^{n_i})^{p^i} = \prod_{i=0}^{m-1} \, \varphi^i(g^{n_i}), \tag{2}$$

thus reducing the problem to that of computing the powers g^{n_i}. It is usually much faster to compute m powers of the same base g to different u-bit exponents than only one power to a mu-bit exponent, as many common computations can be done only once. (Also, on a parallel device several units can compute many powers g^{n_i} simultaneously, but we are not concerned here with this situation.)

The model of the field $\mathbb{F}_{p^m}$ is of the utmost importance, since it influences the performance of the system. In particular we wish that:

A. Reduction modulo the defining polynomial $f(X)$ be *easy*.
B. The Frobenius matrix be sparse.
C. Modular reduction in the base field be particularly fast.

A. Lenstra proposed in [18] a model based on irreducible cyclotomic polynomials $f(X) = (X^{m+1} - 1)/(X - 1) \bmod p$, with $m + 1$ a prime. This leads to so called optimal normal bases of type I, on which the Frobenius maps act like permutation matrices. These fields are very practical, yet quite *scarce* because of the restrictions that $\ell = m + 1$ be a prime and that p be *primitive* modulo ℓ.

Bailey and Paar [4], following Mihăilescu [22], observed that taking $f(X)$ of the form $f(X) = X^m - b$ – a *binomial* – gives a much larger choice of fields while still fulfilling properties **A** and **B**. The corresponding power basis is called a *binomial basis*. The only restriction is that $\mathbb{F}_p$ must contain some $m-$th root of unity, i.e. $m|(p - 1)$. An element b such that $X^m - b$ is irreducible is quickly found. The Frobenius map is still relatively efficient, even though less than for optimal normal bases. All one has to do is to precompute first c_1 and J such that $X^p \bmod (X^m - b) = c_1 X^J$, then c_1^t for $1 < t < m$. Then $\varphi(x) = c_1 x^J$ and $\varphi(x^t) = c_1^t x^{Jt \bmod m}$. The matrix of φ is the product of a permutation matrix and of a diagonal one. A Frobenius operation in $\mathbb{F}_{p^m}$ costs m multiplications in $\mathbb{F}_p$.

For efficient implementation in restricted environments, Bailey and Paar restricted the characteristic p to be of the form $2^k - c$ for $c \ll 2^{k/2}$, i.e. a *quasi Mersenne prime*, to adopt the reduction algorithm from [20, §14.3.4]. Extension fields defined by irreducible binomials and of quasi Mersenne characteristic are usually called *optimal extension fields* (OEF).

Definition. *A* Processor Adequate Finite Field *(PAFF) is field of odd characteristic $p < 2^w$ were w is some processor related* word length.

The algorithms in Section 4 show that the gains achieved in modular reduction by means of OEFs can essentially be obtained for all PAFFs.

3 Exponentiation Algorithms

Let G be the multiplicative group of the field $\mathbb{F}_q$ where $q = p^m$, p a prime and $m > 1$. For $g \in G$ and $n \in \mathbb{N}$ we want to compute g^n. Just a few of the methods which can be used are: square and multiply [16] with the usual binary development or with the non-adjacent form (NAF, see [28]); unsigned sliding windows [20]; sliding windows across NAFs [2]; and the wNAF [32], which is called also *signed sliding windows*. These algorithms assume only that group multiplication and in some cases also inversion are available. In their basic forms, they do not take advantage of the Frobenius φ. This is what we are going to do.

In the next two Subsections we shall assume that φ can be computed in a reasonable time – i.e. of the order of one multiplication in G – but we do not require its cost to be negligible. Thus the requirement **B** of Section 2 is not stringent. If $\mathbb{F}_{p^m}$ is represented by a non binomial polynomial basis, in general the Frobenius will act on the basis like a non-sparse $m \times m$ matrix, and applying it costs like a schoolbook multiplication.

Based upon (2), the Horner scheme for the evaluation of g^n is

$$g^n = \prod_{i=0}^{m-1} \varphi^i(g^{n_i}) = \varphi(\cdots \varphi(\varphi(g^{n_{m-1}}) \cdot g^{n_{m-2}}) \cdots g^{n_1}) \cdot g^{n_0} \tag{3}$$

and requires $m-1$ multiplications and Frobenius evaluations.

3.1 Right-to-Left Square-and-Multiply

This algorithm was suggested in [18] and uses m temporary variables $x_0, \ldots, x_{m-1}$, which are initially set to 1. The elements g, g^2, $g^4, \ldots, g^{2^u}$ are computed by repeated squarings. When g^{2^j} is computed, for each i, $0 \le i \le m-1$, if the j-th bit of n_i is equal to 1 then x_i is multiplied by g^{2^j}. At the end $x_i = g^{n_i}$ and g^n is recovered by means of (3).

This method requires $u-1$ squarings, on average $m(u/2-1)$ multiplications (for each variable x_i the first multiplication can be replaced by an assignment) and $m-1$ Frobenius operations. This deceptively simple algorithm performs very well.

A. Lenstra [18] claims that the exponentiation method of Yao [36] as improved by Knuth [16] is 20% faster. Our implementations confirm this for fields of about 500 to 2000 bits. In the next two Subsections we shall describe even faster methods.

3.2 Baby-Windows, Giant-Windows

We consider again (3). We call the representation in base p the subdivision of n in *giant windows*, as the exponent is first read through windows of size $\log_2 p$, and then each n_i will be scanned by smaller windows. The exponents n_i are limited by a (relatively) small power of 2 and we must calculate all the g^{n_i}. A first approach would consist in precomputing all powers g^{2^j} for $2^j \le \max\{n_i\}$; Then, for each i, we obtain g^{n_i} by multiplying together the g^{2^j} where j runs through the bits of n_i equal to one. This is the core idea of the *baby windows*, which read the exponents in a radix 2^ℓ, for some integer $\ell \ge 1$.

Fix a small integer ℓ – the *width* of the baby windows. Let u be the bit-length of p, so $2^u > p > 2^{u-1}$ and write $u = K\ell + R$, with $R < \ell$. A 2^u-bit word will be subdivided in K' baby windows, with $K' = K$ if $R = 0$ and $K' = K + 1$ otherwise.

Upon developing the exponents n_i in base 2^ℓ, i.e. $n_i = \sum_{j=0}^{K'-1} n_{ij} 2^{j\ell}$ where all $n_{ij} \in \{0, 1, \ldots, 2^\ell - 1\}$, we have

$$g^{n_i} = \prod_{\substack{0 \le j < K' \\ n_{ij} \ne 0}} g^{n_{ij} 2^{j\ell}} . \tag{4}$$

If the values $g^{t\,2^{j\ell}}$ for $0 \le j < K'$ and $1 \le t \le 2^\ell - 1$ are given, then each g^{n_i} can be computed with just $K' - 1$ multiplications. In fact, if $R \ne 0$, for $j = K$ we shall only need the values with $1 \le i \le 2^R - 1$ for $j = K$, so the last window is smaller. A similar approach is described in [9] for fields of even characteristic.

After ℓ has been selected (this is discussed later) one precomputes the set $\Im = \left\{ g^{i\,2^{j\ell}} : 0 \le j < K'; 1 \le i \le 2^\ell - 1 \right\}$. Then one computes all the values g^{n_i}, each by at most $K' - 1$ multiplications of elements of $\Im$. Last, g^n is obtained by (3).

Algorithm 1: ***Baby–windows giant–windows exponentiation***

```
1. Expand $n$ $p$-adically: $n = \sum_{i=0}^{m-1} n_i p^i$ with $0 \le n_i < p$ (if not already
   given in that form).
2. Compute (or retrieve from a storage) the set
     $\Im = \left\{ g^{i\,2^{j\ell}} : 0 \le j < K'; 1 \le i \le 2^\ell - 1 \right\}$.
3. Compute $g^n$ by means of formula (3) where $g^{n_{d-1}}, \ldots . g^{n_0}$ are com-
   puted using (4) by multiplying together elements of $\Im$.
```

The algorithm just described can be seen as a particular case of Pippenger's exponentiation algorithm: see Stam's Ph.D. Thesis for an accessible description of the latter [34].

To choose the optimal value of ℓ note that the cardinality of $\Im$ is $v := K(2^\ell - 1) + (2^R - 1) - 1$ so $(v - 1)/2$ squarings and $(v + 1)/2$ multiplications are required in Step 2. Step 3 uses $(K - 1)m$ or Km multiplications, according to $R = 0$ or $R \ne 0$. One of the factors in (4) is $1 = g^0$ with probability $1/2^\ell$ (with probability $1/2^R$ in the last baby–window if $R \ne 0$). Therefore the total cost of the algorithm is roughly

$$f(\ell) = \left(K(2^\ell - 1) + (2^R - 1)\right)\frac{\tau_M + \tau_S}{2} - \tau_S + mK\left(1 - \frac{1}{2^\ell}\right)\tau_M + (m - 1)\tau_\varphi$$

where τ_M, τ_S, and τ_φ denote the timings for a multiplication, a squaring and a Frobenius. It is tempting to optimize analytically the baby-window size by determining the minimum of $f(\ell)$: however this would require the evaluation of trascendental functions. In the applications the sizes of the exponents are bounded, and only few integer values of ℓ are admissible. Hence the best strategy to pick the optimal baby-window size consists in counting the amount of average operations required for several consecutive values of ℓ until a minimum is found. Many examples of this strategy are found in Table 1.

3.3 Abusing the Frobenius

We return to the problem of evaluating g^n in $\mathbb{F}_{p^m}$, but this time we require that computing the Frobenius is fast. We begin again by writing $n = \sum_{i=0}^{m-1} n_i p^i$, with $0 \le n_i < p$, and $n_i = \sum_{j=0}^{K'-1} n_{ij} 2^{j\ell}$, for some window size $\ell \ll u = \lceil \log_2 p \rceil$. Put $K = \lceil u/\ell \rceil$. While writing down the expansion of g^x we immediately change the order of the operations:

$$g^n = \prod_{i=0}^{m-1} \varphi^i(g^{n_i}) = \prod_{i=0}^{m-1} \prod_{j=0}^{K-1} g^{n_{ij} 2^{j\ell} p^i} = \prod_{j=0}^{K-1} \left(\prod_{i=0}^{m-1} g^{n_{ij} p^i} \right)^{2^{j\ell}} . \tag{5}$$

Then for $j = 0, 1, \ldots, K-1$ we define the quantities

$$\Pi_j := \prod_{i=0}^{m-1} g^{n_{ij} p^i} = \prod_{i=0}^{m-1} \varphi^i(g^{n_{ij}})$$

which can be evaluated by a Horner scheme in φ, so that a second Horner scheme (in the guise of a square-ℓ-times-and-multiply loop) yields $g^n = (\cdots(\Pi_{K-1}^{2^\ell} \cdot \Pi_{K-2})^{2^\ell} \cdots \Pi_1)^{2^\ell} \cdot \Pi_0$.

Algorithm 2: *Frobenius-abusing exponentiation*

INPUT: An element g of $\mathbb{F}_{p^m}$, an exponent $n = \sum_{i=0}^{m-1} n_i p^i$ with $0 \le n_i < p$, and a window length ℓ.
OUTPUT: g^n.

```
1. Compute (or retrieve) $g^2, \ldots, g^{2^\ell - 1}$.
   Set $u \leftarrow \lceil \log_2 p \rceil$ and $K \leftarrow \lceil u/\ell \rceil$.
   Write $n_i = \sum_{j=0}^{K-1} n_{ij} 2^{j\ell}$ for $0 \le i < m$, with $0 \le n_{ij} < 2^\ell$.
   Set $x \leftarrow 1$.
   for $j = K-1$ down to 0 do {
2.     $\Pi \leftarrow 1$
       for $i = m-1$ down to 0 do {
3.         $\Pi \leftarrow \Pi \cdot g^{n_{ij}}$
4.         if $i \ne 0$ then $\Pi \leftarrow \varphi(\Pi)$ }
5.     $x \leftarrow x \cdot \Pi$
6.     if $j \ne 0$ then $x \leftarrow x^{2^\ell}$ }
7. return($x$)
```

We now write down the operation count. In Step 1, $2^{\ell-1}$ multiplications and $2^{\ell-1} - 1$ squarings are required. In Steps 3 and 4, an expected amount $mK(1-2^{-\ell})$ of multiplications and, respectively, $(m-1)K$ Frobenius operations are performed. In Step 6 $(K-1)\ell$ squarings are done. Hence the total is $\big(2^{\ell-1} + mK(1-2^{-\ell})\big)M + (2^{\ell-1} - 1 + (K-1)\ell)S + (m-1)K\varphi$. This algorithm bears

Table 1. Operation counts for one exponentiation in $\mathbb{F}_{4086122041^m}$

m	Ratios		Simple (§ 3.1)		Algorithm 1				Algorithm 2			
	$\frac{\tau_S}{\tau_M}$	$\frac{\tau_\varphi}{\tau_M}$	M/S/φ	Cost	ℓ	$\#\Im$	M/S/φ	Cost	ℓ	$\#\Im$	M/S/φ	Cost
4	.83	.42	60 / 31 / 3	87.95	1	31	63 / 31 / 3	90.95	4	15	33.4 / 32 / 20.5	69.70
6	.76	.25	90 / 31 / 5	114.83	2	47	87 / 31 / 5	111.83	4	15	47 / 32 / 34.2	80.30
8	.80	.24	120 / 31 / 7	146.48	2	47	111 / 31 / 7	137.48	5	31	60.2 / 32 / 40.4	95.53
16	.68	.15	240 / 31 / 15	264.30	3	72	182 / 41 / 15	213.34	5	31	106.4 / 32 / 86.7	143.27
32	.57	.12	480 / 31 / 31	501.07	4	119	295 / 63 / 31	334.88	6	63	190.7 / 32 / 155.6	223.75
64	.63	.06	960 / 31 / 63	983.67	5	188	510 / 97 / 63	576.25	7	127	347 / 32 / 280.5	384.00
128	.65	.04	1920 / 31 / 127	1943.15	6	317	881 / 161 / 127	985.86	7	127	632 / 32 / 565.5	668.43

the name "Frobenius-abusing exponentiation" because of the large amount of applications of φ, whereas the amount of other group operations is much smaller than with Algorithm 1.

Remark 1. If a binomial basis is used computing a power of φ is as costly as computing φ. All it takes is to precompute the constants and permutation map associated to φ^a: If $\varphi(x) = c_1 x^J$ then $\varphi^a(x) = c_1^a x^{J^a \bmod m}$. Hence if some $n_{ij} = 0$ in Step 3 the application of the Frobenius to Π can be delayed until a nonzero coefficient is found or $i = 0$. This non-trivial optimisation is most effective if ℓ is small.

3.4 Comparing the Exponentiation Algorithms

Table 1 contains the expected operations counts and costs of the three above exponentiation algorithms in a typical set of examples. We take $p = 4086122041$ so that $u = \lceil \log_2 p \rceil = 32$, and let m vary. $\mathbb{F}_{p^m}$ is, roughly, a $32\,m$-bit field. All the fields have been represented by binomial bases. The labels $\Im$, M, S, φ denote the number of memory registers used, and the amount of multiplications, squarings and Frobenius operations required. The relative speed ratios of squarings and Frobenius operations with respect to multiplications have been taken from our highly optimised library `MGFez`, which uses also the other arithmetic improvements described in this paper, by timing it on an Intel Pentium processor. For each value of m, in Algorithms 1 and 2 the optimal choice of ℓ for speed is used. Total costs are given relative to a multiplication. The results are then compared in Table 2 to the simple square-and-multiply algorithm and the sliding window method (using the optimal window size). The costs are computed with the same weights as before.

Table 2. 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 Subsection 3.1	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 $(\ell = 7)$

For a 1024-bit field ($m = 32$) Algorithm 2 is 24% faster than Algorithm 1. However, if the ratios are different, then Algorithm 2 can be slower. This is the case, for example, if a generic (non binomial) polynomial basis is used: For $m = 32$, in this case we observed $\frac{\tau_S}{\tau_M} = 0.86$ and $\frac{\tau_\varphi}{\tau_M} = 2.13$ – then the cost of Algorithm 1 is $366.95M$ and that of Algorithm 2 is $505.83M$.

Even though the estimates apply to specific examples of finite fields, the advantages of the methods presented here in general for PAFFs should be obvious.

3.5 On Curves

Algorithms 1 and 2 can easily be adapted to compute scalar products in elliptic curves or in Jacobians of hyperelliptic curves with an efficiently computable endomorphism φ. For *subfield curves* – i.e. curves defined over $\mathbb{F}_{p^r}$ but considered over $\mathbb{F}_{p^m}$ with $r|m$ – this endomorphism is induced by the Frobenius automorphism of the field extension $\mathbb{F}_{p^m}/\mathbb{F}_{p^r}$. To perform an exponentiation in the chosen subgroup G, one uses in conjunction with φ the so-called τ-adic expansion of the exponent, where τ is a complex root of the characteristic polynomial of φ [15, 32], or exponent splitting techniques such as those by Gallant, Lambert and Vanstone [10], Müller [24], and Sica, Ciet and Quisquater [31]. Efficient ad-hoc exponent splitting techniques exist for trace zero varieties [17, 3].

For brevity, we skip further details. We just point out that little needs to be done other than changing the notation from multiplicative to additive. On curves and Jacobians the speed ratios of the operations (addition, doubling, endomorphism) differ greatly from the finite field case. Hence the optimal algorithm must be determined for every case.

4 Polynomial Arithmetic and Mixed Modular Reduction

Multiplication in extension fields is based on two types of modular reduction: reduction modulo the irreducible polynomial $f(X)$ and reduction of integers modulo the characteristic p.

The first one is a well investigated problem: we consider here only the case $f(X) = X^m - b$, which makes the Frobenius easy to compute and thus allows the use of Algorithm 2. If $X^m - b$ is irreducible and $x = X \bmod f(X)$, like in Section 2, we consider $\mathbb{F}_{p^m}$ as a $\mathbb{F}_p$-vector space with basis $[1, x, \ldots, x^{p-1}]$. Every element of $\mathbb{F}_{p^m}$ is represented uniquely as a polynomial in x of degree at most $m-1$ over $\mathbb{F}_p$. Addition is performed componentwise. The multiplication in $\mathbb{F}_{p^m}$ under this representation is induced from the polynomial multiplication in $\mathbb{F}_p[X]$, with the relation $x^m = b$: Write $\alpha = \sum_{i=0}^{m-1} a_i x^i$ and $\beta = \sum_{i=0}^{m-1} b_i x^i$. Then

$$\alpha\beta = \sum_{i=0}^{2m-2} c_i x^i = c_{m-1}x^{m-1} + \sum_{i=0}^{m-2} (c_i + bc_{i+m})\, x^i \ textwhere\ c_j = \sum_{\substack{j+k=i \\ 0 \le j,k < m}} a_j b_k .$$

In what follows w denotes the bit-length of the integer registers of a given computing architecture (for example 32 or 64).

The quantities c_j are computed modulo p: In a straightforward implementation the biggest part of the computation time is spent performing modulo p reductions. To speed up the multiplication in $\mathbb{F}_{p^d}$ we can:

(i) Speed up modulo p reduction.
(ii) Reduce modulo p less often while doing the polynomial arithmetic.

Instance (i) can be addressed in many ways, not equally satisfactory:

– Using the division-with-remainder operations of the CPU or some software macros to that effect [20, §14.3.1]. This solution is very slow [7].
– Barrett ([5] and [20, §14.3.3]) or Quisquater reduction [26, 27]. These methods are fast only for large inputs. See also [7].
– Use quasi-Mersenne primes, which are of the form $2^b \pm c$ with $c \le 2^{b/2}$ (see [20, Algorithm 14.47]).
– Montgomery reduction [21]. Very efficient [7], but employs an alternate representation of the integers. See [20, §14.3.2].
– Modular reduction for generalized Mersenne primes according to [8].

Instance (ii) can be addressed in a simple way for small enough p, i.e. $m \cdot p^2 < 2^{2w}$: in this case several products can be added together and only the final result reduced – the application to schoolbook multiplication is obvious. Another solution is to allow triple precision intermediate operands, but efficient modular reduction for the accumulated result can only be done for specific types of primes (e.g. quasi-Mersenne). The Montgomery and Barrett algorithms cannot be used to reduce triple precision operands modulo a single precision prime, unless one *increases* the precision of all operands – resulting in a big performance penalty.

We shall provide more satisfactory answers to both problems.

4.1 A Modulo p Reduction Algorithm

In this Subsection we descrive a fast application of Barret's [5] modular reduction algorithm to the single precision case, which can be used as an alternative to Montgomery's [21]. The quantity to be reduced modulo p will be denoted by x and shall satisfy $x < p2^w$.

Barret's algorithm in single precision approximates the quotient q of $\lfloor x/p \rfloor$ by the formula $\lfloor \frac{x}{p} \rfloor \approx \lfloor (\lfloor \frac{2^{2w}}{p} \rfloor \cdot x / 2^{2w} \rfloor$, then computes a remainder $r = x - qp$ and decrements it by p until it is $\leq p$: this adjustment needs to be done at most twice. Often a further approximation is done by truncating x, i.e. $\lfloor \frac{x}{p} \rfloor \approx \lfloor (\lfloor \frac{2^{2w}}{p} \rfloor \cdot \lfloor \frac{x}{2^w} \rfloor) / 2^w \rfloor$ with 3 single precision multiplications, whereas the adjustment loop is repeated at most 3 times.

We observe that, when $\lceil \log_2(p) \rceil = w$, we have $\mu = 2^w + \mu_0$ with $0 < \mu_0 < 2^w$ and $q < 2^w$. By this the second approximation of $\lfloor x/p \rfloor$ is calculated by just one multiplication and and one addition. The complexity of the resulting method is comparable to that of Montgomery's.

By adjusting the parameters to the bit-size of the modulus we can perform such a cheap reduction modulo *all* primes, and not only to those with maximal bit length fitting in a word. If $u = \lceil \log_2(p) \rceil$, we replace 2^{2w} by 2^{w+u}. The following approximation of the quotient is used:

$$\left\lfloor \frac{x}{p} \right\rfloor \approx \left\lfloor \left(\left\lfloor \frac{2^{w+u}}{p} \right\rfloor \cdot x \right) / 2^{w+u} \right\rfloor \approx \left\lfloor \left(\left\lfloor \frac{2^{w+u}}{p} \right\rfloor \cdot \left\lfloor \frac{x}{2^u} \right\rfloor \right) / 2^w \right\rfloor .$$

We obtain the following algorithm:

Algorithm 3: ***Modular reduction***

```
INPUT: $p$ prime $\leq 2^w$; $x$ an integer with $x \leq p \cdot 2^w$
OUTPUT: $x$ mod $p$  (optional: $x$ div $p$)
Assume $u$ and $\mu_0$ known, where:  $u$ is s.t. $2^u > p > 2^{u-1}$, and
        $\mu_0 \leftarrow \lfloor 2^{w+u}/p \rfloor - 2^w = \lfloor 2^w(2^u - p)/p \rfloor$
1. $x_1 \leftarrow \lfloor x/2^b \rfloor$
2. $q \leftarrow x_1 + \left\lfloor \frac{x_1 \cdot \mu_0}{2^w} \right\rfloor$        [ note that $q = \lfloor \frac{x_1 \cdot \mu}{2^w} \rfloor$ ]
3. $r \leftarrow x - q \cdot p$
4. while $r \geq p$ do { $r \leftarrow r - p$  (optional: $q \leftarrow q + 1$) }
5. return $r$ (and optionally also $q$)
```

The correctness is obvious. Step 4 is iterated at most 3 times and, practice, it is repeated only 0.5–1.5 times, depending on the prime.

Only Mersenne-type moduli with very small coefficients allow a sensibly faster reduction, such as those of the form $2^m \pm 1$. Note, however, that multiplication in PAFFs is less sensitive to further improvements in modular reduction than multiplication in other algebraic structures, because of incomplete reduction, described in the next Subsection.

4.2 Incomplete Reduction

We return to the problem of computing the sum $\sum_{i=0}^{d-1} a_i b_i \mod p$ given a_i and b_i with $0 \le a_i, b_i < p$, where as always p is a prime smaller than 2^w, w being the single precision word size.

To use Algorithm 3 or Montgomery's reduction procedure [21] at the *end* of the summation, we just have to make sure that all partial sums of $\sum a_i b_i$ are smaller than $p\,2^w$: the number obtained by removing the least significant w bits should stay reduced modulo p. Hence, we add the double precision products a_i, b_i together in succession and we check if there has been an overflow or if the most significant word of the intermediate sum is $\ge p$: if so we subtract p. This requires as many operations as allowing intermediate results in triple precision, but either less memory accesses are needed, or less registers have to be allocated.

Furthermore, at the end we have to reduce a possibly smaller number, making the modular reduction faster than in the case with triple precision intermediate sums and, as already remarked, allowing a bigger choice of algorithms and primes.

Algorithm 4: *Incomplete reduction*

INPUT: p (and μ_0), a_i and b_i for $i = 0, \dots, t$,
OUTPUT: x with $x \equiv \sum_{i=0}^{t} a_i b_i \bmod p2^w$ and $0 \le x \le p2^w$
NOTATION: $x = (x_{\rm hi}, x_{\rm lo})$ is a double precision variable.

```
1. Initialise $x \leftarrow a_0 b_0$
   for $i = 1$ to $t$ do {
2.         $x \leftarrow x + a_i b_i$
3.         if carry or $x_{\rm hi} \ge p$ then $x_{\rm hi} \leftarrow x_{\rm hi} - p$ }
4. return $x$
```

4.3 Fast Convolutions

Like in the case of long integer arithmetic, fast and nested convolution techniques [29, 6] can be used for additional speed-ups. We illustrate the methodology, and the issues, with two examples.

To multiply two degree m polynomials we can use a Karatsuba [12] step to reduce the problem to three multiplications of degree $\lceil m/2 \rceil$ polynomials. It is very well known that this leads to an $O(m^{\log_2 3})$ algorithm. For m small, say $m < 10$ or $m < 30$ depending on the implementation, schoolbook multiplication is faster.

Similar algorithms split their inputs in three or more parts, as the k-points Toom-Cook algorithm (cfr. [16, § 4.4.3]). Its time complexity is $O(m^{\log_k (k+1)})$, but in practice it gives gains only for input sizes steeply increasing with k. Karatsuba's method is the particular case $k = 2$.

Different types of convolutions can be nested, always choosing the optimal one for a given operand size. Since the schoolbook method is the fastest for small operands, it is always used at the bottom level of the recursion, called the *socle*. For many applications the only effective “short convolution” will be the shortest one, i.e. Karatsuba.

Care is required for the decision to keep which intermediate results in double precision. In fact, even if we save modular reductions by doing so, the number of memory accesses will increase and divisions by small constants will have to operate on double precision operands instead of on single precision ones. There are two basic partially delayed modular reduction variants:

– In the first one, all results are kept in double precision and the full modular reduction is done only at the very end.
– In the second variant the modular reduction is always done in the schoolbook socle. The convolutions above the socle operate only on single precision integers.

The first variant uses less CPU multiplications but more memory accesses, hence it is the approach of choice on workstations with large memory bandwidth. The second solution seems the best one on a PC.

5 A Software Experiment

We implemented the algorithms of Sections 3 and 4 in the software library `MGFez` (a quasi-acronym for Middle Galois Field Extensions), previously known as `PAFFlib`. For large extension degrees `MGFez` uses simple convolutions and recognizes automatically small primes, so that even without incomplete reduction the accumulated sums do not exceed 2^{2w}. The current version of `MGFez` works only with $w = 32$ and can be compiled on Pentium, Alpha and POWER processors.

For several values of a parameter B we compare the timings required by `gmp` version 4.0 to perform $a^n \mod b$ where a, b and n are B-bit numbers, against the time required by `MGFez` to compute g^n, where g is an element of a finite extension field with about 2^B elements and $n \approx 2^B$. The manual of `gmp` describes all algorithms employed. For operands of the sizes which we tested, 3 points Toom-Cook convolutions and exact division algorithms are used.

Table 3 contains some sample timings in milliseconds on a 400 Mhz Pentium III. We explain the meanings of the captions:

– Inline asm = `gmp-4.0` compiled only using the C version with inline assembler macros for some double precision operations. *This is the same kind of optimization used in* `MGFez`.
– Asm kernel = `gmp-4.0` linked with aggressively optimised assembler portions, exploiting MMX operations for a bigger speed-up. This is the stardard installation.
– Large prime = the prime 4086122041 was used, close to 2^{32} but *not* of special form. Incomplete reduction (Algorithm 4) is used.
– Small prime = the prime 637116481 was used (29 bit). In this case no incomplete reduction is required in the schoolbook socle.

Table 3. *Some Timings.*

Bits	`gmp-4.0`		`MGFez`	
(B)	inline asm	asm kernel	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

Even though such comparisons must be taken *cum granum salis*, it is worth noting that an implementation of our algorithms with just a few assembler macros shows the performance advantages of PAFFs over prime fields - even when the latter are aggressively optimized.

6 Conclusions

We have presented techniques for arithmetic in PAFFs. The first are two exponentiation algorithms which make heavy use of the structure induced by the automorphism of the group. We illustrated them in detail in the case of the Frobenius of the multiplicative group of a field extension. The other techniques reduce the cost of a field reduction involved in a multiplication in $\mathbb{F}_{p^m}$, i.e. in typical field operations.

We implemented these techniques in a relatively portable software library. The run times compared to the fastest general purpose long integer arithmetic library `gmp`, version 4.0, display a speed up ranging between 10 and 20 over prime fields of cryptographically relevant sizes. Our exponentiation algorithms outperform also the previously used methods for the particular instances of the extension fields which we consider.

Acknowledgements

The authors wish to express their gratitude to Alessandro Baldaccini and to the anonymous referees for their valuable comments and suggestions.

References

[1] L. M. Adleman and J. DeMarrais. *A subexponential algorithm for discrete logarithms over all finite fields.* Advances in Cryptology: CRYPTO'93, Douglas R. Stinson, ed., LNCS, **773**, pp. 147–158. Springer, 1994. 321

[2] R. Avanzi. *On multi-exponentiation in cryptography.* Preprint. 2002.
Available from `http://eprint.iacr.org`
Newer version: *On the complexity of certain multi-exponentiation techniques in cryptography.* Submitted. 322

[3] R. Avanzi and T. Lange. *Cryptographic Applications of Trace Zero Varieties.* Preprint. 321, 327

[4] D. V. Bailey and C. Paar. *Efficient Arithmetic in Finite Field Extensions with Applications in Elliptic Curve Cryptography.* Journal of Cryptography **14**, No. 3, 2001, pp. 153-176. 322

[5] P. Barrett. *Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor*, In: Advances in Cryptology–Proceedings of crypto'86, LNCS vol. 263, pp. 311–323. Springer, 1987. 328, 329

[6] R. Blahut. *Fast Algorithm for Digital Signal Processing.* Addison Wesley, (1987). 330

[7] A. Bosselaers, R. Govaerts and J. Vandewalle. *Comparison of three modular reduction functions.* In: *Advances in cryptology: CRYPTO '93*, D. R. Stinson (editor). LNCS vol. 773, pp. 175–186. Springer, 1994. 328

[8] J. Chung and A. Hasan. *More generalized Mersenne Numbers (Extended Abstract).* This volume. 328

[9] J. von zur Gathen and M.Nöcker. *Exponentiation in finite fields: theory and practice.* Proceedings of AAECC–12, LNCS, vol. 1255, pp. 88–133. Springer, 1997, 324

[10] R. P. Gallant, R. J. Lambert, S. A. Vanstone. *Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms* In *Advances in Cryptology – CRYPTO 2001 Proceedings*, pp. 190–200. Springer, 2001. 327

[11] T. Grandlund. GMP. A software library for arbitrary precision integers. Available from `http://www.swox.com/gmp/`

[12] A. Karatsuba and Yu. Ofman. *Multiplication of multidigit numbers on automata.* Soviet Physics-Doklady **7** (1963), pp. 595–596. 330

[13] N. Koblitz. *Elliptic curve cryptosystems*, Math. Comp. **48** (177), pp. 203–209, (1987). 320

[14] N. Koblitz. *Hyperelliptic cryptosystems*, J. of Cryptology **1**, pp. 139–150, (1989). 320

[15] N. Koblitz. *CM-Curves with good Cryptographic Properties.* In: Advances in Cryptology - CRYPTO '91, LNCS vol. 576, pp. 279–287. Springer, 1992. 327

[16] D. E. Knuth. *The art of computer programming. Vol. 2, Seminumerical algorithms*, third ed., *Addison-Wesley Series in Computer Science and Information Processing.* Addison-Wesley, 1997. 322, 323, 330

[17] T. Lange. *Trace Zero Subvariety for Cryptosystems.* Submitted. 321, 327

[18] A. K. Lenstra. *Using Cyclotomic Polynomials to Construct Efficient Discrete Logarithm Cryptosystems over Finite Fields.* In: Proceedings ACISP'97, LNCS vol. 1270, pp. 127-138. Springer, 1997. 322, 323

[19] S. Lim, S. Kim, I. Yie, J. Kim and H. Lee. *XTR Extended to $GF(p^{6m})$.* In: Proceedings of *SAC 2001.* LNCS 2259, pp. 301–312. Springer, 2001. 321

[20] A. Menezes, P. van Oorschot and S. Vanstone. *Handbook of Applied Cryptography*, CRC Press (1997). 322, 328
[21] P. L. Montgomery. *Modular multiplication without trial division*, Math. Comp. **44** (170), pp. 519–521 (1985). 328, 329, 330
[22] P. Mihăilescu. *Optimal Galois Field Bases which are not Normal.* Recent Results Session, Fast Software Encryption Symposium, Haifa (1997). 322
[23] V. S. Miller. *Use of elliptic curves in cryptography*, In: Advances in cryptology—crypto '85, LNCS vol. 218, pp. 417–426. Springer, 1986. 320
[24] V. Müller. *Efficient Point Multiplication for Elliptic Curves over Special Optimal Extension Fields.* In "Public-Key Cryptography and Computational Number Theory", September 11-15, 2000, Warschau, pp. 197–207. De Gruyter, 2001. 327
[25] A. Odlyzko. *Discrete Logarithms: The past and the future*, Designs, Codes and Cryptography **19** (2000), pp. 129–145. 321
[26] J.-J. Quisquater. *Procédé de Codage selon la Methode dite RSA, par un Microcontrôleur et Dispositifs Utilisant ce Procédé.* Demande de brevet Français. (Dépôt numéro: 90 02274), February 1990. 328
[27] J.-J. Quisquater. *Encoding System According to the So-called RSA Method, by Means of a Microcontroller and Arrangement Implementing this System.* U. S. Patent 5,166,978, November 1992. 328
[28] G. W. Reitwiesner. *Binary arithmetic.* Advances in Computers **1**, pp. 231–308 (1960). 322
[29] A. Schönhage and V. Strassen. *Schnelle Multiplikation großer Zahlen*, Computing **7**, pp. 281-292. 330
[30] O. Schirokauer. *Using number fields to compute logarithms in finite fields*, Math. Comp. **69** (2000), pp. 1267–1283. 321
[31] F. Sica, M. Ciet and J.-J. Quisquater. *Analysis of the Gallant-Lambert-Vanstone Method based on Efficient Endomorphisms: Elliptic and Hyperelliptic Curves.* In: *Selected Areas of Cryptography 2002*. LNCS 2595. pp. 21–36. Springer 2002. 327
[32] J. A. Solinas. *An improved algorithm for arithmetic on a family of elliptic curves.* In: B. S. Kaliski, Jr. (Ed.), *Advances in Cryptology – CRYPTO '97*. LNCS vol. 1294, pp. 357–371. Springer, 1997. 322, 327
[33] M. Stam and A. K. Lenstra, *Efficient subgroup exponentiation in quadratic and sixth degree extensions.* In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems – CHES 2002*. LNCS 2523, pp. 318–332. Springer, 2003. 321
[34] M. Stam. *Speeding up Subgroup Cryptosystems.* Ph.D. Thesis, Technical University of Eindhoven, 2003. ISBN 90-386-0692-3. 324
[35] E. R. Verheul and A. K. Lenstra. *The XTR public key system.* In *Advances in Cryptography – Crypto '00*, M. Bellare, ed., LNCS vol. 1880, pp. 1–19. Springer, 2000. 321
[36] Andrew C. Yao. *On the evaluation of powers.* SIAM Journal on Computing **5** (1976), pp. 100–103. 323