

# Effects of Optimizations for Software Implementations of Small Binary Field Arithmetic

Roberto Avanzi<sup>1</sup> and Nicolas Thériault<sup>2</sup>

<sup>1</sup> Fakultät für Mathematik, Ruhr-Universität Bochum and the Horst Görtz Institut für IT-Sicherheit, Germany.

Email: `roberto.avanzi@rub.de`

<sup>2</sup> Instituto de Matemática y Física, Universidad de Talca, Chile.

Email: `ntheriau@inst-mat.otalca.cl`

**Abstract.** We describe an implementation of binary field arithmetic written in the C programming language. Even though the implementation targets 32-bit CPUs, the results can be applied also to CPUs with different granularity.

We begin with separate routines for each operand size *in words* to minimize performance penalties that have a bigger relative impact for shorter operands – such as those used to implement modern curve based cryptography. We then proceed to use techniques specific to operand size *in bits* for several field sizes.

This results in an implementation of field arithmetic where the curve representing field multiplication performance closely resembles the theoretical quadratic bit-complexity that can be expected for small inputs. This has important practical consequences: For instance, it will allow us to compare the performance of the arithmetic on curves of different genera and defined over fields of different sizes without worrying about penalties introduced by field arithmetic and concentrating on the curve arithmetic itself. Moreover, the cost of field inversion is very low, making the use of affine coordinates in curve arithmetic more interesting. These applications will be mentioned.

**Keywords.** Binary fields, efficient implementation, curve-based cryptography.

## 1 Introduction

The performance of binary field arithmetic is crucial in several contexts. In many cases, the most important operation to optimize is the multiplication, which is based on the multiplication of polynomials over  $\mathbb{F}_2$ . A lot of work has been devoted to improve the speed of multiplication of binary polynomials of very large degree, for example to break polynomial factorization records. For other important applications such as the

implementation of elliptic curve (EC) and hyperelliptic curve (HEC) cryptography the focus is instead on relatively small fields, of roughly 40 to 300 bits. Only in special circumstances will larger fields be contemplated.

One of the most celebrated advantages of using higher genus curves in place of EC is the fact that to achieve the same level of security of the latter they need to be defined over smaller fields. For hardware designers this is a bonus since they can use smaller multiplication and inversion circuits, and thus higher genus curves hold their own advantage, such as a low area-time product [22].

However, most software implementations of finite fields penalize smaller fields more than larger ones: [1] shows that HEC in odd characteristic are often heavily penalized with respect to EC because of many types of overheads in the field arithmetic whose impact increases as field sizes get smaller. This is also the case in characteristic 2, but the nature of the worst overheads and the techniques used to address them differ.

*The techniques described in this paper are thus aimed at small fields such as those used in curve-based cryptography. They are also devised to improve the use of basic field operations in other practical contexts, such as polynomial factorization algorithms.*

The paper is structured as follows: Section 2 describes the types of overheads we must handle. We then look at the field multiplication (§ 3.1), sequential multiplications (§ 3.2), squaring (§ 3.3), modular inversion (§ 3.5), and modular reduction (§ 3.4). We conclude in Section 4 with performance results, including timings of EC and HEC arithmetic (that will be discussed in detail in a forthcoming paper) and their consequences.

## 2 Types of Overheads

We begin by introducing some notation. A field  $\mathbb{F}_{2^n}$  is represented using a polynomial basis as the quotient ring  $\mathbb{F}_2[t]/(p(t))$ , where  $p(t)$  is an irreducible polynomial of degree  $n$ . This is the most usual representation of small fields and their elements for software implementations (cf. [2, Ch. 11]). Field elements are represented by binary polynomials of degree less than  $n$ . Multiplication (resp. squaring) in  $\mathbb{F}_{2^n}$ , is performed by first multiplying (resp. squaring) the polynomial(s) representing the input(s), and then reducing the result modulo  $p(t)$ . Let  $\gamma$  be the number of bits in a computer word. A field element  $a$  occupies  $s = \lceil n/\gamma \rceil$  words  $A_0, A_1, \dots, A_{s-1}$ . The  $\gamma$  least significant bits of the polynomial representing  $a$  are contained in the words  $A_0$ , the next  $\gamma$  bits in  $A_1$  and so on.

In our implementation, we address the following types of overheads:

---

**Algorithm 1.** Field multiplication [13]

---

INPUT:  $A = (A_{s-1}, \dots, A_0)$ ,  $B = (B_{s-1}, \dots, B_0)$ , and a comb size  $w$

OUTPUT:  $R = (R_{2s-1}, \dots, R_0) = A \times B$

---

```
1. for  $j = 0$  to  $2^w - 1$  do
2.      $P_j(t) \leftarrow (j_{w-1}t^{w-1} + \dots + j_1t + j_0)A(t)$  where  $j = (j_{w-1} \dots j_2 j_1 j_0)_2$ .
           [Here the polynomial  $P_j(t)$  is at most  $s + 1$  words long]
3. for  $i = 0, \dots, 2s - 1$  do  $R_i \leftarrow 0$ 
4. for  $j = \lceil \gamma/w \rceil - 1$  downto  $0$  do
5.     for  $i = 0$  to  $s - 1$  do
6.          $u \leftarrow (B_i \gg jw) \bmod t^w$  [mask out  $w$  bits at time]
7.         for  $k = 0$  to  $s$  do
8.              $R_{k+i} \leftarrow R_{k+i} \oplus P_u[k]$ 
9.         if  $j \neq 0$  then  $R \leftarrow t^w R$  [bit shift]
10. return R
```

---

### 1. *Software loops to process operands, pipelining, and branch mispredictions.*

Algorithm 1 is the standard method for performing multiplication of two small polynomials over  $\text{GF}(2)$ . The issue of expensive branch mispredictions at loop boundaries is addressed by full loop unrolling for all input sizes. This is common in the long integer arithmetic underlying the implementation of prime fields [1]. Loop unrolling is also useful in even characteristic, but it is used differently: see § 3.1 for more details. In all cases, loop unrolling also helps the compiler to produce code which exploits the processor pipelines more efficiently.

### 2. *Architecture granularity.*

Algorithm 1 requires a precomputation phase to compute the multiples of the first input by all binary polynomials of degree less than  $w$  (Steps 1 and 2). The complexity of this stage is exponential in  $w$ , and the optimal value of  $w$  to minimize the total number of operations is an integer of the form  $O(\log \log n)$  (Theorem D, § 4.6.3 in [9] also applies to binary multiplication). The constants vary depending on the architecture however, and in most cases the optimal  $w$  can only be found by trial-and-error.

For large ranges of the input sizes, the value of  $w$  will remain constant. The complexity of multiplication in these ranges is therefore roughly quadratic in  $s$ , as a generic implementation does not distinguish between fields of  $s\gamma + 1$ -bits and  $(s + 1)\gamma$ -bits. This *granularity*

introduces irregular performance penalties. To see its effect in curve cryptography, let us consider an example.

For the same security level, we can use an EC over a field of 223 bits, or a genus 2 curve over a field of roughly 112 bits, using respectively 7 and 4 words per field element. The expected number of field multiplications for a group operation increases roughly quadratically with the genus<sup>3</sup>, but in this scenario the cost of a multiplication in the smaller field (which requires 4 words store each element) is about 1/3 of the cost of a multiplication in the larger field (where elements are 7 words long), as  $4^2 = 16$  is 32.6% of  $7^2 = 49$ . As a result, the granularity would penalize the HEC of genus two by a factor of 1.32.

Little can be done to defeat granularity problems in the prime field case [1], but more options are available in even characteristic (cf. § 3.1).

### 3. *Multiplications of a single field element by a vector of field elements.*

Once again, let us consider curve-based cryptography. In this context, occurrences of several multiplications by the same element become more common as the genus increases. It is possible to speed up these multiplications appreciably by treating them as vector multiplications rather than sets of independent multiplications. A similar situation occurs when multiplying polynomials of low degree, where vector multiplications can be faster than Karatsuba methods. The technique for doing this, described in § 3.2, is not new, but so far its impact has not been thoroughly evaluated in the literature. Similar optimization technique do not seem to be possible in the prime field case.

Another important difference with the prime field case lies in the relative cost of modular reduction compared to multiprecision multiplication.

For prime fields, this relative cost increases as the operand size decreases, and in odd characteristic HEC implementations more time is spent doing modular reductions than in EC implementations. To decrease the total cost of reductions, one can delay modular reductions when computing the sum of several products. The idea is described in [17], and [1] shows that this approach also works with the Montgomery representation of modular integers.

---

<sup>3</sup> This is an average assuming a windowed scalar multiplication method, where the most common group operation is doublings. By choosing curves of a special type, group doublings can be implemented with quadratically many field multiplications in the genus, even though group additions have cubic complexity. This approximation works nicely for curve of genus up to 4 (and possibly more), see for example [11, 14, 15, 5, 3, 23] and [2, Ch. 13, 14].

In even characteristic, modular reduction is much cheaper, and the additional memory traffic caused by delaying reductions can even reduce performance. After doing some operation counts and practical testing, we opted not to allow delayed reductions.

This in turn raises the issue of the number of function calls, since calling the reduction code after each multiplication increases this number.

#### 4. *Function call overheads.*

Function call overheads also play a different role in even characteristic. In fact, the code for implementing arithmetic routines is bigger than in the prime field case. Inlining it would usually cause the code size to increase dramatically, creating a large penalty in performance. As a result, field additions are inlined in the main code of our programs (such as explicit formulæ for curve arithmetic) but multiplications are not. However, we inline the reduction code into the special multiplication and squaring routines for each field: the reduction code is compact and it is inlined only a few times, thus keeping code size manageable while saving function calls.

### 3 Implementation Techniques

#### 3.1 Field Multiplication and Architecture Granularity

The starting point for our implementation of field multiplication is Algorithm 1, by López and Dahab [13]. It is based on comb exponentiation, that is usually attributed to Lim and Lee [12], but that in fact is due to Pippenger [16]. Recall that  $s$  is the length of the inputs in computer words and  $\gamma$  the number of bits per word. There are a few obvious optimizations of Algorithm 1.

If the inputs are at most  $s\gamma - w + 1$  bits long, all the  $P_j(t)$ 's fit in  $s$  words. In this case, the counter  $k$  of the loop at Step 7 will only need to go from 0 to  $s - 1$ .

If operands are between  $s\gamma - w + 2$  and  $s\gamma$  bits long, proceed as follows; Zero the  $w - 1$  most significant bits of  $A$  for the computations in Steps 2 and 3, thus obtaining polynomials  $P_j(t)$  that fit in  $s$  words; Perform the computation as in the case of shorter operands, with  $s - 1$  as the upper bound in Step 7; Finally, add the multiples of  $B(t)$  corresponding to the  $w - 1$  most significant bits of  $A$  to  $R$  before returning the result. This leads to a much faster implementation since several memory writes in Step 2 and memory reads in Step 8 are traded for a minimal amount of memory operations later to “adjust” the final result.

More optimizations can be applied if the field size is known in advance, such as partial or full loop unrolling. Also, some operands (containing the most significant bits of  $R$ ) are known to be zero in the first repeats of the loop bodies, hence parts of the computation can be explicitly omitted during the first iterations. This can be done in the full unrolling or just by splitting a loop in two or more parts whereby one part contain less instructions than the following one.

Steps 7 and 8 could be skipped when  $u = 0$  but inserting an “If  $u \neq 0$ ” statement before Step 7. *de facto* slows down the algorithm (because of frequent branch mispredictions), so it is more efficient to allow these “multiplications by zero”.

We considered different comb sizes and different loop splitting and unrolling techniques for several different input sizes, ranging from 43 to 283 bits – the exact choice being determined by the application described in § 4.2.)

As we mentioned in the previous section, the optimal choice for the comb size  $w$  will depend both on the field size and the architecture of the processor. *It should be noted that the choice of the comb size and the point at which Karatsuba multiplication becomes interesting (see below) are the only optimizations that are processor-dependent (given a fixed word size), whereas the other techniques presented in this paper are not affected by the processor’s architecture.*

Special treatment is reserved to polynomials whose size is just a few bits more than a multiple  $k$  of the architecture granularity. Two such cases are polynomials of 67 and 97 bits, where the values of  $k$  are 64 and 96, respectively. We perform scalar multiplication of the polynomials consisting of the lower  $k$  bits first, and then handle the remaining most significant bits one by one. In other words, write  $A = A' + t^k \cdot A''$  and  $B = B' + t^k \cdot B''$  with the degrees of  $A', B'$  smaller than  $k$ , and perform  $A \cdot B = A' \cdot B' + t^k \cdot A'' \cdot B' + t^k \cdot A' \cdot B'' + t^{2k} \cdot A'' \cdot B''$ . In some cases, a little regrouping such as  $A \cdot B = A' \cdot B' + t^k \cdot A'' \cdot B' + t^k \cdot B'' \cdot A$  is slightly more efficient. The resulting code is 10 to 15% faster than if we applied the previous approaches to the one-word-longer multiplication.

Multiplication of polynomials whose degree is high enough is also done using Karatsuba’s technique [7] of reducing the multiplication of two polynomials to three multiplications of half size polynomials. After some testing, we observed that Karatsuba multiplication performs slightly better than comb-based multiplication for  $s \geq 6$  on the PowerPC, and for  $s \geq 7$  on the Intel Core architecture (the half-size multiplications are performed with the comb method), but not for smaller sizes.

The performance of multiplication routines can be seen in Tables 1 and 2 and Figures 1 and 2. They will be discussed in more details in Section 4, but we can already observe that the cost of multiplication grows quite smoothly with the field size, and in fact it approached a curve of quadratic bit complexity (as we might expect from theory) much better than a coarser approach that works at the word level.

### 3.2 Sequential Multiplications

In certain situations, for instance in explicit formulæ for curves of genus two to four, we find sets of multiplications with a common multiplicand. This usually occurs as a result of polynomial arithmetic.

A natural approach to reduce average multiplication times is to keep the precomputations (Steps 1 and 2 of Algorithm 1) associated to the common multiplicand and re-use them in the next multiplications. However, this would require extra variables in the implementation of the explicit formulæ and demand additional memory bookkeeping. We thus opted for a slightly different approach: we wrote routines that perform the precomputations once and then repeat Steps 3 to 10 of Algorithm 1 for each multiplication. We call this type of operation *sequential multiplication* (the more common terminology of scalar multiplication having another signification in curve based cryptography...).

An important observation is that the optimal comb size used in the multiplication (on a given architecture) may vary depending on the number of multiplications that are performed together. For example, for the field of 73 bits on the PowerPC, a comb of size 3 is optimal for the single multiplication and for sequences of two multiplications, but for 3 to 5 multiplications the optimal comb of size is 4. For the field of 89 bits, the optimal comb size for single multiplications is again 3, but it already increases to 4 for the double multiplication.

If a comb method is used for 6-word fields on the PowerPC and Core architectures, then 4 is the optimal comb size for single multiplications and 5 is optimal for groups of at least 3 multiplications. However, on the PowerPC, Karatsuba is used not only for the single multiplications, but also for the sequential ones, where a sequential multiplication of  $s$ -word operands is turned into three sequential multiplications of  $s/2$ -word operands for  $s \geq 6$ . On the Core CPU, Karatsuba's approach becomes more efficient for sequential multiplications only when  $s \geq 8$ .

To keep function call overheads low, the sequential multiplication procedures for at least 3 multiplications use input and output *vectors* of

elements which are adjacent in memory. This also speeds up memory accesses, taking better advantage of the structure of modern caches.

Static precomputations have already been used for multiplications by a constant parameter (coming from the curve or the field) – for example, [4] suggests this in the context of square root extraction. In [8], King uses a similar approach to reduce the number of precomputed tables in the projective formulæ for elliptic curves, however he does not estimate the costs of several multiplications performed by this method in comparison to the cost of one multiplication, nor does he adapt the comb size to the number of multiplications performed.

### 3.3 Polynomial Squaring

Squaring is a linear operation in even characteristic: if  $p(t) = \sum_{i=0}^n e_i t^i$  where  $e_i \in \mathbb{F}_2$ , then  $(p(t))^2 = \sum_{i=0}^{n-1} e_i t^{2i}$ . That is, the result is obtained by inserting a zero bit between every two adjacent bits of the input. To efficiently implement this process, a 512-byte table is precomputed for converting 8-bits polynomials into their expanded 16-bits counterparts [18]. In practice, this technique is faster than King’s method [8] (which otherwise has the advantage of requiring less memory).

### 3.4 Modular Reduction

We implemented two sets of routines for modular reduction.

The first is a generic routine that reduces arbitrary polynomials over  $\mathbb{F}_2$  modulo another arbitrary polynomials over  $\mathbb{F}_2$ . This code is in fact rather efficient, and a reduction by means of this routine can often take less than 20% of the time of a multiplication. The approach is similar to the one taken in SUN’s ECC contributions to OpenSSL [20] or in NTL [19]. Let the reduction polynomial be  $\sum_{i=0}^{k-1} t^{n_i}$ . Its degree is  $n_0$  and its *sediment* is  $\sum_{i=1}^{k-1} t^{n_i}$ . If an irreducible trinomial ( $k = 3$ ) exists, we use it, otherwise we use a pentanomial ( $k = 5$ ).

The second set of routines uses fixed reduction polynomials, and is therefore specific for each polynomial degree. The code is very compact. Here we sometimes prefer reduction eptanomials ( $k = 7$ ) to pentanomials when the reduction is faster due to the form of the polynomial, for instance when the sediment has lower degree and/or it factors nicely.

As an example, for degree 59 we have two good irreducible polynomials:  $f_1(t) = t^{59} + (t+1)(t^5 + t^3 + 1)$  and  $f_2(t) = t^{59} + t^7 + t^4 + t^2 + 1$ . The first C code fragment takes a polynomial of degree up to 116 ( $= 2 \cdot 58$ )

stored in variables **r3** (most significant word), **r2**, **r1** and **r0** (least significant word), and reduces it modulo  $f_1(t)$ , leaving the result in **r1** and **r0**:

```
#define bf_mod_59_6_5_4_3_1_0(r3,r2,r1,r0) do {           \
    r3 = ((r3) << 5) ^ ((r3) << 6);                    \
    r1 ^= (r3) ^ ((r3) << 3) ^ ((r3) << 5);            \
    r3 = ((r2) << 5) ^ ((r2) << 6);                    \
    r0 ^= (r3) ^ ((r3) << 3) ^ ((r3) << 5);            \
    r3 = ((r2) >> 22) ^ ((r2) >> 21);                 \
    r1 ^= (r3) ^ ((r3) >> 2) ^ ((r3) >> 5);           \
    r2 = (r1) >> 27; r2 ^= (r2) << 1;                  \
    r1 ^= 0x07ffffff; r0 ^= (r2) ^ ((r2) << 3) ^ ((r2) << 5); \
} while (0)
```

The C code to reduce the same input modulo  $f_2(t)$  is

```
#define bf_mod_59_7_4_2_0(r3,r2,r1,r0) do {           \
    r1 ^= ((r3) << 5) ^ ((r3) << 7) ^ ((r3) << 9) ^ ((r3) << 12); \
    r2 ^= ((r3) >> 25) ^ ((r3) >> 23) ^ ((r3) >> 20); \
    r0 ^= ((r2) << 5) ^ ((r2) << 7) ^ ((r2) << 9) ^ ((r2) << 12); \
    r1 ^= ((r2) >> 27) ^ ((r2) >> 25) ^ ((r2) >> 23) ^ ((r2) >> 20); \
    r2 = (r1) >> 27; r1 ^= 0x07ffffff;                \
    r0 ^= (r2) ^ ((r2) << 2) ^ ((r2) << 4) ^ ((r2) << 7); \
} while (0)
```

We found the first reduction routine slightly more efficient. A similar choice occurs at degree 107 (the eptanomial being  $t^{107} + (t^6 + t^2 + 1)(t + 1)$ ), and the idea of factoring the “lower degree part” of the reduction polynomial is also used for degree 109 (the polynomial is  $t^{109} + (t^6 + 1)(t + 1)$ ).

These considerations were applied to the degrees 43, 47, 53, 59, 67, 73, 79, 83, 89, 97, 101, 107, 109, 113, 127, 131, 137, 149, 157, 163, 179, 199, 211, 233, 239, 251, 269 and 283. For degrees 47, 79, 89, 97, 113, 127, 137, 199, 233 and 239 we used a trinomial. For degrees 59 and 107 we opted for eptanomials (cf. remarks above), and in all other cases pentanomials were used. The time for the modular reduction is kept between 3 and 5% of the time required for a multiplication if we use a trinomial, and between 6 and 10% in the other cases. Reduction modulo a trinomial is about twice fast as polynomial squaring because the latter requires more memory accesses.

### 3.5 Modular Inversion

There are many algorithms for computing the inverse of a polynomial  $a(t)$  modulo another polynomial  $p(t)$ , where both polynomials are defined over the field  $\mathbb{F}_2$ . In [6] three methods are compared:

- The Extended Euclidean Algorithm (EEA), where partial quotients are approximated by powers of  $t$ , and no polynomial division is required.
- The Almost Inverse Algorithm (AIA), a variant of the binary extended GCD algorithm that computes a polynomial  $b(t)$  together with an integer  $k$  such that  $b(t)a(t) \equiv t^k \pmod{p(t)}$ . The final result must then be adjusted.
- The Modified Almost Inverse Algorithm (MAIA), which is a variant of the binary extended GCD algorithm which returns the correct inverse as a result.

We refer to [6] for details. In agreement with [6], we find that EEA performs consistently better than the two other methods in our context.

For inputs of up to 8 words, we always keep all words (limbs) of all multiprecision operands in separate integer variables explicitly, not in indexed arrays. This allows the compiler to allocate a register for each of these integer variables if enough registers are provided by the architecture (such as on RISC processors). Furthermore, it does not penalize architectures with fewer registers: the contents of many variables containing individual words are spilled on the stack, but this data would still be stored in memory if we used arrays.

Another advantage of the EEA is that it offers good control on the bit lengths of the intermediate variables. We can therefore split the main loop in several copies optimised for the different sizes of the intermediate operands, with  $n$  sections of code for inputs of  $n$  words. Since some intermediate values grow in size as other values get shorter, we can reduce the local usage of registers, allowing an increase in the size of the inputs before the compiler starts to produce code that spills some data to memory. See Section 4 for inversion performance.

## 4 Performance Results, Comparisons, and Conclusions

We compiled and ran our code on several architectures. Due to space constraints we present here two sets of timings taken on very different CPUs. The first set was obtained on a 1.5 Ghz PowerPC G4 (Motorola 7447) CPU, a 32-bits RISC architecture with 32 general purpose integer registers, no level 3 cache support and slow memory bus. The second set was taken on a 1.83 Ghz Intel Core 2 Duo (running 32-bit code on one core), an architecture with fewer registers, a better cache system and faster memory bus.

In our C code, we declare all limbs of all intermediate operands as single 32-bit word variables, and operate on them with logic and shift operations. The code compiles on any 32-bits architecture supported by the gnu compiler collection (we used gcc 4.0.1, Apple branch, build 5367, under Mac OS X 10.4.9 on both architectures). We did not use assembler code – which would be necessary to get satisfactory performance in long integer arithmetic – since near-optimal performance can be attained for binary field arithmetic by carefully crafting the C code. This is a known fact: the binary arithmetic of NTL [19], which is very well known for its performance, is written in C (in fact C++), as are SUN’s ECC contributions to OpenSSL [20].

Tables 1 and 2 contain the timings of the fundamental operations in the fields that we considered. The operations are: single multiplication (Mul), squaring (Sqr), multiplication of 2 to 5 different field elements by a fixed one (columns from Mul2 to Mul5) and inversion (Inv). Modular reduction is always included. We give the absolute times in microseconds and the relative costs compared to a single multiplication. We also give the timings of our best generic routines for field multiplications (more or less on-par with the NTL libraries) together with the speedup factor gained by the field-specific routines.

Figures 1 and 2 represent the results of the single multiplications. The timings of our per-field ad-hoc routines are compared to those of our generic implementation. The parabola arcs represent interpolations of the form  $c_1 \cdot n^2 + c_0$  bit operations using our best performance values.

The way we implemented inversion (§ 3.5) allows us to get extra performance from CPUs with several registers. For several input sizes, most of the computation can take place only between registers on these processors, without accessing external memory except to load the inputs and to store the final result – thus, a slow memory bus or a high level one cache latency are not big problems. This is reflected in the exceptionally low I/M ratio on the PowerPC: for inputs of up to 6 words all the operands of the EEA fit in the registers. For longer inputs the registers no longer suffice, the compiler must store some partial data in the main memory (as confirmed by disassembly of the compiled code), and a “bump” in inversion performance occurs.

The multiplication routines make extensive use of tables of precomputations, hence they are more penalized by slow memory bus or high level one cache latency than by register paucity. This is reflected in the fact the Intel Core 2 Duo offers better multiplication performance than the PowerPC, especially for larger fields, that use larger tables. Some of the

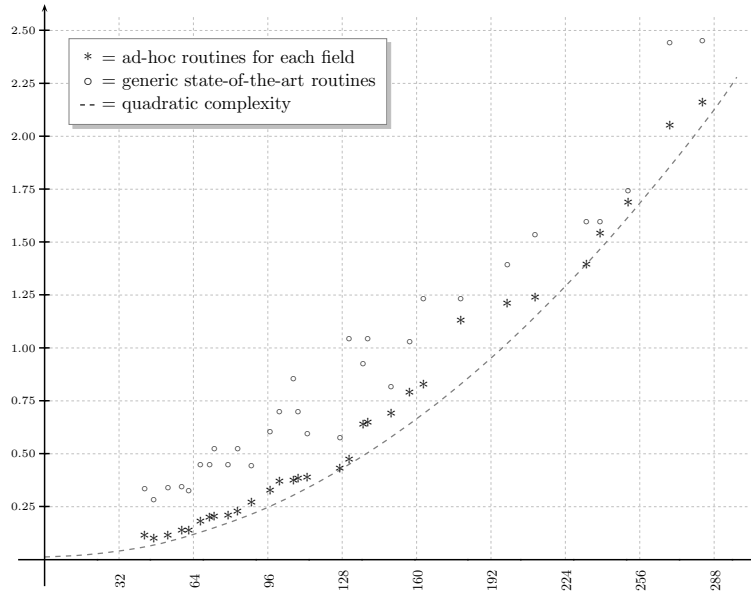
**Table 1.** Timings of field operations in  $\mu\text{sec}$  and ratios (1.5 GHz Powerpc G4)

Field Size	Timings of optimized library														Standard Mult.	
	Absolute timings							Timings relative to multiplication							Time	Speed-up
	Mul	Sqr	Mul2	Mul3	Mul4	Mul5	Inv	Sqr	Mul2	Mul3	Mul4	Mul5	Inv			
43	.100	.017	.169	.227	.287	.344	.450	.169	1.693	2.271	2.874	3.451	4.505	.331	3.310	
47	.087	.014	.142	.192	.243	.296	.483	.164	1.640	2.225	2.795	3.421	5.511	.338	3.885	
53	.098	.018	.167	.218	.276	.337	.512	.183	1.707	2.224	2.817	3.434	5.224	.280	2.857	
59	.121	.021	.210	.272	.333	.394	.531	.172	1.740	2.257	2.762	3.267	4.402	.336	2.776	
67	.168	.025	.276	.348	.436	.516	.820	.148	1.647	2.073	2.598	3.077	4.886	.444	2.627	
73	.190	.024	.329	.466	.572	.674	.857	.128	1.732	2.453	3.011	3.547	4.508	.521	2.742	
79	.193	.019	.321	.452	.549	.649	.899	.099	1.662	2.342	2.848	3.364	4.658	.448	2.321	
83	.213	.050	.387	.518	.640	.760	.922	.236	1.818	2.430	3.003	3.569	4.329	.521	2.446	
89	.254	.025	.420	.538	.667	.796	.962	.100	1.653	2.119	2.627	3.136	3.790	.443	1.744	
97	.311	.025	.455	.612	.761	.913	1.589	.081	1.462	1.967	2.445	2.933	5.105	.602	1.936	
101	.353	.057	.552	.722	.903	1.083	1.621	.161	1.564	2.046	2.557	3.068	4.592	.695	1.969	
107	.358	.058	.559	.760	.954	1.149	1.659	.162	1.560	2.122	2.665	3.209	4.633	.852	2.380	
109	.371	.029	.567	.799	.998	1.201	1.692	.079	1.528	2.155	2.693	3.241	4.564	.695	1.873	
113	.373	.029	.580	.783	.981	1.185	1.702	.078	1.554	2.099	2.629	3.177	4.561	.593	1.590	
127	.415	.053	.674	.957	1.201	1.447	1.832	.128	1.625	2.306	2.894	3.486	4.415	.574	1.383	
131	.460	.067	.763	1.046	1.329	1.605	3.664	.147	1.659	2.275	2.890	3.489	7.968	1.042	2.265	
137	.625	.062	1.084	1.485	1.907	2.325	3.733	.100	1.734	2.375	3.050	3.718	5.969	.926	1.539	
149	.677	.090	1.191	1.680	2.170	2.656	3.908	.133	1.760	2.482	3.206	3.924	5.773	.817	1.207	
157	.774	.090	1.469	1.951	2.441	2.910	4.055	.116	1.899	2.522	3.155	3.762	5.243	1.027	1.327	
163	.815	.085	1.342	1.902	2.455	3.009	5.336	.105	1.647	2.334	3.012	3.692	6.547	1.229	1.508	
179	1.116	.124	2.117	2.916	3.623	4.508	5.552	.111	1.896	2.613	3.246	4.038	4.974	1.229	1.101	
199	1.195	.091	2.085	2.716	3.423	4.126	12.114	.076	1.745	2.273	2.865	3.454	10.140	1.390	1.163	
211	1.225	.145	2.169	2.937	3.695	4.435	12.559	.119	1.771	2.398	3.016	3.621	10.254	1.531	1.250	
233	1.380	.114	2.347	3.206	4.013	4.883	14.613	.079	1.701	2.323	2.907	3.537	10.596	1.594	1.155	
239	1.528	.187	2.630	3.637	4.604	5.638	14.828	.122	1.722	2.381	3.014	3.691	9.706	1.596	1.060	
251	1.675	.228	2.978	4.157	5.297	6.498	15.157	.137	1.778	2.482	3.163	3.879	9.050	1.741	1.040	
269	2.035	.210	3.790	5.146	6.431	7.853	20.495	.103	1.863	2.529	3.161	3.860	10.073	2.438	1.197	
283	2.148	.229	3.943	5.338	6.817	8.134	20.845	.107	1.835	2.485	3.174	3.787	9.704	2.447	1.139	

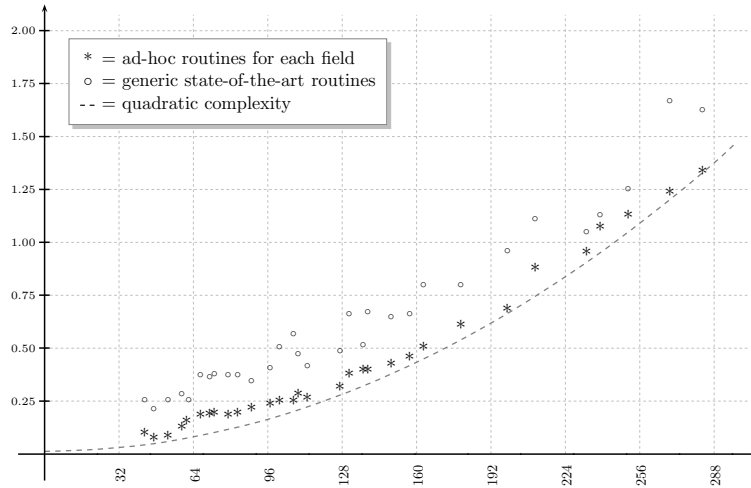
**Table 2.** Timings of field operations in  $\mu\text{sec}$  and ratios (1.83 GHz Intel Core 2 Duo)

Field Size	Timings of optimized library														Standard Mult.	
	Absolute timings							Timings relative to multiplication							Time	Speed-up
	Mul	Sqr	Mul2	Mul3	Mul4	Mul5	Inv	Sqr	Mul2	Mul3	Mul4	Mul5	Inv			
43	.087	.017	.137	.182	.228	.260	1.381	.195	1.575	2.091	2.621	2.989	15.830	.253	2.908	
47	.064	.012	.108	.153	.194	.239	1.502	.191	1.688	2.393	3.028	3.742	23.484	.211	3.297	
53	.072	.025	.131	.185	.240	.294	1.623	.350	1.820	2.573	3.350	4.107	22.635	.253	3.514	
59	.117	.027	.201	.279	.361	.444	1.740	.228	1.723	2.396	3.097	3.813	14.935	.285	2.436	
67	.174	.027	.227	.313	.410	.502	2.386	.154	1.302	1.797	2.354	2.881	13.693	.371	2.132	
73	.174	.026	.321	.473	.545	.667	2.512	.151	1.840	2.716	3.128	3.827	14.413	.377	2.167	
79	.171	.025	.257	.332	.425	.511	2.649	.146	1.506	1.947	2.487	2.994	15.520	.374	2.187	
83	.184	.041	.285	.350	.458	.554	2.727	.224	1.553	1.908	2.492	3.018	14.857	.371	2.016	
89	.207	.028	.318	.426	.526	.619	2.813	.135	1.536	2.106	2.589	3.039	13.589	.345	1.665	
97	.223	.032	.379	.531	.687	.838	3.677	.142	1.701	2.388	3.085	3.765	16.523	.405	1.816	
101	.240	.039	.431	.594	.767	.945	3.760	.164	1.796	2.472	3.194	3.934	15.659	.507	2.113	
107	.240	.038	.427	.620	.813	1.001	3.895	.157	1.779	2.582	3.385	4.171	16.223	.566	2.358	
109	.274	.035	.505	.702	.918	1.131	3.961	.128	1.840	2.559	3.345	4.121	14.433	.473	2.111	
113	.255	.047	.444	.637	.828	1.022	4.008	.183	1.741	2.496	3.245	4.005	15.708	.414	1.624	
127	.305	.038	.563	.836	1.084	1.335	4.375	.124	1.848	2.745	3.560	4.385	14.365	.488	1.467	
131	.366	.064	.679	.961	1.269	1.581	5.278	.174	1.855	2.624	3.466	4.316	14.413	.660	1.803	
137	.384	.049	.706	.933	1.219	1.485	5.396	.129	1.837	2.426	3.172	3.863	14.039	.515	1.341	
149	.413	.072	.740	1.081	1.420	1.760	5.676	.174	1.791	2.617	3.437	4.261	13.741	.647	1.567	
157	.447	.072	.848	1.170	1.508	1.844	5.887	.161	1.900	2.621	3.378	4.129	13.185	.660	1.477	
163	.495	.069	.942	1.358	1.787	2.221	6.855	.139	1.904	2.744	3.611	4.488	13.855	.798	1.612	
179	.598	.085	1.101	1.513	1.951	2.390	7.301	.143	1.843	2.533	3.266	4.000	12.218	.798	1.334	
199	.674	.067	1.211	1.975	2.128	2.604	9.542	.099	1.797	2.930	3.157	3.864	14.157	.957	1.420	
211	.868	.089	1.580	2.256	2.926	3.572	10.591	.102	1.821	2.600	3.372	4.117	12.206	1.112	1.281	
233	.945	.086	1.759	2.532	3.256	4.005	12.246	.091	1.860	2.678	3.444	4.236	12.954	1.047	1.080	
239	1.061	.187	2.014	2.880	3.748	4.584	12.378	.176	1.898	2.715	3.533	4.321	11.668	1.131	1.066	
251	1.120	.206	2.075	3.008	3.884	4.809	12.641	.184	1.853	2.686	3.468	4.294	11.287	1.260	1.116	
269	1.224	.127	2.209	3.128	4.050	4.939	14.924	.104	1.805	2.556	3.309	4.036	12.195	1.667	1.362	
283	1.325	.141	2.282	3.206	4.135	5.070	15.284	.106	1.722	2.420	3.120	3.826	11.535	1.625	1.316	

**Fig. 1.** Field multiplication performance on the 1.5 GHz PowerPC G4 ( $\mu\text{sec}$ ).



**Fig. 2.** Field multiplication performance on the 1.83 GHz Intel Core 2 Duo ( $\mu\text{sec}$ ).



deficiencies of older CISC architectures, such as the small register set, are now mitigated by register renaming and wide-execution units, but their impact is still noticeable. This can be seen, for example, by the higher I/M ratio on the Intel Core 2 Duo.

The choice of trinomials, pentanomials or eptanomials is reflected in the timings. The ratio Sqr/Mul is higher when pentanomials or eptanomials are used because the reduction is significantly slower than it would have been if an irreducible trinomial of the same degree had existed. The reduction has a smaller impact on field multiplication than on field squaring. In the case of generic routines, the variations due to the choice of polynomials are bigger, and it is easy to recognize that the zigzagging multiplication performance curve follows the shape of a staircase over which pentanomial-induced “wedges” are placed. Apart from this “zigzagging”, it is difficult to find more patterns in the performance gap between specialized routines and generic ones. It seems to remain (on average) similar for small and larger fields, with maybe only a small linear component in the bit size. This suggests that most of the costs we eliminated were roughly constant, and therefore had much bigger impact on the performance of smaller fields

#### 4.1 Comparisons with Other Literature

It can be difficult to compare our results with the literature, since in most cases only a few fields were implemented and benchmarked, and usually only the larger ones (for EC) or the smaller ones (for HEC), but not both.

Our goal was to get performance curves that would resemble the parabola arcs expected from theory instead of the usual “broken staircases” (which can only be shown with the implementation of several fields), and at the same time matching or improving on the best results for individual fields that are scattered in the literature. We wanted to show that in even characteristic the granularity of the architecture does not play the same crucial role as in the prime field case. when, say comparing performances of different types of curves over fields of various sizes. For this reason we did not use vector extensions like MMX, SSE, or AltiVec: these only “change the granularity”, but results would have been similar.

In [15], two sets of implementations are reported for binary fields of 32, 40 and 47 bits, one on an ARM 7 and the other on a 1.8 Ghz Pentium 4. For the canonical trio of fields of 163, 233 and 283 bits, a few more papers [21, 6, 4] are available. These timings can be seen in Table 3. The best timings in [13] essentially agree with those in [6] for the Pentium II and are slightly slower than those [4] for the Sparc.

In [8], the timing for the multiplications must be adjusted by a factor of 10 so they correspond to the overall timings of the elliptic curve operations. This is most likely due to a simple error in notation. The results as they are stated, suitably scaled, mean that King’s code would perform a multiplication about 8 times faster than we and [6] do, but a whole scalar multiplication 20% slower.

Scaling the 47-bit field performance on the 80 Mhz ARM 7, we see that our routines are about 6 times more efficient, even though in this case a factor at least 2 is due to the processor architecture. The 1.8 Pentium 4 performance from [15] would scale to about 0.1008  $\mu$ sec at 3.0 Ghz – our code performs a 47-bit finite field multiplication in 0.079  $\mu$ sec on a 3.0 Ghz Pentium 4. Comparable scaled performance of our code with those from [6, 4], two reference papers, can be noted (the timings of their gcc-compiled non-mmx versions are given).

**Table 3.** *Timings in  $\mu$ secs of field operations in other papers*

Field Size	[15]: ARM 7, 80 MHz			[15]: Intel P4, 1.8 GHz		
	Mult.	Inv.	Ratio	Mult.	Inv.	Ratio
32	2.6	26.8	10.16	0.168	1.650	9.82
40	7.3	49.2	6.73	0.413	2.519	6.04
47	7.3	77.5	10.5	0.402	3.752	9.33

Field Size	[6]: Intel Pentium II 400 Mhz, gcc			[4]: Intel Pentium 3 800 Mhz, gcc			[4]: Sparc 500 Mhz, gcc			[21]: 900 Mhz Sparc	[21]: 1 Ghz Intel
	Mult.	Inv.	Ratio	Mult.	Inv.	Ratio	Mult.	Inv.	Ratio	Mult.	Mult.
163	3.00	30.99	10.33	1.8	12.0	6.67	1.9	16.8	8.85	2.9	2.3
233	5.07	53.22	10.49	3.0	21.9	7.3	4.0	36.8	9.2	3.1	3.2
283	6.23	70.32	11.29							5.2	4.9

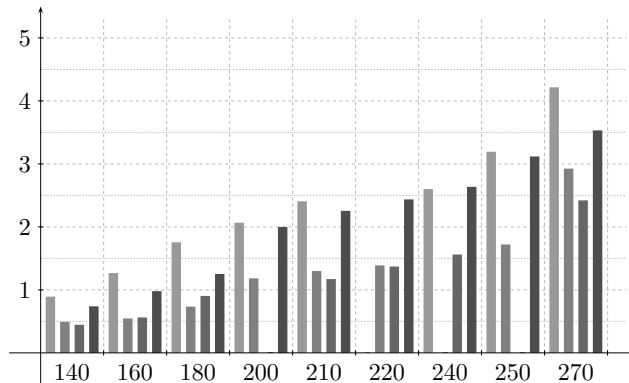
## 4.2 Application to Curve-Based Cryptography

An implementation of EC and of HEC of genus up to four has been written. The aim was to compare the performance of curves of different genera offering the same security level. Therefore, under consideration of the best attacks for each type of curve (see [2, Chs. 19, 20, 21 and 23]), we tried to find suitable quadruplets of fields to use as fields of definition for EC and HEC of genera 2, 3 and 4 at each of the chosen security levels. Due to the irregularity of the distribution of the primes, we had to choose security levels (expressed in bit-equivalents for EC) that permitted us to find matches for at least three curves, with tolerances of at most 2% (in bits) of security level between the curves in each level. The details of

this implementation, which include several new explicit formulæ, will be presented elsewhere.

Figure 3 shows the timings for each curve type and security level on the 1.5 Ghz PowerPC G4. The scalar multiplication algorithm used is a windowed method. Precomputation times are included in the timings. For EC, the best choice of coordinate system proved to be mixed affine/López-Dahab coordinates. For higher genus HEC, only affine coordinates are used.

**Fig. 3.** Scalar multiplication timings ( $\mu\text{sec}$ ) for curves of different genera at various security levels



Elliptic curves, genus 2, genus 3, genus 4.

*The performance of curves of genus four is comparable to that of EC, sometimes better. Curves of genus two and three have similar performance, with genus three winning in some cases despite the use of the Lange-Stevens genus two doubling [11], and both perform better than curves of genus one and four.*

We mention the fact that for curves of genus 3 and 4 the gain obtained by using sequential multiplications is significant, often around 15% and 20% respectively. On the other hand, there are no gains for EC in affine coordinates and a very small gain using López-Dahab coordinates (less than 1%.) For curves of genus two, the improvement is around 2%. In other words, without sequential multiplications the relative performance of genus 3 and 4 curves would have been much worse.

### 4.3 Conclusions and Perspectives

Our implementation “smooths” the performance of binary fields as a function of the extension degree: jumps corresponding to crossing granularity boundaries and erratic behavior depending on modular reduction mostly disappear. The shape of the graph of multiplication timings is quite close to the parabola arc given by the theoretical quadratic bit-complexity. For small fields, the performance gain due to our optimizations can be close to a factor 4 for multiplication and even around 10 for inversion, with respect to state-of-the-art generic libraries.

Sequences of several multiplications with a common multiplicand can be implemented faster by reusing the precomputations. We assess the resulting gains. These vary wildly but we can still see that 2, 3, 4 and 5 multiplications with a common multiplicand can be performed at roughly the cost of 1.75, 2.35, 3 and 3.65 single multiplications respectively.

This prompts the development of new explicit formulae for arithmetic on elliptic and hyperelliptic curves that take into account these routines and ratios. In fact, we can already report on an implementation of curve-based cryptographic primitives that depends in a significant way on our optimized routines.

**Acknowledgements:** *Parts of this work were done while the second author was at the Department of Combinatorics and Optimization, University of Waterloo, Canada, and at the Fields Institute, Toronto, Canada, and while the first author was visiting the same two institutions.*

### References

1. R. Avanzi. *Aspects of hyperelliptic curves over large prime fields in software implementations*. CHES 2004, LNCS 3156, 148–162, Springer, 2004.
2. R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *The Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2005.
3. X. Fan, T. Wollinger and Y. Wang. *Efficient Doubling on Genus 3 Curves over Binary Fields*. IACR ePrint 2005/228.
4. K. Fong, D. Hankerson, J. López, A. Menezes. *Field Inversion and Point Halving Revisited*. IEEE Trans. Computers 53(8), 1047–1059, 2004.
5. C. Guyot, K. Kaveh and V. M. Patankar. *Explicit algorithm for the arithmetic on the hyperelliptic Jacobians of genus 3*. J. Ramanujan Math. Soc., **19**(2), 75–115, 2004.
6. D. Hankerson, J. López-Hernandez, and A. Menezes. *Software Implementation of Elliptic Curve Cryptography over Binary Fields*. CHES 2000. LNCS 1965, 1–24, Springer, 2001.
7. A. Karatsuba and Y. Ofman. *Multiplication of Multidigit Numbers on Automata*. *Soviet Physics - Doklady* **7**, 595–596, 1963.

8. B. King. *An Improved Implementation of Elliptic Curves over  $GF(2^n)$  when Using Projective Point Arithmetic*. SAC 2001, LNCS 2259, 134–150. Springer, 2001.
9. D. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*. Third Edition. Addison Wesley Longman, 1998
10. T. Lange. *Efficient Arithmetic on Genus 2 Hyperelliptic Curves over Finite Fields via Explicit Formulae*. Cryptology ePrint Archive, Report 2002/121.
11. T. Lange and M. Stevens. *Efficient doubling for genus two curves over binary fields*. SAC 2004. LNCS 3357, 170–181, Springer, 2005.
12. C. Lim and P. Lee. *More flexible exponentiation with precomputation*. Crypto '94. LNCS 839, 95–107, Springer 1994.
13. J. López and R. Dahab. *High-speed software multiplication in  $\mathbb{F}_{2^m}$* . INDOCRYPT 2000. LNCS 1977, 203–212, Springer, 2000.
14. J. Pelzl, T. Wollinger, J. Guajardo and C. Paar. *Hyperelliptic curve cryptosystems: closing the performance gap to elliptic curves (Update)*. IACR ePrint 2003/026.
15. J. Pelzl, T. Wollinger and C. Paar. *Low cost Security: Explicit Formulae for Genus 4 Hyperelliptic Curves*. SAC 2003. LNCS 3006, 1–16, Springer, 2004.
16. N. Pippenger. *On the evaluation of powers and related problems (preliminary version)*. 17th Annual Symp. on Foundations of Comp. Sci., IEEE Computer Society, 1976, 258–263.
17. A. Schönhage, A.F.W. Grotefeld, and E. Vetter. *Fast Algorithms—A Multitape Turing Machine Implementation*. BI Wissenschafts-Verlag, Mannheim, 1994.
18. R. Schroepel, H. Orman, S. O'Malley and O. Spatscheck. *Fast key exchange with elliptic curve systems*. Crypto '95. LNCS 963, 43–56, Springer, 1995.
19. V. Shoup. *NTL: A Library for doing number theory*. URL: <http://shoup.net/ntl/>
20. Sun Corporation's Elliptic Curve Cryptography contributions to OpenSSL. Available at <http://research.sun.com/projects/crypto/>
21. A. Weimerskirch, D. Stebila, and S.C. Shantz. *Generic  $GF(2^m)$  Arithmetic in Software and its Application to ECC*. ACISP 2003, LNCS 2727, 79–92. Springer, 2003.
22. T. Wollinger. *Software and Hardware Implementation of Hyperelliptic Curve Cryptosystems*. Ph.D. Thesis, Ruhr-Universität Bochum, Germany, 2004.
23. T. Wollinger, J. Pelzl, and C. Paar. *Cantor versus Harley: Optimization and Analysis of Explicit Formulae for Hyperelliptic Curve Cryptosystems*. To appear in IEEE Transactions on Computers.

**Disclaimer:** *The information in this document reflects only the authors' views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.*