The mpFq library and implementing curve-based key exchanges

Pierrick Gaudry, Emmanuel Thomé

The $\mathtt{mp}\mathbb{F}_q$ library and implementing curve-based key exchanges

P. Gaudry and E. Thomé

June 8, 2007

Abstract

We present a library for finite field arithmetic. The originality of this library lies in the fact that specialized code is automatically produced for the selected finite fields. The opportunity of compile-time optimizations yields substantial performance improvements compared to libraries which initialize the finite field at runtime. This library is used to present benchmarks on some curve-based public key cryptosystems.

1 Introduction

Cryptosystems based on the discrete logarithm problem in (Jacobians of) curves are competitive in many contexts. The main advantage compared to systems based on the factorization problem or on the discrete logarithm problem in finite fields is that the best known algorithm for attacking has exponential time instead of subexponential. In practice it means that for obtaining a given security, the sizes of the parameters are smaller.

The speed of an implementation of a curve-based cryptosystem is mostly given by the speed of the underlying finite field arithmetic. Once the particularities of the finite field implementation are known, one can search in the literature the most suitable choice for the coordinate systems and for the addition chain. This also depends on the amount of memory available, and if resistance to side-channel attacks is required.

In this paper we describe a new (still in development) finite field library called $\mathtt{mp}\mathbb{F}_q$ that we have used to write curve-based cryptosystems. The main objective of this library is speed. Portability, readability, ease of maintenance, ease of use are also wanted, but we accept no feature in the design of the library that would prevent us to apply certain optimizations.

Setting as a goal the implementation of fast curve-based cryptosystems forces a particularity of the underlying finite field operations: The finite field is known at compile-time. This enables a considerable amount of optimizations. The design of the $\mathtt{mp}\mathbb{F}_q$ library is suited to this problem: provide optimized code that takes advantage of all the information that is known at compile-time.

In Section 2 we give an overview of the existing software, and list the requirements for the $\texttt{mp}\mathbb{F}_q$ library. In Section 3 we explain with more detail the design of $\texttt{mp}\mathbb{F}_q$ and why certain choices were made. In Section 4 we give some timings. Finally in Section 5 we describe a few BATs that have been implemented with the help of $\texttt{mp}\mathbb{F}_q$. We conclude with some plans for the future.

2 Why yet another finite field library?

2.1 Existing finite field libraries

Obviously, there are several already existing software libraries that can be used to perform computations in finite fields. We briefly review here some of them.

The NTL library [17] by V. Shoup provides arithmetic modulo finite fields, and also goes well beyond that. NTL is written in a small subset of C++, and based on selectable multiprecision arithmetic packages (including the Gnu MP library [12]). NTL has good performance in general and very good performance for small prime fields, using IEEE floating point arithmetic for the reduction step.

The ZEN library [7] by F. Chabaud and R. Lercier is a C library for finite field arithmetic. ZEN handles arbitrary finite field (extension of extensions for instance). Although ZEN is written in ANSI C, it should really be regarded as an object-oriented implementation in the same spirit as X11: Almost every user-land identifier from the high-level interface is a macro that calls a function obtained by dereferencing a pointer in the last passed argument (the field). The high-level interface is well-documented. However, for best performance, if the finite field one is working in is known in advance when writing the code, it is possible to call directly the lower-level functions, thereby saving an indirect function call. ZEN has several lower-level layers, including specialized arithmetic for one-word-long modulus. This only goes to a limited extent, however, since the lower-level interface of ZEN is not documented.

The Miracl library by M. Scott [16] provides an optimized feature set for cryptographical operations. It relies on a finite field layer whose performance appears to be good. Miracl goes well beyond finite field operations, since algorithms such as elliptic curve point counting or algorithms for computing pairings are included.

We also mention the Givaro library by J.-G. Dumas *et al.* [9]. It is written in C++ templates, and claims to perform well for one-word-long modulus.

Besides the libraries cited above, which strive for providing arithmetic for all finite fields, there are also software libraries which focus on particular finite fields. The NuMongo library by R. Avanzi [3] handles specially selected prime fields, with the modulus having a special form. As far as we know, it contains only 32-bit code and is not publicly available.

2.2 The opportunity of compile-time optimizations

As mentioned in the introduction, the primary goal that started the development of $\texttt{mp}\mathbb{F}_q$ was the implementation of fast curve-based cryptosystems. For such an application, the finite field is known in advance. Since speed is desired, one wants to take advantage of this knowledge when building the library. While the performance improvement might seem limited if one has in mind fields modulo 1024-bit primes, this kind of optimization makes a vast difference for small fields.

The contexts with an opportunity for compile-time optimization also includes long-running computations on one or several selected finite fields. In particular, cryptanalysis attempts such as the breaking of the CERTICOM ECC Challenges [6, 13] fall in this category. In the precise example of [13], specially crafted code was written in order to have fast finite field operations. Other settings in which specific code was written include [10, 4].

Several kinds of optimizations are made possible by the knowledge of the finite field at compile time. These include the following list of methods. All these optimizations can easily be performed at compile time, but it is not so easy to do the same at runtime, and in many cases it is impossible.

- Code inlining. At runtime, this is not clear which code to inline, because it might depend on the field.
- Branch elimination. If the values determining the control flow are constant at compile time, then the branches can be avoided. This reduces the cost of loops and conditionals.
- Loop unrolling. Nowadays compilers do it automatically, but the unrolling is more efficient if the length of the loop is a constant.
- Choice of the best algorithm for a given task. This can be done also at runtime (ZEN does so), but is more comfortable at compile time (and comes with zero runtime overhead).

Of the finite field libraries mentioned above, only those focusing on a handful of specialized finite fields (like NuMongo) have the opportunity to take full advantage of the optimizations above. By design, none of the other libraries which provide arithmetic for general finite fields are in position to exploit these optimizations.

It is always conceivable to overload some particular library with a new class for the particular field we are interested in, but no simple mechanism is provided to help the developer in this task.

2.3 Wanted features of $\texttt{mp}\mathbb{F}_q$

From our experience emerges a need for a software library for finite fields which differs from the existing material. Briefly put, the two main differences are:

- $\texttt{mp}\mathbb{F}_q$ has to handle very efficiently finite fields that are known at compile time.
- $\texttt{mp}\mathbb{F}_q$'s specialized code for the selected finite fields should be written automatically, rather than crafted by hand.

$\texttt{mp}\mathbb{F}_q$ sets some goals. The utmost concern is speed, obviously. $\texttt{mp}\mathbb{F}_q$ is not contented with merely working code, since this is not good enough for our claimed purposes. While the automatic process of generating specialized code should take sensible choices, we require that these choices may be overridden easily, or that radically different implementation choices may be taken by the user in a way that is reasonably compatible with the rest of the library. Indeed, there is quite often no single answer to the question "which is *the* fastest implementation of $\mathbb{F}_q$?" for a given value of q. Depending on the intended application, the different finite field operations are not necessarily used equally frequently; in situations where optimizing an operation penalizes another one at the same time, the relevant optimization choices therefore depend on the application.

Since the underlying finite field might be in several cases a parameter of an algorithm, $\texttt{mp}\mathbb{F}_q$ has to provide a consistent application programming interface (API) in order to allow code reuse.

In the last few years, the processors that are available in average workstations have become multi-core. This means that for most applications, having a multi-threaded implementation is a good way to gain efficiency. For this reason, we require that the $\texttt{mp}\mathbb{F}_q$ code be reentrant, so that they are usable within a multi-threaded application.

The dependencies of $\texttt{mp}\mathbb{F}_q$ are free software, $\texttt{mp}\mathbb{F}_q$ itself being free software licensed under the terms of the Gnu Lesser General Public License (LGPL)[1].

3 The design of $\texttt{mp}\mathbb{F}_q$

3.1 Choice of the programming languages

The programming languages used by $\texttt{mp}\mathbb{F}_q$ are `perl`, C, and assembly. The automatic generation of specialized code for a selectable finite field is done by `perl`. From the description of the finite field, the `perl` code creates a C source file and header, which provides the required set of functionalities.

The choice of `perl` calls for some comments. The C++ language offers several methods which could provide the needed genericity. An object-oriented approach with a virtual base class would fail on the speed requirement. The unavoidable indirect function call for virtual methods would be a major performance hit, in particular for the sizes we are focusing on (for small sizes, an inlined implementation of the operation would be most suitable). Furthermore, the variety of fields on which different specialized code is needed would require different classes to be generated by other means anyway.

The template mechanism from C++ provides essentially what is needed. Indeed, this allows a static overriding of functions, leaving the possibility of inlining. We believe however

[1] By the time of the Speed conference, we plan to distribute a first beta release of $\texttt{mp}\mathbb{F}_q$.

that the kind of syntactic manipulation that is offered by C++ templates can also be obtained by other means. At the expense of losing the type checking of code specializations, we have opted for doing the code generation using `perl`, which is best suited for text manipulation.

3.1.1 The dilemma of assembly

In many cases, writing critical routines in assembly is required for gaining speed. This is due to strict limitations of the C language concerning arithmetic. For instance, after adding two machine-words, if there is an overflow (a carry), the C language ignores it, whereas on many platforms this information is still available at the assembly level. Also the popular `x86` architecture gives access to the full double-word result of a multiplication, whereas only the low significant word is reachable from C. A third example is assembly instructions specific to some platform, like for instance the SSE-2 instructions set [1]. Such instructions and the corresponding data types are not accessible with standard C.

There are several approaches to writing assembly. It is possible to write a standalone function, that respects the application binary interface (ABI) of the C compiler/system we use. This is very convenient but it means that a possibly costly function call has to be payed each and every time one needs it. The other approach is not standard, but available in many compilers, including the Gnu C compiler (GCC). It consists in inline assembly language insertion using the `asm()` keyword. The programmer must tell the compiler which registers are used for input, ouput and temporary usage during the assembly stage. This has the advantage of avoiding the function call. Another option is language extensions which are specific to some compilers. Several compilers (including GCC as well as compilers from Intel, Microsoft) provide the `emmintrin.h` include file, with for instance the `_mm_slli_epi64` macro which corresponds to the `psrlq` assembly instruction (which right-shifts a 128-bit SSE-2 register).

Deciding between standalone functions and inline assembly language is a matter of length of the assembly code, but not only. One could think that writing a C function as a list of small blocks of `asm()` interleaved with pure C code is a good idea, but sometimes it appears that it is better to write the whole function in assembly to help the compiler in register allocation and handling the data flow. Unfortunately, right now, we have not yet been able to find a strict rule for when an approach should be chosen or another, and trying several implementations is the only resort.

3.2 The API and function naming

The API of `mp`$\mathbb{F}_q$ is as follows: Let `TAG` be a mnemonic that corresponds to a finite field or a family of finite fields. In fact, `TAG` will also be different if the internal representation of elements of the same finite field are different. For instance, `2_27` can be the mnemonic for the finite field with 2^{27} elements in classical (polynomial basis) representation; or `p_mgy_3` can be the mnemonic for the family of prime finite fields where the modulus fits in 3 machine-words and the elements are in Montgomery representation. Then the C types and

the C functions corresponding to this mnemonic start with `mpfq_TAG`. For instance, an element of $\mathbb{F}_{2^{27}}$ will have a type `mpfq_2_27_elt`, and the multiplication function between two such elements will be called `mpfq_2_27_mul`.

3.3 Macros and inline functions

All these types and functions are stored in two files: `mpfq_TAG.c` and `mpfq_TAG.h`. The speed requirement means that all the functions that take less than (say) a few hundred cycles must be inlined. Instead of using the C preprocessor for defining macros, `mp`$\mathbb{F}_q$ is developed using `static inline` C functions that are included in the `.h` file. The inlining effect is the same, but compared to a macro, this has the advantage to allow the compiler to perform a type checking and to facilitate the debugging (when switching off the inlining optimisation options of the compiler). For instance, in $\mathbb{F}_{2^{89}}$ the `add` function will be as follows:

```
static inline mpfq_2_89_add(mpfq_2_89_field_ptr K,
  mpfq_2_89_dst_elt r, mpfq_2_89_src_elt s1, mpfq_2_89_src_elt s2)
{
    r[0] = s1[0]^s2[0];
    r[1] = s1[1]^s2[1];
}
```

This example calls for additional remarks:

- The first argument of most `mp`$\mathbb{F}_q$ functions is a pointer to the finite field in which the operation takes place. For most implementations, it is not used, but in some case it is convenient. For instance if the implementation of the multiplication covers all prime fields of a given size, then the modulus should be accessible to the function, and is then obtained from the first argument.

- The type for an element is split into two subtypes marked `src` and `dst`. This follows GMP practice to distinguish between `const` and variable arguments. Adding the keyword `const` to a variable sometimes helps the compiler to choose the right optimization.

3.4 Code generation

It should be now evident that there will be a lot of redundancy between different `.c` and `.h` files of `mp`$\mathbb{F}_q$. To avoid the problems with maintaining a code with a lot of code duplication, we have chosen to have most of the C and assembly sources of `mp`$\mathbb{F}_q$ generated automatically by `perl` scripts. The power of `perl` with manipulating files, strings, regular expressions makes it a very nice alternative to any macro-based preprocessing (like CPP or M4) or to a template-based C++ approach.

We give an example of the power of this approach: when writing the code for the trace of an element in a specific finite field of characteristic 2 in polynomial basis representation,

one wants to precompute the powers of the defining element x that have trace 1, in order to create the mask. If you do not allow a powerful enough language for the preprocessing, this precomputation will have to be stored and shipped with the library sources, whereas a `perl` script has no problem to do this precomputation on the fly, just before creating the appropriate C function. It is even possible, if some precomputations would be tedious in `perl`, to hand off some of the work to an external program in C ($\texttt{mp}\mathbb{F}_q$ uses such a convenience).

We have implemented a main `perl` module that helps in the organization of the code generation. Several `perl` scripts in $\texttt{mp}\mathbb{F}_q$ generate code, but they are not handling the global organization themselves. Instead, the API for $\texttt{mp}\mathbb{F}_q$ for all finite fields is concentrated in a single file `api.pl` that lists the functions that should be present, together with their prototypes. The generation of the `.c` and `.h` files for a given mnemonic `TAG` is done by the main module, which iterates over the functions in the API. It delegates the generation of these functions to the specialized scripts, fetching `perl` subroutines named `code_for_<function_to_generate>`. At the end the main module reconstructs both files from all the codes, and creates the appropriate prototypes. This approach allows to enforce conformance to the API.

4 Benchmarks

4.1 Development status of $\texttt{mp}\mathbb{F}_q$

The API and the main `perl` module of $\texttt{mp}\mathbb{F}_q$ are more or less fixed. $\texttt{mp}\mathbb{F}_q$ will probably gain more functionalities gradually, and most importantly we need to improve speed at every possible level. Until now we have focused on 64-bit architectures based on the AMD64 instruction set. This covers essentially all the processors currently sold for personal computers and workstations (Athlon64, Opteron, Core2, recent Xeon). We believe that in a near future, 32-bit architectures will be found only in embedded systems. Our code works on 32-bit architectures but the assembly support is inexistent or very poorly written. Furthermore, on many 32-bit architectures, the floating-point unit is more powerful than the integer unit, so that this would probably give the best performance, and we didn't implement this.

Apart from the target architecture, we have been concentrating in optimizing the finite fields that are needed for our BATs. In particular, for a prime field modulus which is not sparse, the Montgomery representation should probably be used, but this is not yet properly set in $\texttt{mp}\mathbb{F}_q$, so that the benchmarks are somewhat deceiving[2].

While $\texttt{mp}\mathbb{F}_q$ has good performance for some operations, some other are in dire need for improvement (finite field inversion notably).

[2]Montgomery representation arithmetic will probably be in place by the time of the Speed conference.

4.2 Benchmarking methodology

We start with a word of warning: measuring the cost of a small operation is essentially impossible on modern computers. Assume that an operation takes 20 cycles; assume also that this operation is implemented in an inlined function (in C or assembly, this does not really matter for this discussion). The cost of setting up the data and preparing the registers at the beginning might take a few cycles (say 4 cycles), and this task is done by the compiler. However, those 4 cycles might become much less if the context of the function call is favorable (that's one of the advantages of inlining). Therefore, some discrepancy is inherently attached to the measurement.

There are basically two ways of timing an operation: either ask the operating system (with the `getrusage()` function) or use the tick counter of the processor (the `rdtsc` assembly instruction on `x86`). The first approach is fine only for very long tasks, since the precision is of the order of the millisecond. The second approach is suitable only for rather short tasks, since any interruption or context switch of the system will perturbate the measure. Also, there is some kind of "Observer effect" for very small operations: a call to `rdtsc` is not serializing, which means that there is no guarantee that the instructions are executed in the order they are written. This is of course not good for our purpose. There is a variant of `rdtsc` that is serializing (or one can add some serializing operation before and calibrate it), but then we really perturbate the operation we are measuring, since there is a high risk of flushing the pipeline.

In our context, we have mostly used the `getrusage` approach for our measures. Since the operations we want to measure are small, we repeat them a large number of times and divide the running time accordingly. On a few tests we have made, the results are not too far from the other approach based on `rdtsc`, and consistent with the running times of the BATs we have built upon the measured operations. But an operation like an addition in $\mathbb{F}_{2^{113}}$ can definitely not be measured in an optimized implementation, since its cost is essentially just the cost of fetching the appropriate data: if it is already in registers, then the operation will be less than 2 cycles or even zero (if the `xor`'s can be inserted between higher latency instructions), but if this operation must be done at a time where all the registers are already occupied, moving data between the stack and the registers can cost a non-negligible time.

4.3 Cost of basic operations for prime fields

For prime fields, we have written `mp`$\mathbb{F}_q$ implementations for each machine word size of the modulus (up to nine words) and for the two finite fields that we use in our cryptographic applications. The algorithms we have implemented are by no means original (classical representation and we have used a basic binary extended GCD for the inversion). The costs of basic operations for these fields are given in Table 1 and Table 2. We also give similar benchmarks in NTL and ZEN for comparison. NTL and ZEN do not take advantage of the pseudo-Mersenne form of the modulus. However, in ZEN one can activate a Montgomery representation that speeds-up computations, so we give both timings.

We can see that $\texttt{mp}\mathbb{F}_q$ is faster than NTL for all sizes. ZEN with Montgomery representation is comparable or slower than NTL except for 1 word primes, where it is sometimes faster than $\texttt{mp}\mathbb{F}_q$. As one can expect, the difference is more visible for small sizes, and on Opteron, since our assembly code is best suited to this processor. The gain obtained by writing a reduction procedure that is specific to a pseudo-Mersenne modulus is visible on the last two columns.

We conclude with a comment on the usual practice in curve-based cryptography: quite often, to compare the costs of different coordinate systems or addition chains, only the multiplications and squarings are counted, and the additions are said to be negligible. This is clearly not the case, for instance for the field $\mathbb{F}_{2^{255}-19}$ on the Opteron where the `mul/add` ratio is less than 6. The same kind of ratio is observed with the ZEN and NTL libraries, and is even amplified by the fact that an addition or subtraction is much slower than with $\texttt{mp}\mathbb{F}_q$.

Table 1: Time (in nanoseconds, with 2 significant digits) for basic operations in $\mathbb{F}_p$ on an **AMD Opteron** 250 processor at 2.40 GHz.

$\texttt{mp}\mathbb{F}_q$:

	1 word	2 words	3 words	4 words	$2^{127}-735$	$2^{255}-19$
add	2	4	5	7	4	8
sub	2	3	5	5	4	9
sqr	67	108	170	230	14	30
mul	66	109	180	240	16	45
inv	420	2600	4600	7500	2600	7400

NTL:

	1 word	2 words	3 words	4 words
add	40	42	36	47
sub	38	40	28	44
sqr	120	150	230	290
mul	120	150	230	290
inv	1600	4400	6600	9200

ZEN/ZENmgy:

	1 word	2 words	3 words	4 words
add	8/11	44/44	44/44	48/49
sub	7/8	64/71	66/70	73/75
sqr	62/90	270/170	420/270	520/320
mul	68/95	300/180	450/270	600/340
inv	1700/2100	3300/4300	4800/5900	6500/7500

Table 2: Time (in nanoseconds, with 2 significant digits) for basic operations in $\mathbb{F}_p$ on an **Intel Core2** 6700 processor at 2.66 GHz.

$\texttt{mp}\mathbb{F}_q$:

	1 word	2 words	3 words	4 words	$2^{127}-735$	$2^{255}-19$
add	1	2	4	8	3	8
sub	1	4	5	7	3	9
sqr	73	110	180	240	17	40
mul	74	120	190	260	19	53
inv	300	2000	3600	5800	2000	5800

NTL:

	1 word	2 words	3 words	4 words
add	38	45	53	67
sub	38	45	52	64
sqr	110	130	210	270
mul	110	140	210	270
inv	1200	3400	5800	8000

ZEN/ZENmgy:

	1 word	2 words	3 words	4 words
add	6/6	41/41	46/46	57/57
sub	4/4	54/60	60/62	73/78
sqr	52/52	280/120	400/170	550/250
mul	52/60	280/120	400/180	590/260
inv	1000/1000	2500/3000	3800/4300	5000/5900

4.4 Cost of basic operations for binary fields

The binary fields up to $\mathbb{F}_{2^{255}}$ are implemented in $\texttt{mp}\mathbb{F}_q$, using a polynomial basis representation. We have chosen defining polynomials with lowest possible Hamming weight. Figure 1 shows the performance of the multiplication, squaring and inversion using $\texttt{mp}\mathbb{F}_q$ compared to the NTL and ZEN libraries for these fields. The figure also indicates the timings for the "unreduced" multiplication and squaring operations. The graphs on the left side correspond to timings on an AMD Opteron CPU at 2.40 GHz, while the graphs on the right side correspond to timings on an Intel Core2 CPU at 2.66 GHz.

The algorithms implemented for the different operations are the classical ones described for instance in [8, chap. 11]. So far, the multiplication is done using the schoolbook algorithm, but Karatsuba and Toom-Cook variants have to be measured in comparison. It appears that for all sizes, the best performance for the multiplication is attained by using the SSE-2 instruction set [1]. Using these instructions, it is effectively possible to work in parallel with two 64-bit machine words at a time. The performance gain is most remarkable on the Intel Core2 CPU.

The comparison with the NTL library shows that $\texttt{mp}\mathbb{F}_q$ is faster than NTL except in

a few situations. The ZEN library is somewhat slower than NTL in particular for the inversion (we did not investigate where this problem could come from). In our ZEN test program, we have activated the precomputations that could yield speedups, but we have not tried to split the extension in a double extension; in the documentation of ZENfact (a submodule of ZEN), there are examples of such constructions that provide a speedup, but this is not really automatic. We have also skipped the optimization that builds a logarithm table, since this is valid only for tiny fields.

The inversion in $\texttt{mp}\mathbb{F}_q$ has not been looked at seriously, it merely has the merit of giving correct results. Concerning the multiplication, the relative under-performance of the SSE-2 implementation on the AMD Opteron CPU is probably explained by the different implementation of the SSE-2 pipeline on this particular CPU compared to the Intel Core2. On both CPUs, the large steps around 2^{250} call for further optimization, and will be investigated. For this purpose, an automatic tuning program is being prepared. We mention that NTL has a conspicuous problem for finite fields smaller than $\mathbb{F}_{2^{64}}$. This should probably not be worried about and should be considered as an easy tuning issue.

5 Writing BATs with $\texttt{mp}\mathbb{F}_q$

We have used $\texttt{mp}\mathbb{F}_q$ to write efficient software implementation of the Diffie-Hellman key exchange protocol based on curves. We started with the `curve25519` parameters given by Bernstein [4]: this is an elliptic curve in Montgomery form defined over $\mathbb{F}_{2^{255}-19}$, such that both the curve and the twist are secure. We obtain the following timings on our two test machines.

`curve25519`		
	Opteron 2.40 GHz	Core2 2.66 GHz
Time for one scalar mult. in μsecs	128	145
Time for one scalar mult. in cycles	307,000	386,000
Number of scalar mul. per second	7800	6900

We have designed a cryptosystem of genus 2 of the same level of security that we called `surf127eps`. It is based on a genus 2 curve defined over $\mathbb{F}_{2^{127}-735}$ that has complex multiplication by $K = \mathbb{Q}\left(i\sqrt{5+\sqrt{53}}\right)$. The Jacobian of this curve has an order which is 16 times a prime and is suitable for using the Kummer surface formulae of [11] that we have implemented. We obtain the following timings on our two test machines.

`surf127eps`		
	Opteron 2.40 GHz	Core2 2.66 GHz
Time for one scalar mult. in μsecs	116	154
Time for one scalar mult. in cycles	279,000	410,000
Number of scalar mul. per second	8600	6500

We can see that for prime fields, the Opteron behaves better than the Core2. This might be surprising since this Opteron is a 3-year old computer, whereas the Core2 is

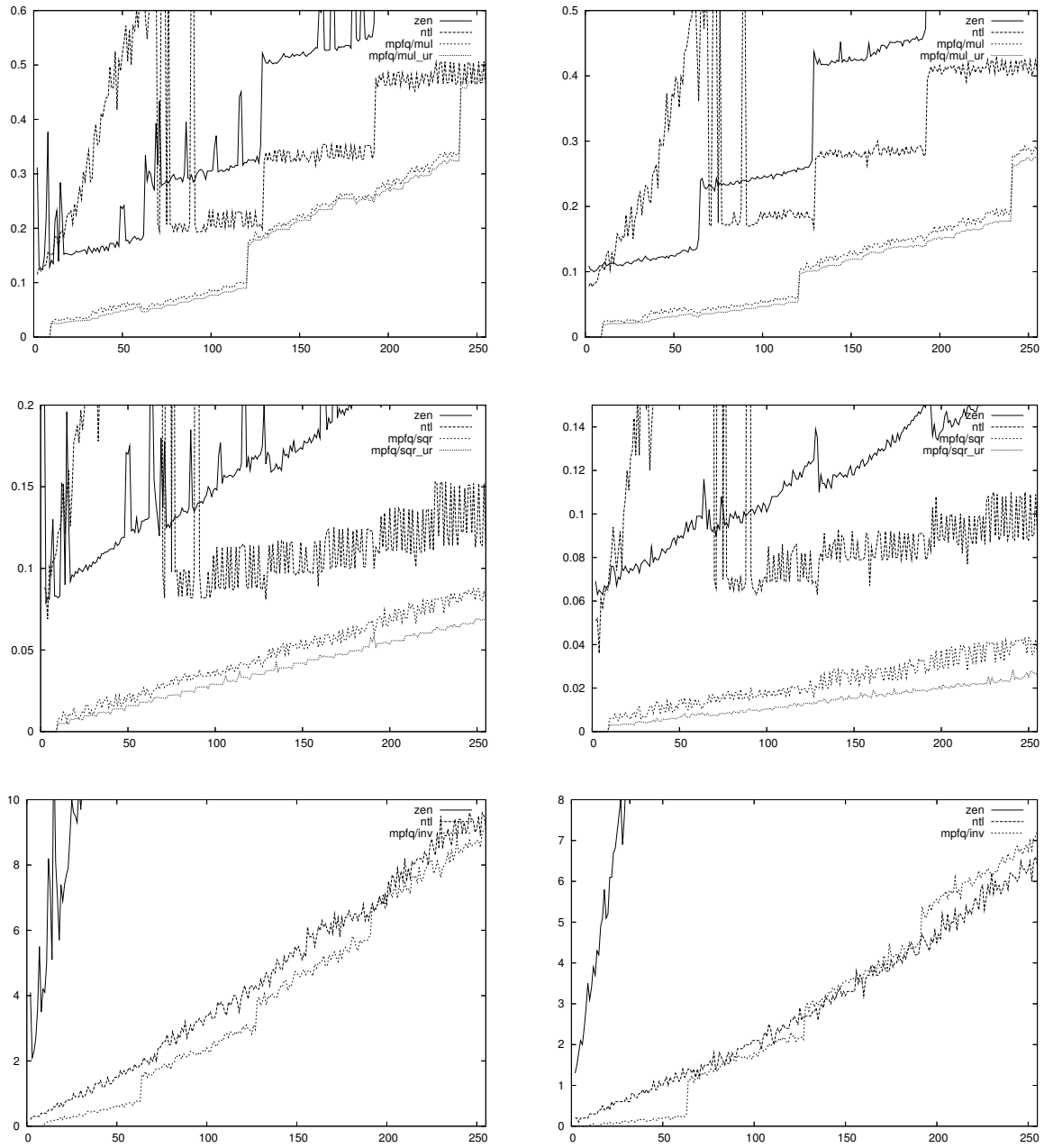

Figure 1: Time (in microseconds) for Multiplication, Squaring, Inversion in $\mathbb{F}_{2^n}$
(Left column is on AMD Opteron 2.40 GHz, right column is on Intel Core 2 2.66 GHz)

a brand new architecture; however our skills to optimize assembly code for the Core2 is clearly not the same as for Opteron.

The other observation is that a genus 2 cryptosystem can beat an elliptic one with our implementation, depending on the processor. However, if a general, efficient genus 2 point counting implementation were available, one could construct a Kummer surface with small coefficients, thus saving a lot of operations (as shown by Bernstein [5]). We expect that the situation would be constantly in favour of genus 2.

In order to test our library and to measure the difference between prime fields and characteristic 2 curve based cryptosystems, we have also implemented the scalar multiplication on elliptic curves in characteristic 2, based on the formulae by Stam [18], over the finite field $\mathbb{F}_{2^{251}}$ and a genus 2 scalar multiplication based on the Kummer surface, over the finite field $\mathbb{F}_{2^{113}}$. This time there is no problem for the point counting in genus 2, thanks to p-adic methods (we have used Magma for these point counting computations).

The results are the following. It should be noted that the performance suffers from the lack of fine-tuning of the multiplication algorithm (in particular, we acknowledge that the multiplication in $\mathbb{F}_{2^{251}}$ is still sub-optimal).

`curve2_251`		
	Opteron 2.40 GHz	Core2 2.66 GHz
Time for one scalar mult. in μsecs	863	506
Time for one scalar mult. in cycles	2,070,000	1,350,000
Number of scalar mul. per second	1100	2000

`surf2_113`		
	Opteron 2.40 GHz	Core2 2.66 GHz
Time for one scalar mult. in μsecs	441	268
Time for one scalar mult. in cycles	1,060,000	713,000
Number of scalar mul. per second	2200	3700

Comparison with other scalar multiplication implementations

The usual problem when comparing timings is that the computers are quickly evolving, so that comparison is difficult. This is particularly true in the present case, where we are concentrating only on 64-bit architectures, whereas almost all implementations reported in the literature rely on 32-bit architecture. This is strange, since the Opteron has been sold for 4 years, now, and the gain of using 64-bit is clear.

Additionally, usually the reported implementations are there to illustrate some improvement in the group law formulae, so that no two papers are really comparable if one is mostly interested in the underlying finite field implementation. Therefore we give the raw data, without trying to scale it to our experiment platform or to the coordinate system we choose.

For each reference, we mention the result that corresponds more or less to the security level we have chosen.

In [2], an implementation of curve arithmetic in characteristic 2 has been written, based on a carefully written set of finite field routines. There timings are given on a 32-bit Power G4 at 1.5 GHz. In genus 1 over $\mathbb{F}_{2^{251}}$, a scalar multiplication takes 3758 microseconds. In genus 2 over $\mathbb{F}_{2^{109}}$, a scalar multiplication takes 1673 microseconds.

In [4] where `curve25519` is described, Bernstein reports an implementation between 620000 and 950000 cycles depending on the processor. All the processors that are considered are 32-bit, and therefore floating point arithmetic used. In [5], he gives a genus 2 implementation (very similar to `surf127eps`) that takes 580000 cycles on a Pentium M.

In [19], an implementation in characteristic 2 gives the following timings on a Pentium 4 at 1.8 GHz: in genus 1 over $\mathbb{F}_{2^{191}}$, a scalar multiplication takes 2780 microseconds and in genus 2 over $\mathbb{F}_{2^{95}}$, it takes 3410 microseconds.

In [3], Avanzi has used his NuMongo library to implement scalar multiplication for curves over prime fields. The timings are for an Athlon 1 GHz: in genus 1 over a 256-bit prime field, it takes 3048 microseconds and in genus 2 over a 128-bit prime field, it takes 3575 microseconds.

6 Future plans

The future directions of $\texttt{mp}\mathbb{F}_q$ are numerous, given the amount of algorithms that would be worth giving a try. For prime fields, we need to adapt our implementation of Montgomery's REDC algorithm [15] to the $\texttt{mp}\mathbb{F}_q$ library. We also plan to improve the inversion both on prime and binary fields.

Given the growing interest in pairing-based cryptography, we will probably provide implementations of extension field arithmetic (with specialized code for the most frequently used extension degrees).

It is planned to extend $\texttt{mp}\mathbb{F}_q$ to handle polynomials and matrices over finite fields and generate optimized source code files for this purpose. This might lead us to consider the case of large polynomials, and include FFT algorithms which are suited to the base fields used.

As for the BATs, we still have room for improvements in the choice of the addition chain. Right now, we have used the most basic binary ladder. We plan to try some of the heuristic algorithms available in the literature for finding better addition chains, for instance the so-called PRAC algorithm by Montgomery [14].

Acknowledgments

Although we are only two authors, we rely heavily on GMP, not only as a dependency library, but also as a source of inspiration for the design of $\texttt{mp}\mathbb{F}_q$. We have also taken ideas from ZEN, NTL and from software written by R. Harley for the ECDL challenges. We wish to thank Paul Zimmermann and Richard Brent who shared several ideas with us on the topic of muliplication in binary fields.

The genus 2 curve that we use for the BAT called Surf127-735 has been generated by the CM method using tools written by T. Houtmann.

References

[1] Advances Micro Devices. *AMD64 Architecture Programmer's Manual, Volume 4: 128-Bit Media Instructions*, 2005.

[2] R. Avanzi, N. Thériault, and Z. Wang. Rethinking low genus hyperelliptic jacobian arithmetic over binary fields: interplay of field arithmetic and explicit formulae, 2006. Preprint available at `http://www.cacr.math.uwaterloo.ca/techreports/2006/cacr2006-07.pdf`.

[3] R. M. Avanzi. Aspects of hyperelliptic curves over large prime fields in software implementations. In M. Joye and J.-J. Quisquater, eds., *CHES 2004*, vol. 3156 of *Lecture Notes in Comput. Sci.*, pp. 148–162. Springer–Verlag, 2004. Proc. 6th International Workshop on Cryptographic Hardware and Embedded Systems, Cambridge, MA, USA, August 11-13, 2004.

[4] D. J. Bernstein. Curve25519: new diffie-hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, eds., *Public Key Cryptography – PKC 2006*, vol. 3958 of *Lecture Notes in Comput. Sci.*, pp. 207–228. Springer–Verlag, 2006. Proc. 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006.

[5] D. J. Bernstein. Elliptic vs. hyperelliptic, part 1, 2006. Talk given at ECC 2006. Slides available at `http://cr.yp.to/talks.html#2006.09.20`.

[6] Certicom corp. The Certicom ECC challenges, 1997. Description at `http://www.certicom.com/index.php?action=ecc,ecc_challenge`.

[7] F. Chabaud and R. Lercier. ZEN, user manual, 1996–2007. Homepage at `http://zenfact.sourceforge.net/`.

[8] H. Cohen and G. Frey, eds. *Handbook of elliptic and hyperelliptic curve cryptography*. Chapman & Hall / CRC, 2005.

[9] J.-G. Dumas, T. Gautier, P. Giorgi, J.-L. Roch, and G. Villard. Givaro, une bibliothèque C++ pour le calcul formel, 1987–2007. Homepage at `http://ljk.imag.fr/CASYS/LOGICIELS/givaro/`.

[10] M. Fouquet, P. Gaudry, and R. Harley. Finding secure curves with the Satoh-FGH algorithm and an early-abort strategy. In B. Pfitzmann, ed., *Advances in Cryptology – EUROCRYPT 2001*, vol. 2045 of *Lecture Notes in Comput. Sci.*, pp. 14–29. Springer-Verlag, 2001.

[11] P. Gaudry. Fast genus 2 arithmetic based on Theta functions. *J. of Mathematical Cryptology*, 2007. To appear. Preprint available at `http://eprint.iacr.org/2005/314`.

[12] T. Granlund. `GMP`, the GNU multiple precision arithmetic library, 1993–2007. Homepage at `http://gmplib.org/`.

[13] R. Harley. The ECDL project. `http://cristal.inria.fr/~harley/ecdl/`, 2000.

[14] P. L. Montgomery. Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, xm - n)$ via Lucas chains, 1983. Preprint available at `ftp.cwi.nl:/pub/pmontgom/Lucas.ps.gz`.

[15] P. L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(170):519–521, Apr. 1985.

[16] M. Scott. MIRACL: Multiprecision integer and rational arithmetic c/c++ library, 1988–2007. Homepage at `http://www.shamus.ie/`.

[17] V. Shoup. NTL: A library for doing number theory, 1990–2007. Homepage at `http://www.shoup.net/ntl/`.

[18] M. Stam. On Montgomery-like representations for elliptic curves over $GF(2^k)$. In Y. G. Desmedt, ed., *Public Key Cryptography – PKC 2003*, vol. 2567 of *Lecture Notes in Comput. Sci.*, pp. 240–254. Springer–Verlag, 2003.

[19] T. Wollinger, J. Pelzl, and C. Paar. Cantor versus Harley: Optimization and analysis of explicit formulae for hyperelliptic curve cryptosystems. *IEEE Trans. Comput.*, 54:861–872, 2005.