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

(yet another finite field library)

Emmanuel Thomé, Pierrick Gaudry

INRIA LORRAINE

C A C A O

Context

This talk is about

- software
- for finite field arithmetic ($+ * \div \ldots$; most importantly over $\mathbb{F}_p$ and $\mathbb{F}_{2^n}$)
- at high SPEED.

Plan

1. **Introduction**
2. **What's inside**
3. **Typical optimizations**
4. **Results**

1. Introduction
2. What’s inside
3. Typical optimizations
4. Results

Finite field arithmetic

Finite field arithmetic is ubiquitous !

- in computational mathematics ;
- in coding theory ;
- in public-key cryptography (curve-based cryptosystems, pairings, . . .) ;
- in cryptanalysis ;
- . . .
- . . .

Two ways of using a finite field library

Either :

- The same compiled code can compute in $\mathbb{F}_{2^{31}}$, $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{255}-19}$.

  $\Rightarrow$ run-time mode.

  Example : magma, ...

- Or each new field requires the code be compiled again.

  $\Rightarrow$ compile-time mode.

  Examples :
  - fast software implementations of a cryptosystem ;
  - Computations involving a huge amount of CPU time, handling one particular finite field (e.g. for cryptanalysis).

CACAO

Existing situation

Several (countless ?) software libraries exist : NTL, ZEN, . . . no *de facto* standard.

- Software libraries are suited for run-time mode.
- For compile-time mode, most libraries fall short of speed expectations.

Quite often one reinvents the wheel.

- $\mathtt{mp}\mathbb{F}_q$ aims at providing code for compile-time mode.
- $\mathtt{mp}\mathbb{F}_q$ is more a code generator than a library.

We give a few examples of optimizations allowed by compile-time mode

1. Introduction

2. What’s inside

3. Typical optimizations

4. Results

Flowchart for mp$\mathbb{F}_q$

- A finite field is fixed (or almost ; could be « $\mathbb{F}_p$ with $2^{64} < p < 2^{128}$ »)
- A machine is fixed (or almost ; could be « any 64-bit machine »)

  mp$\mathbb{F}_q$ generates a `.h` and (sometimes) a `.c` file,

  e.g. `mpfq_p_25519.h` and `mpfq_p_25519.c`

- self-contained.
- implementing a common API : `mpfq_p_25519_mul` ; `mpfq_p_25519_sqrt` ; ...
- C with compiler extensions ; can be used in either C or C++ programs.

C A C A O

Design choices (1)

The code generator does

- a lot of text manipulation ;
- some calculations ;
- I/O to text files.

We rely on Perl code, with a little help from C programs for calculations.

Design choices (2)

The generated code does all sorts of (dirty ?) things.

- For prime fields, assembly is required for carry propagation (`addc`) and long multiplies.
- For binary fields, best SPEED calls for SIMD.
- As long as maximum SPEED is reached, we want good portability.

$\mathtt{mp}\mathbb{F}_q$ generates
- C code ; lots of inlines (macros are frowned upon)
- with inline assembly
- using some compiler extensions (« built-ins »).

This is OK with at least gcc, icc, msvc.

C A C A O

1. Introduction
2. What’s inside
3. **Typical optimizations**
4. Results

Typical compile-time optimizations

When specifying a fixed field :

- Data types can be simplified ; Data management is easier ;
- Many repeat counts become constant $\Rightarrow$ unroll !
- Modulus, definition polynomial become constants as well.

Remark : such optimizations are most relevant for small fields.

We give a few examples for binary fields.

Example for $\mathbb{F}_{2^{47}}$

Elements are polynomials of degree 46, taking up one 64-bit machine word : no indirection.

To multiply a by b, we first compute Pb for $\deg P \leqslant 3$. Then :

```
u = pb[a       & 15]; t[0]  = u;
u = pb[a >>  4 & 15]; t[0] ^= u <<  4;
u = pb[a >>  8 & 15]; t[0] ^= u <<  8;
u = pb[a >> 12 & 15]; t[0] ^= u << 12;
u = pb[a >> 16 & 15]; t[0] ^= u << 16; t[1]  = u >> 48;
/* some more */
u = pb[a >> 44 & 15]; t[0] ^= u << 44; t[1] ^= u >> 20;
```

Example for $\mathbb{F}_{2^{47}}$ (cont'd)

We have $\deg(ab) \leqslant 92$. Reduction mod $X^{47} + X^5 + 1$:

```
t[1] <<= 17; t[0] ^= t[1];
t[1] <<=  5; t[0] ^= t[1];
y = t[0] >> 47; t[0] ^= y;
y <<=  5; t[0] ^= y;
t[0] &= (1UL << 47) - 1;
```

much (much) faster than a full-length division.

Several data-dependent branches are saved.

CACAO

Hard-coding Karatsuba

Karatsuba multiplication obviously pays off very early ; example for $\mathbb{F}_{2^{256}}$.

```
mp_limb_t x1[2] = { s1[0] ^ s1[2], s1[1] ^ s1[3] };
mp_limb_t x2[2] = { s2[0] ^ s2[2], s2[1] ^ s2[3] };
mpfq_2_256_mul_basecase128x128(t,s1,s2);
mpfq_2_256_mul_basecase128x128(t+4,s1+2,s2+2);
t[2] = t[4] = t[2] ^ t[4]; t[2] ^= t[0]; t[4] ^= t[6];
t[3] = t[5] = t[3] ^ t[5]; t[3] ^= t[1]; t[5] ^= t[7];
mpfq_2_256_addmul_basecase128x128(t+2,x1,x2);
```

The tuning is done once and for all by the code generator.

C A C A O

Using SIMD instructions

- With SSE, we handle two values of 64-bit each.
- The set of possible instruction is restricted, but well-suited for binary fields.
- Different processing unit in the CPU $\Rightarrow$ different behaviour.
  On the Core-2, faster than the 64-bit ALU (!).
- Considerable speed improvements for binary fields.

Prime fields

There are other tricks for prime fields.

It is (or may be) wise to have, for instance :

- Code for $\mathbb{F}_p$ where p fits in n machine words, for $n = 1, 2, \ldots$.
- Code for $\mathbb{F}_p$ in Montgomery representation ;
- Code for $\mathbb{F}_p$ where p fits in 1.5 machine word ;
- Code for $\mathbb{F}_p$ where p fits in half a machine `double`...

The ultimate goal is execution speed. There are many possible optimizations to explore.

One size does not fit all

Note that even when restricting to only one finite field, there is NO *one-size-fits-all* implementation.

The most important benchmark is the user's application !

Depending on the balance between operations, not always the same code will be the best.

1. Introduction
2. What’s inside
3. Typical optimizations
4. **Results**

Current state

- $\mathtt{mp}\mathbb{F}_q$ already contains some optimizations, but there's a lot more to do.
- Timings are more up-to-date here than in the paper.
- We give results for multiplication only.

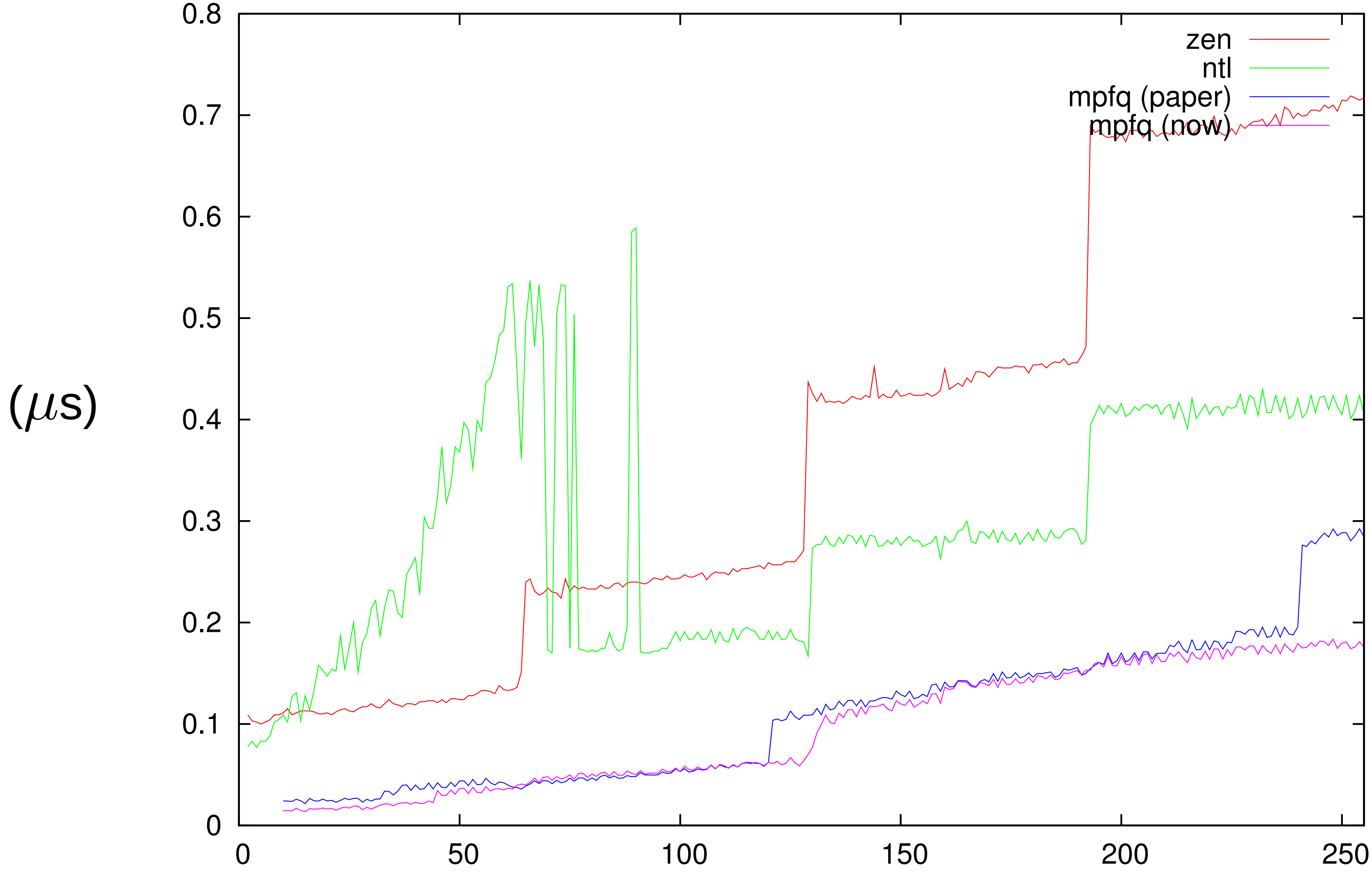
Multiplication in $\mathbb{F}_p$

(everything in ns, Intel Core2 2.667GHz)

	NTL	ZEN	ZENmgy	$\mathtt{mp}\mathbb{F}_q$	$\mathtt{mp}\mathbb{F}_q$mgy
1 word	110	52	60	74	17
2 words	140	280	120	120	32
$2^{127} - 735$				19	
3 words	210	400	170	190	58
4 words	270	550	250	260	97
$2^{255} - 19$				53	

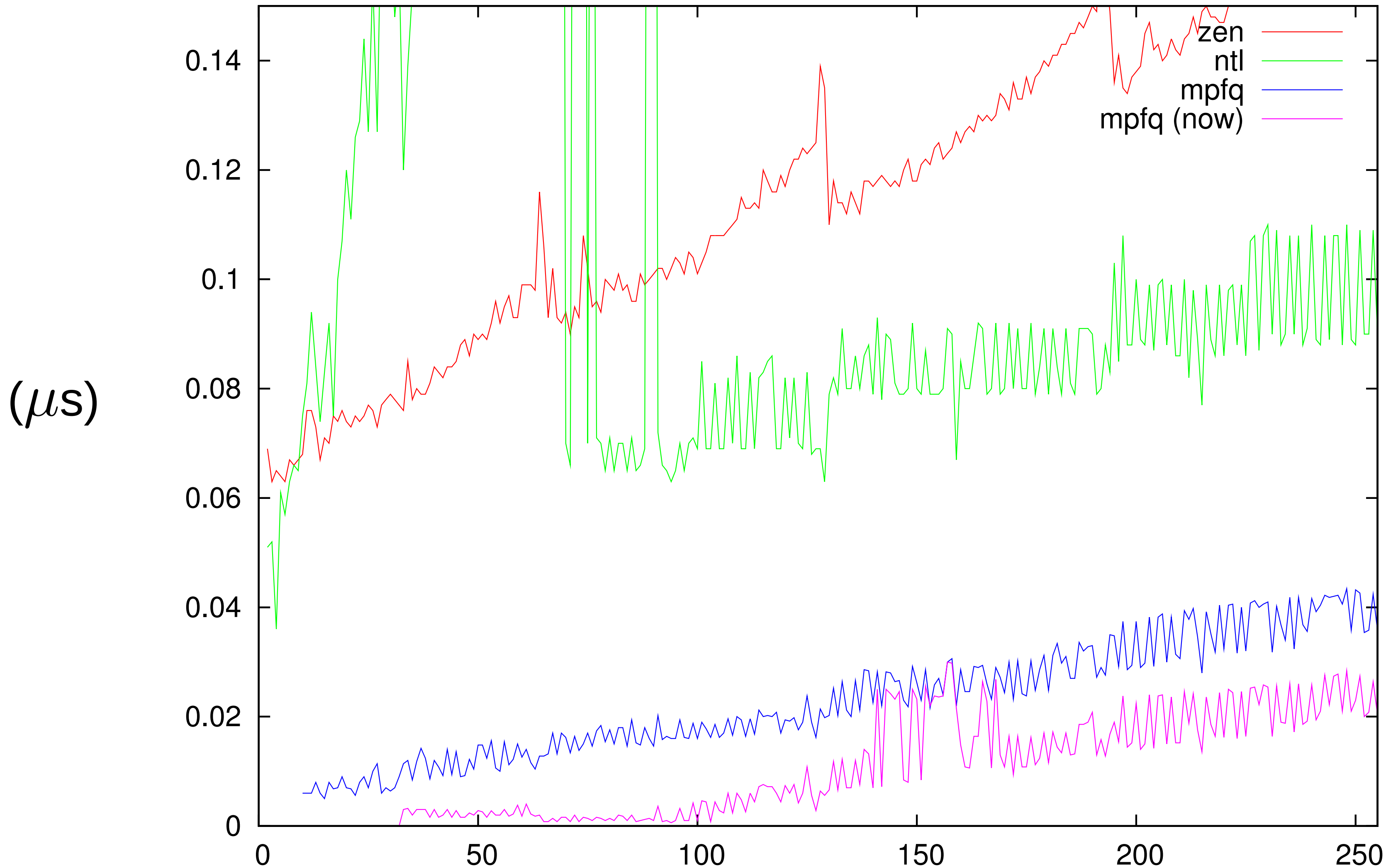
CACAO

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



CACAO

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



CACAO

Code

- The code generator works satisfyingly, but there is room for improvement.
- Some road ahead before distribution (LGPL) :
  - more documentation
  - unification ; at least I/O is a complete mess.
- generated files are already available on request.

  Do ask for one if you're interested ; feedback is most welcome.

CACAO