Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets

Diego F. Aranha
Department of Computer Science
University of Brasília

Joint work with

Julio López and Darrel Hankerson
and Francisco Rodríguez-Henríquez

Binary fields ($\mathbb{F}_{2^m}$) are omnipresent in Cryptography:

- Efficient Curve-based Cryptography (ECC, PBC)
- Post-quantum Cryptography
- Symmetric ciphers

Many algorithms/optimizations already described in the literature:

- Is it possible to unify the fastest ones in a simple formulation?
- Can such a formulation reflect the state-of-the-art **and** provide new ideas?

Objective

Contributions

- Formulation of state-of-the-art binary field arithmetic using vector instructions
- New strategy for the implementation of multiplication
- Side-channel resistance
- Time-memory trade-offs to compensate for native multiplier
- Experimental results

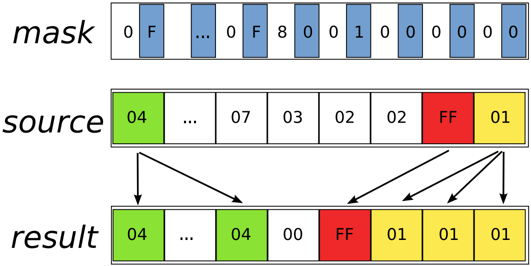
Intel Core architecture:

- 128-bit *Streaming SIMD Extensions* instruction set
- *Super shuffle engine* introduced in 45 nm series

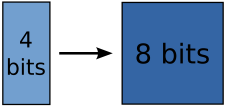
Relevant vector instructions:

Instruction	Description	Cost	Mnemonic
MOVDQA	Memory load/store	2.5	$\leftarrow$
PSLLQ, PSRLQ	64-bit bitwise shifts	1	$\ll_{\nmid 8}, \gg_{\nmid 8}$
PXOR,PAND,POR	Bitwise XOR,AND,OR	1	$\oplus, \wedge, \vee$
PUNPCKLBW/HBW	Byte interleaving	3	*interlo/hi*
PSLLDQ,PSRLDQ	128-bit bytewise shift	2 (1)	$\ll_8, \gg_8$
PSHUFB	Byte shuffling	3 (1)	*shuffle,lookup*
PALIGNR	Memory alignment	2 (1)	$\lhd$

PSHUFB instruction (_mm_shuffle_epi8):

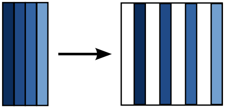


Real power: We can implement **in parallel** any function:



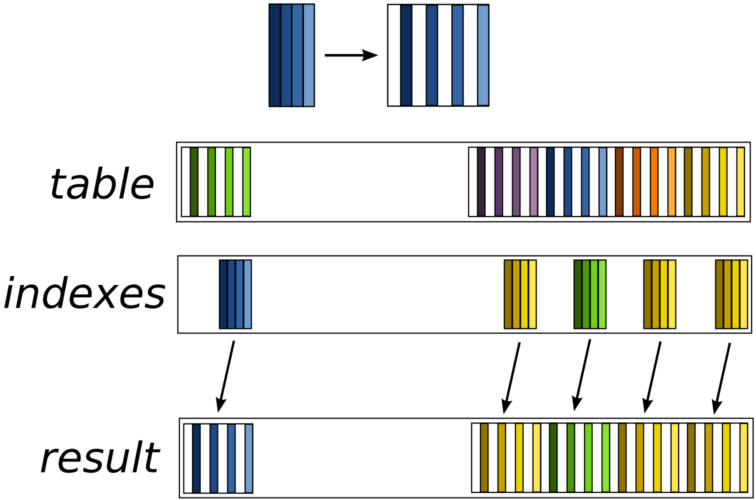
New SSSE3 instructions

Example: Bit manipulation



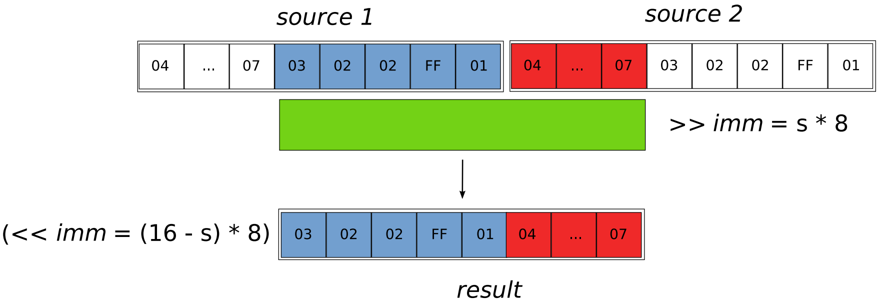
New SSSE3 instructions

Example: Bit manipulation



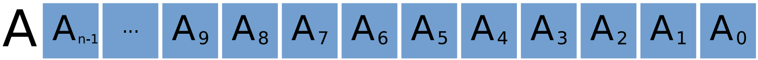
New SSSE3 instructions

PALIGNR instruction (_mm_alignr_epi8):



Binary field $\mathbb{F}_{2^m}$

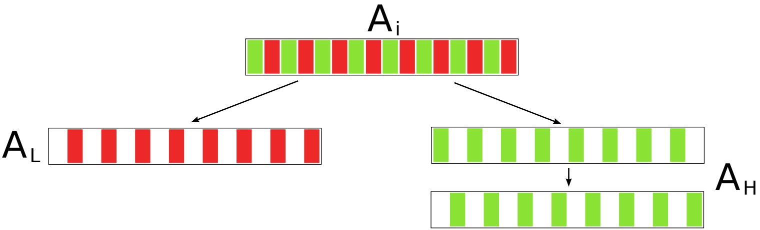
- Irreducible polynomial: $f(z)$ (trinomial or pentanomial)
- Polynomial basis: $a(z) \in \mathbb{F}_{2^m} = \sum_{i=0}^{m-1} a_i z^i$.
- Software representation: vector of $n = \lceil m/64 \rceil$ words.
- Graphical representation:



Proposed representation

To employ 4-bit granular arithmetic, convert to *split form*:

$$a_L = \sum_{\substack{0 \le i < m, \\ 0 \le i \bmod 8 \le 3}} a_i z^i, \quad a_H = \sum_{\substack{0 \le i < m, \\ 4 \le i \bmod 8 \le 7}} a_i z^{i-4},$$



Proposed representation

Easy to convert to split form:

$$A_L = A_i \wedge \texttt{0x0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F}$$
$$A_H = (A_i \wedge \texttt{0xF0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0}) >> 4$$

Easy to convert back:

$$a(z) = a_H(z)z^4 + a_L(z).$$

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

$$a(z) = \sum_{i=0}^{m} a_i z^i = a_{m-1} + \cdots + a_2 z^2 + a_1 z + a_0$$

$$a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i} = a_{m-1} z^{2m-2} + \cdots + a_2 z^4 + a_1 z^2 + a_0$$

Example:

$$a(z) = (a_{m-1}, a_{m-2}, \ldots, a_2, a_1, a_0)$$

$$a(z)^2 = (a_{m-1}, 0, a_{m-2}, 0, \ldots, 0, a_2, 0, a_1, 0, a_0)$$

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

Since squaring is a linear operation:

$$a(z)^2 = a_H(z)^2 \cdot z^8 + a_L(z)^2.$$

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

Since squaring is a linear operation:

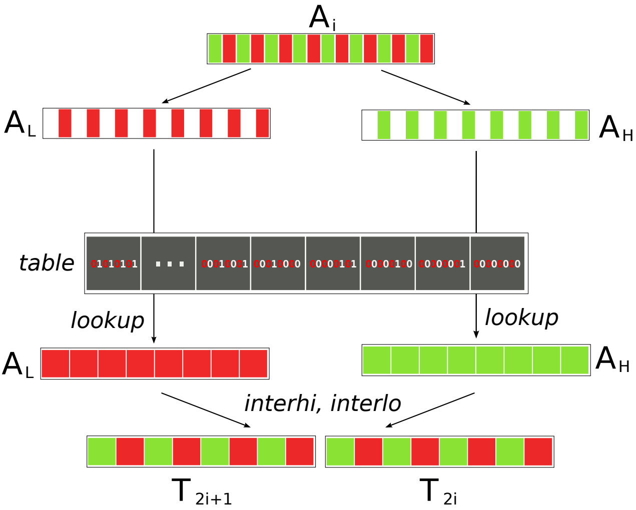
$$a(z)^2 = a_H(z)^2 \cdot z^8 + a_L(z)^2.$$

We can compute $a_L(z)^2$ and $a_H(z)^2$ with a lookup table.

For $u = (u_3, u_2, u_1, u_0)$, use $table(u) = (0, u_3, 0, u_2, 0, u_1, 0, u_0)$:

	0	1	2	3	4	5	6	7
	00000000	00000001	00000100	00000101	00010000	00010001	00010100	00010101
table	8	9	10	11	12	13	14	15
	01000000	01000001	01000100	01000101	01010000	01010001	01010100	01010101

Proposed squaring in $\mathbb{F}_{2^m}$



$$a(z)^2 = a_L(z)^2 + a_H(z)^2 \cdot z^8.$$

Square root extraction in $\mathbb{F}_{2^m}$

Algorithm by Fong et al.:

$$\sqrt{a(z)} = a_{\text{even}}(z) + \sqrt{z} \cdot a_{\text{odd}}(z)$$

Square root extraction in $\mathbb{F}_{2^m}$

Algorithm by Fong et al.:

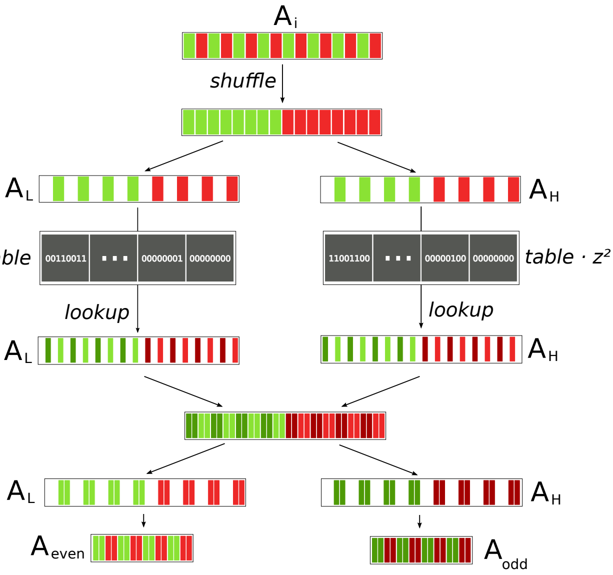
$$\sqrt{a(z)} = a_{\text{even}}(z) + \sqrt{z} \cdot a_{\text{odd}}(z)$$

Since square-root is also a linear operation:

$$\begin{aligned} \sqrt{a(z)} &= \sqrt{a_H(z)z^4 + a_L(z)} \\ &= \sqrt{a_H(z)}z^2 + \sqrt{a_L(z)} \\ &= \sqrt{z} \cdot (a_{L_{odd}}(z) + a_{H_{odd}}(z)z^2) + a_{L_{even}}(z) + a_{H_{even}}(z)z^2 \end{aligned}$$

Note: Multiplication by $\sqrt{z}$ ideally requires shifted additions only.
If not possible, precompute product by $\sqrt{z}$.

Proposed square root in $\mathbb{F}_{2^m}$

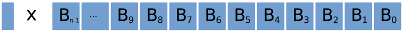


$$\sqrt{a(z)} = \sqrt{z} \cdot (a_{L_{odd}}(z) + a_{H_{odd}}(z)z^2) + a_{L_{even}}(z) + a_{H_{even}}(z)z^2$$

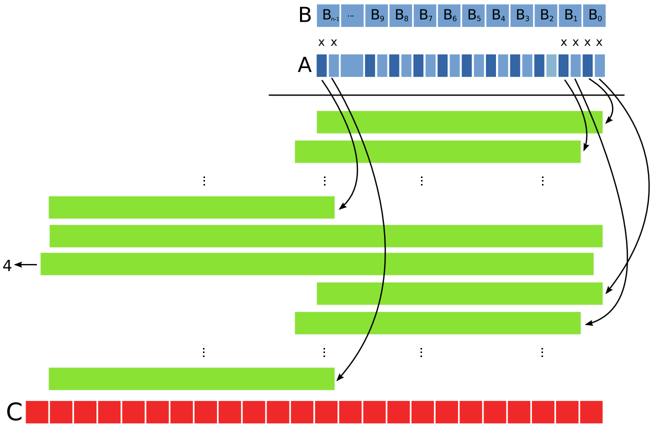
Multiplication in $\mathbb{F}_{2^m}$

1. Three strategies:
   - López-Dahab *comb* method
   - Shuffle-based multiplication
   - Native multiplication

We can compute $u \cdot b(z)$ using shifts and additions.



If $a(z)$ is divided into 4-bit polynomials, compute $a(z) \cdot b(z)$ by:



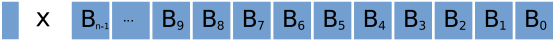
López-Dahab multiplication in $\mathbb{F}_{2^m}$

If the multiplier is represented in split form:

$$
\begin{aligned}
a(z) \cdot b(z) &= b(z) \cdot (a_H(z)z^4 + a_L(z)) \\
&= b(z)z^4 a_H(z) + b(z)a_L(z)
\end{aligned}
$$

This is a well-known technique for removing expensive 4-bit shifts!

Note: The core operation is accumulating $u \times$ dense $b(z)$.



López-Dahab multiplication in $\mathbb{F}_{2^m}$

Algorithm 1 LD multiplication implemented with n 128-bit registers.

Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..n-1]$.
Note: m_i denotes the vector of $\frac{n}{2}$ 128-bit registers $(r_{(i-1+n/2)}, \ldots, r_i)$.

Compute $T_0(u) = u(z) \cdot b(z)$, $T_1(u) = u(z) \cdot (b(z)z^4)$ for all $u(z)$ of degree < 4.
$(r_{n-1} \ldots, r_0) \leftarrow 0$
for $k \leftarrow 56$ **downto** 0 **by** 8 **do**
  for $j \leftarrow 1$ **to** $n-1$ **by** 2 **do**
    Let $u = (u_3, u_2, u_1, u_0)$, where u_t is bit $(k+t)$ of $a[j]$.
    Let $v = (v_3, v_2, v_1, v_0)$, where v_t is bit $(k+t+4)$ of $a[j]$.
    $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_0(u)$, $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_1(v)$
  end for
  $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
end for
for $k \leftarrow 56$ **downto** 0 **by** 8 **do**
  for $j \leftarrow 0$ **to** $n-2$ **by** 2 **do**
    Let $u = (u_3, u_2, u_1, u_0)$, where u_t is bit $(k+t)$ of $a[j]$.
    Let $v = (v_3, v_2, v_1, v_0)$, where v_t is bit $(k+t+4)$ of $a[j]$.
    $m_{j/2} \leftarrow m_{j/2} \oplus T_0(u)$, $m_{j/2} \leftarrow m_{j/2} \oplus T_1(v)$
  end for
  if $k > 0$ **then** $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
end for
return $c = (r_{n-1} \ldots, r_0) \bmod f(z)$

Shuffle-based multiplication in $\mathbb{F}_{2^m}$

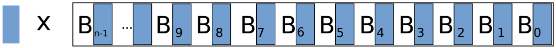
If both multiplicand and multiplier are represented in split form:

$$a(z) \cdot b(z) = (b_H(z)z^4 + b_L(z)) \cdot (a_H(z)z^4 + a_L(z))$$

Using Karatsuba formula, we can reduce it to 3 multiplications:

$$a(z) \cdot b(z) = a_H b_H z^8 + [(a_H + a_L)(b_H + b_L) + a_H b_H + a_L b_L] \, z^4 + a_L b_L$$

Note: The core operation is accumulating $u \times$ sparse $b_{L,H}(z)$.



Shuffle-based multiplication in $\mathbb{F}_{2^m}$

Algorithm 2 Multiplication in split form.

Input: Operands a, b in split representation.
Output: Result $a \cdot b$ stored in registers $(r_{n-1} \ldots, r_0)$.

```
◇ table stores all products of 4-bit × 4-bit polynomials.
(r_{n-1} ..., r_0) ← 0
for k ← 56 downto 0 by 8 do
    for j ← 1 to n − 1 by 2 do
        Let u = (u_3, u_2, u_1, u_0), where u_t is bit (k + t) of a[j].
        for i ← 0 to n/2 − 1 do r_i ← r_i ⊕ shuffle(table[u], b[i])
    end for
    (r_{n-1} ..., r_0) ← (r_{n-1} ..., r_0) ◁ 8
end for
for k ← 56 downto 0 by 8 do
    for j ← 0 to n − 2 by 2 do
        Let u = (u_3, u_2, u_1, u_0), where u_t is bit (k + t) of a[j].
        for i ← 0 to n/2 − 1 do r_i ← r_i ⊕ shuffle(table[u], b[i])
    end for
    if k > 0 then (r_{n-1} ..., r_0) ← (r_{n-1} ..., r_0) ◁ 8
end for
```

Native multiplication

Guidelines:

- As memory access is expensive, do work on registers.
- To minimize number of registers, use 128-bit granularity.
- Use Karatsuba for each 128×128-bit multiplication.
- Use maximum number of Karatsuba levels for $\lceil \frac{n}{2} \rceil$ digits.

Comparison

López-Dahab multiplication:

- Explores highest-granularity XOR operation
- Consumes memory space proportional to field size

Shuffle-based multiplication:

- Relies on sparser core operation
- Consumes constant memory space (apart from Karatsuba)
- Depends on constants stored in memory

Native multiplication:

- Faster and with constant memory consumption.
- No widespread support.

Modular reduction

Requires heavy shifting, so split representation does not help.

Some guidelines:

- If $f(z)$ is a trinomial, implement with vector digits
- If $f(z)$ is a pentanomial, process pairs of digits in parallel or in 64-bit mode
- Accumulate writes into registers before writing to memory
- Reduce squaring/multiplication results in registers

Half-trace

We want to compute $H(c) = \sum_{i=0}^{(m-1)/2} c^{2^{2i}}$.

Important: For even i, $H(z^i) = H(z^{i/2}) + z^{i/2} + Tr(z^i)$.

Algorithm 3 Solve $x^2 + x = c$

Input: $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$ where m is odd and $Tr(c) = 0$
Output: a solution s of $x^2 + x = c$.

1: Compute $H(l_0 c^{8i+1} + l_1 c^{8i+3} + l_2 c^{8i+5} + l_3 c^{8i+7})$ for $0 \leq i \leq \lfloor \frac{m-3}{8} \rfloor$ and $l_j \in \mathbb{F}_2$.
2: $s \leftarrow 0$
3: **for** $i = (m-1)/2$ **downto** 1 **do**
4:     **if** $c_{2i} = 1$ **then**
5:         $c \leftarrow c + z^i$, $s \leftarrow s + z^i$
6:     **end if**
7: **end for**
8: **return** $s + \sum_{i \in I} c^{8i+1} H(z^{8i+1}) + c^{8i+3} H(z^{8i+3}) + c^{8i+5} H(z^{8i+5}) + c^{8i+7} H(z^{8i+7})$

Fixed 2^k power [Bos et al.]

Precompute a table T of $16\lceil \frac{m}{4} \rceil$ field elements such that

$$T[j, i_0 + 2i_1 + 4i_2 + 8i_3] = (i_0 z^{4j} + i_1 z^{4j+1} + i_2 z^{4j+2} + i_3 z^{4j+3})^{2^k}$$

Then we can compute a^{2^k} as:

$$\sum_{j=0}^{\lceil \frac{m}{4} \rceil} T[j, \lfloor a/2^{4j} \rfloor \bmod 2^4].$$

Inversion

Guidelines:

- If memory is not available, implement Extended Euclidean Algorithm in 64-bit mode.
- If memory is available, implement Itoh-Tsuji with precomputed 2^i powers:

$$a^{-1} = a^{(2^{m-1}-1)2}$$

Implementation

Material:

- GCC 4.1.2 (fastest SSE intrinsics, GCC 4.5.0 is good again)
- RELIC cryptographic library[1]
- Intel Core 2 65,45nm processors and Intel Core i7

Parameters:

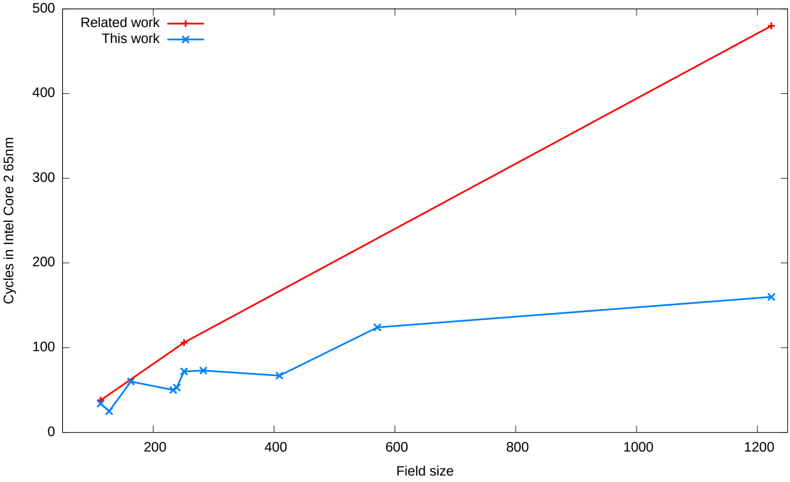
- 16 different binary fields ranging from 113 to 1223 bits
- Choices of square-root friendly and standard $f(z)$
- Elliptic curves over 6 of these fields

Comparison:

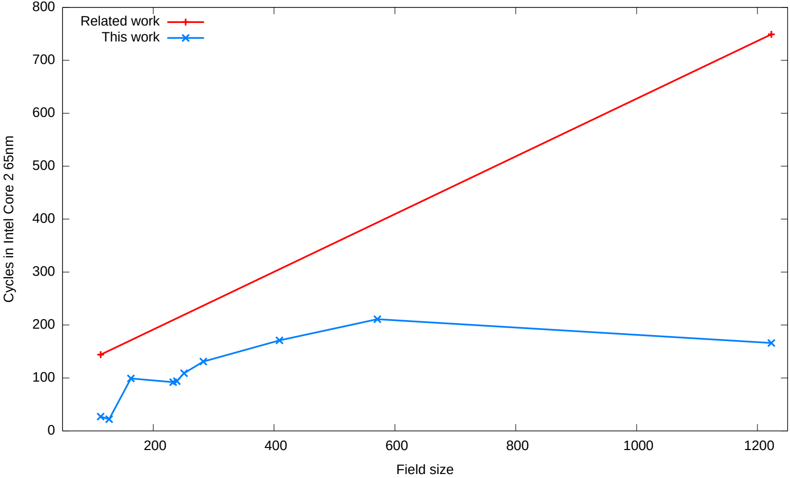
- Only vector implementations (**mp**$\mathbb{F}_q$, Beuchat et al. 2009)
- Only in entry-level Intel Core 2 65 nm (more on the paper)

[1] `http://code.google.com/p/relic-toolkit/`

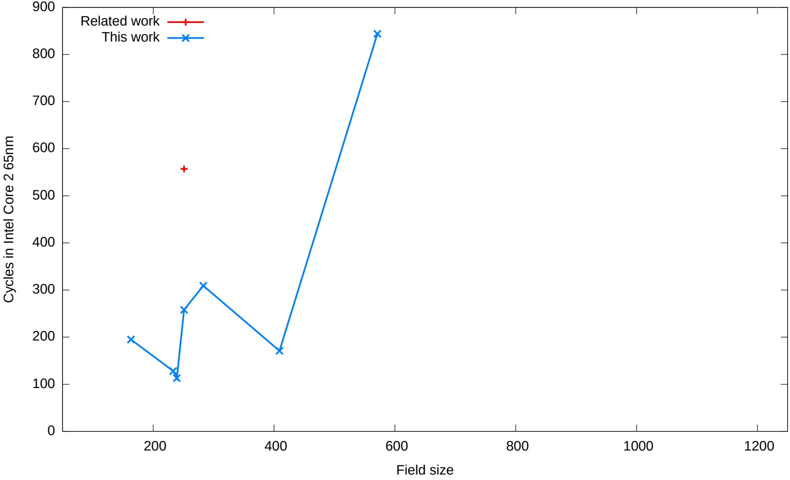
Experimental results – Squaring



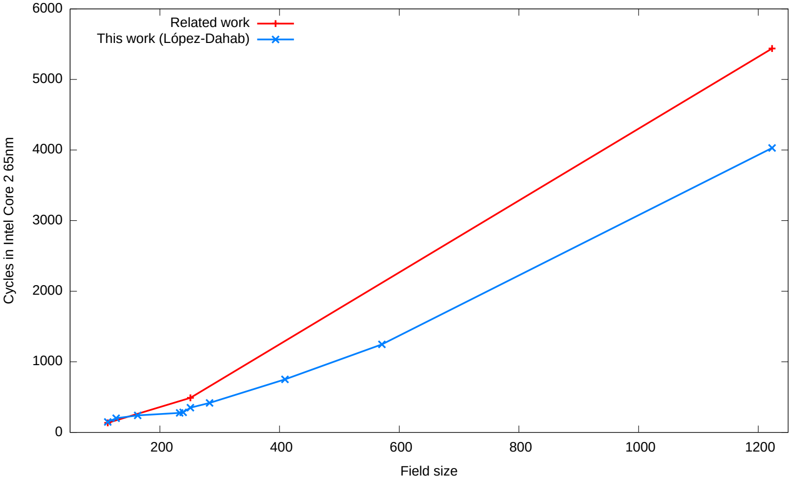
Experimental results – Square-root with friendly $f(z)$



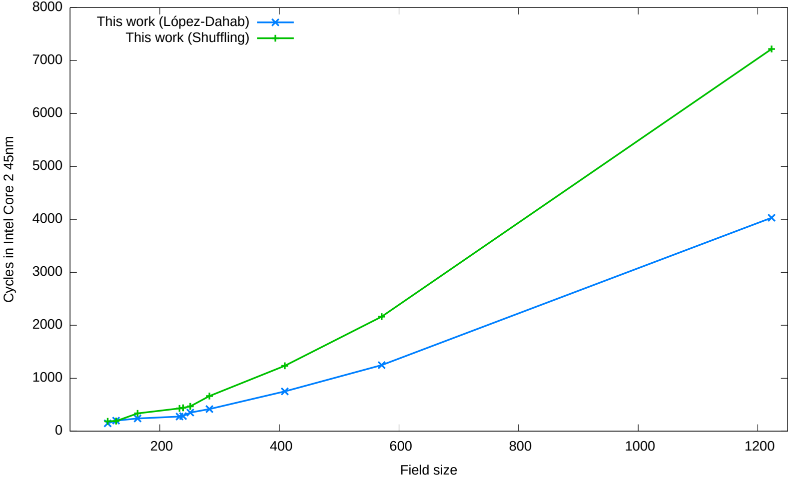
Experimental results – Square-root with standard $f(z)$



Experimental results – López-Dahab multiplication



Experimental results – Shuffle-based multiplication



Note: Native multiplier on newer machines is twice faster than LD.

Observations

Squaring and square-root are:

- Efficiently formulated with M/S ratio up to 34
- Faster when shuffling throughput is higher
- Heavily dependent on the choice of $f(z)$

Shuffle-based multiplication:

- Has a bottleneck with constants stored in memory
- Requires faster table addressing scheme
- Is only 50%-90% slower than López-Dahab!

Other operations:

- Restore the ratio to native multiplication ($H \approx M, I \approx 25M$).

Experimental results – Elliptic curve arithmetic

Table: Timings given in 10^3 cycles for elliptic curve operations.

	Point multiplication (kP)
Curve	Core 2 65nm
CURVE2251 - Core 2	594
CURVE2251 - CLMUL	282
CURVE2251 - CLMUL + AVX	225
	Related work for $E(\mathbb{F}_{2^{251}})$
BBE (Bernstein) - Core 2	314
eBACS ($\mathbf{mp}\mathbb{F}_q$) - Core 2	855

Conclusions

New formulation and implementation of binary field arithmetic:

- Follows trend of faster shuffle instructions
- Improve results from related work by 8%-84%
- Induces a new implementation strategy for multiplication
- Still requires architectural features to be optimal
- May be cheaper to support than a full native multiplier

Timings for non-batched arithmetic on binary elliptic curves:

- Provide new speed record for side-channel resistant scalar multiplication on binary curves
- Improve results for kP on eBACS by at least 27%-30%

Thank you for your attention!
Any questions?