Software implementation of binary elliptic curves: impact of the carry-less multiplier on scalar multiplication

Jonathan Taverne[1]*, Armando Faz-Hernández[2], Diego F. Aranha[3]**, Francisco Rodríguez-Henríquez[2], Darrel Hankerson[4], and Julio López[3]

[1] Université Claude Bernard Lyon 1
43 Bld. 11 Novembre 1918, 69622 Villeurbanne, France
jonathan.taverne@etu.univ-lyon1.fr,
[2] Computer Science Department, CINVESTAV-IPN,
Av. Instituto Politécnico Nacional No. 2508, 07300 México City, México.
armfaz@computacion.cs.cinvestav.mx, francisco@cs.cinvestav.mx
[3] Institute of Computing, University of Campinas
Av. Albert Einstein, 1251, CEP 13084-971, Campinas, Brazil
dfaranha@ic.unicamp.br, jlopez@ic.unicamp.br
[4] Auburn University,
221 Parker Hall Auburn, AL 36849-5307, USA
hankedr@auburn.edu

Abstract. The availability of a new carry-less multiplication instruction in the latest Intel desktop processors significantly accelerates multiplication in binary fields and hence presents the opportunity for reevaluating algorithms for binary field arithmetic and scalar multiplication over elliptic curves. We describe how to best employ this instruction in field multiplication and the effect on performance of doubling and halving operations. Alternate strategies for implementing inversion and half-trace are examined that restore most of their competitiveness relative to the new multiplier. These improvements in field arithmetic are complemented by a study on serial and parallel approaches for Koblitz and random curves, where parallelization strategies are implemented and compared. The contributions are illustrated with experimental results improving the state-of-the-art performance of halving and doubling-based scalar multiplication on NIST curves at the 112- and 192-bit security levels, and a new speed record for side-channel resistant scalar multiplication in a random curve at the 128-bit security level.

Key words: Elliptic curve cryptography, finite field arithmetic, parallel algorithm.

1 Introduction

Improvements in the fabrication process of microprocessors allow the resulting higher transistor density to be converted into architectural features such as inclusion of new instructions or faster execution of the current instruction set. Limits on the conventional ways of increasing a processor's performance such as incrementing the clock rate, scaling the memory hierarchy [38] or improving support for instruction-level parallelism (ILP) [37] have pushed manufacturers to embrace parallel processing as the mainstream computing paradigm and consequently amplify support for computing resources such as multiprocessing and vectorization. Examples of the latter are the recent inclusions of the SSE4 [22], AES [19] and AVX [14] instruction sets in the latest Intel microarchitectures.

Since the dawn of elliptic curve cryptography in 1985, several field arithmetic assumptions have been made by researchers and designers regarding its efficient implementation in software platforms. Some analysis (supported by experiments) assumed that inversion to multiplication ratios (I/M) were

* This work was performed while the author was visiting CINVESTAV-IPN.
** A portion of this work was performed while the author was visiting University of Waterloo.

sufficiently small (e.g., $I/M \approx 3$) that point operations would be done in affine coordinates, favoring certain techniques. However, the small ratios were a mix of old hardware designs, slower multiplication algorithms compared with [32], and composite extension degree, and it seems clear that the ratio is sufficiently large that there will be incentive to use projective coordinate operations. Our interest in the face of much faster multiplication is at the other end—is I/M large enough to affect methods that commonly assumed this ratio is modest?

On the other hand, authors in [16] considered that the equivalent cost of a point halving computation was roughly equal to 2 field multiplications. We recall that the expensive computations in halving are a field multiplication, solving a quadratic $z^2 + z = c$, and finding a square root over $\mathbb{F}_{2^m}$. However, quadratic solver algorithms presented in [21] are multiplication-free and hence, provided that a fast binary field multiplier is available, there would be concern that the ratio of point halving to multiplication may be much larger than 2. Having a particularly fast multiplier would also push for computing square roots in $\mathbb{F}_{2^m}$ as efficiently as possible.

There exist several other folklore assumptions. Cryptographic software designers frequently assume that the cost of binary field squaring computations can be safely neglected when compared with the more expensive field multiplication operation. But once again, it is conceivable that if in modern software platforms a much faster multiplier is available, then the notion that a field squaring operation is essentially free might not be valid anymore. A prevalent assumption is that large-characteristic fields are faster than binary field counterparts for software implementations of elliptic curve cryptography.[5] In spite of simpler arithmetic, binary field realizations could not be faster than large-characteristic analogues mostly due to the absence of a native *carry-less multiplier* in contemporary high-performance processors. However, using a bit-slicing technique, Bernstein [6] was able to compute a batch of 251-bit scalar multiplications on a binary Edwards curve, employing 314,323 clock cycles per scalar multiplication, which, before the results presented in this work and to the best of our knowledge, was the fastest reported time for a software implementation of binary elliptic point multiplication.

In this work, we evaluate the impact of the recently introduced carry-less multiplication instruction [20] in the performance of binary field arithmetic and scalar multiplication over elliptic curves. We also consider parallel strategies in order to speed scalar multiplication when working on multi-core architectures. In contrast to parallelization applied to a batch of operations, the approach considered here applies to a single point multiplication. These approaches target different environments: batching makes sense when throughput is the measurement of interest, while the lower level parallelization is of interest when latency matters and the device is perhaps weak but has multiple processing units.

As the experimental results will show, our implementation of multiplication via this native support was significantly faster than previous timings reported in the literature. This motivated a study on alternative implementations of binary field arithmetic in hope of restoring the performance ratios among different operations in which the literature is traditionally based [21]. A direct consequence of this study is that performance analysis based on these conventional ratios [5] will remain valid in the new platform. Our main contributions are:

- A strategy to efficiently employ the native carry-less multiplier in binary field multiplication.
- Branchless and/or vectorized approaches for implementing half-trace computation, integer recoding and inversion. These approaches allow the halving operation to become again competitive with doubling in the face of a significantly faster multiplier and help to reduce the impact of in-

[5] In hardware realizations, the opposite thesis is widely accepted: elliptic curve point multiplication can be computed (much) faster using binary extension fields.

teger recoding and inversion in the overall speed of scalar multiplication, even when projective coordinates are used.

– Parallelization strategies for dual core execution of scalar multiplication algorithms in random and Koblitz binary elliptic curves.

We obtain a new state-of-the-art implementation of arithmetic in binary elliptic curves, including improved performance for NIST-standardized random curves and Koblitz curves suitable for halving and a new speed record for point multiplication in a random curve at the 128-bit security level.

This paper reports a software library that performs scalar multiplication over the NIST Koblitz and random binary curves K-233, B-233, K-409 and B-409 in 83, 166, 302 and 656 thousand cycles, respectively, on a single core of a 3.326 GHz Intel i5 660 processor. Our library was also adjusted to compute scalar multiplication in a 251-bit binary Edwards curve in 264 thousand clock cycles. Moreover, we also introduce parallel strategies that accelerate scalar multiplication by using both cores present in the i5 processor. In this setting, we are able to compute scalar multiplication on K-233, B-233, K-409 and B-409 in 56, 111, 181 and 429 thousand cycles, respectively. These dual core timings imply, for example, that scalar multiplication on K-233 can be computed in less than 17μs.

The remainder of the paper progresses as follows. Section 2 elaborates on exploiting carry-less multiplication for high-performance field multiplication along with implementation strategies for half-trace and inversion. Sections 3 and 4 discuss serial and parallel approaches for scalar multiplication. Section 5 presents extensive experimental results and comparison with related work. Section 6 concludes the paper with perspectives on the interplay between the proposed implementation strategies and future enhancements in the architecture under consideration.

2 Binary field arithmetic

A binary extension field $\mathbb{F}_{2^m}$ can be generated by means of a degree-m polynomial f irreducible over $\mathbb{F}_2$ as $\mathbb{F}_{2^m} \cong \mathbb{F}_2[z]/\left(f(z)\right)$. In the case of software implementations in modern desktop platforms, field elements $a \in \mathbb{F}_{2^m}$ can be represented as polynomials of degree at most $m-1$ with binary coefficients a_i packed in n_{64} 64-bit processor words. In this context, the recently introduced carry-less multiplication instruction can play a significant role in order to efficiently implement a multiplier. Along with field multiplication, other relevant field arithmetic operations such as squaring, square root, and half-trace, will be discussed in the rest of this section.

2.1 Multiplication

Field multiplication is the performance-critical operation for implementing several cryptographic primitives relying on binary fields, including arithmetic over elliptic curves and the Galois Counter Mode of operation (GCM). For accelerating the latter when used in combination with the AES block cipher [19], Intel introduced the carry-less multiplier in the 32nm Nehalem microarchitecture as an instruction operating on 64-bit words stored in 128-bit vector registers with opcode *pclmulqdq* [20]. The instruction latency currently peaks at 12 cycles while reciprocal throughput ranks at 8 cycles. In other words, when the instruction operands are not in a dependency chain, effective latency is 8 cycles [15].

The instruction certainly looks expensive when compared to the 3-cycle 64-bit integer multiplier present in the same platform, which raises speculation whether Intel aimed for an area/performance trade-off or simply balanced the latency to the point where the carry-less multiplier did not interfere with the throughput of the hardware AES implementation. Either way, the instruction features suggest the following empirical guidelines for organizing the field multiplication code: (i) as memory access by vector instructions continues to be expensive [6], the maximum amount of work should be done in registers, for example through a Comba organization [12]; (ii) as the number of registers employed in multiplication should be minimized for avoiding false dependencies and maximize throughput, the multiplier should have 128-bit granularity; (iii) as the instruction latency allows, each 128-bit multiplication can be implemented with three carry-less multiplications in a Karatsuba fashion [25].

In fact, the overhead of Karatsuba multiplication is minimal in binary fields and the Karatsuba formula with the smaller number of multiplications for multiplying $\lceil \frac{n_{64}}{2} \rceil$ 128-bit digits proved to be optimal in all the considered field sizes. This observation comes in direct contrast to previous vectorized implementations of the *comb* method for binary field multiplication due to López and Dahab [32, Algorithm 5], where the memory-bound precomputation step severely limits the number of Karatsuba steps which can be employed, fixing the cutoff point to large fields [2] such as $\mathbb{F}_{2^{1223}}$. To summarize, multiplication was implemented as a 128-bit granular Karatsuba multiplier with each 128-digit multiplication solved by another Karatsuba instance requiring 3 carry-less multiplications, cheap additions and efficient shifts by multiples of 8 bits. A single 128-digit level of Karatsuba was used for fields $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{251}}$ where $\lceil \frac{n_{64}}{2} \rceil = 2$, while two instances were used for field $\mathbb{F}_{2^{409}}$ where $\lceil \frac{n_{64}}{2} \rceil = 4$. Particular approaches which led to lower performance in our experiments were organizations based on optimal Toom-Cook [10] due to the higher overhead brought by minor operations; and on a lower 64-bit granularity combined with alternative multiple-term Karatsuba formulas [33] due to register exhaustion to store all the intermediate values, causing a reduction in overall throughput.

2.2 Squaring, square-root and multi-squaring

Squaring and square-root are considered cheap operations in a binary field, especially when $\mathbb{F}_{2^m}$ is defined by a square-root friendly polynomial [3,1], because they only require linear manipulation of individual binary coefficients [21]. These operations are traditionally implemented with the help of large precomputed tables, but vectorized implementations are possible with simultaneous table lookups through byte shuffling instructions [2]. This approach is enough to keep square and square-root efficient relative to multiplication even with a dramatic acceleration of field multiplication. For illustration, [2] reports multiplication-to-squaring ratios as high as 34 without a native multiplier, far from the conventional ratios of 5 [5] or 7 [21] and with a large room for future improvement.

Multi-squaring, or exponentiation to 2^k, can be efficiently implemented with a time-memory trade-off proposed as m-squaring in [1,11] and here referred as *multi-squaring*. For a fixed k, a table T of $16\lceil \frac{m}{4} \rceil$ field elements can be precomputed such that:

$$\begin{aligned} T[j, i_0 + 2i_1 + 4i_2 + 8i_3] &= (i_0 z^{4j} + i_1 z^{4j+1} + i_2 z^{4j+2} + i_3 z^{4j+3})^{2^k} \\ a^{2^k} &= \sum_{j=0}^{\lceil \frac{m}{4} \rceil} T[j, \lfloor a/2^{4j} \rfloor \bmod 2^4]. \end{aligned}$$

The threshold where multi-squaring became faster than simple consecutive squaring observed in our implementation was around $k \geq 6$ for $\mathbb{F}_{2^{233}}$ and $k \geq 10$ for $\mathbb{F}_{2^{409}}$.

2.3 Inversion

Inversion modulo $f(z)$ can be implemented via the polynomial version of the Extended Euclidean Algorithm (EEA), but the frequent branching and recurrent shifts by arbitrary amounts present a performance obstacle for vectorized implementations, which makes it difficult to write consistently fast EEA codes across different platforms. A branchless approach can be implemented through Itoh-Tsuji inversion [23] by computing $a^{-1} = a^{(2^{m-1}-1)2}$, as proposed in [18]. In contrast to the EEA method, the Itoh-Tsujii approach has the additional merit of being similarly fast (relative to multiplication) across common processors.

The overall cost of the method is $m - 1$ squarings and a number of multiplications dictated by the length of an addition chain for $m - 1$. The cost of squarings can be reduced by computing each required 2^i-power as a multi-squaring [11]. The choice of an addition chain allows the implementer to control the amount of required multiplications and the precomputed storage for multi-squaring, since the number of 2^i-powers involved can be balanced.

Previous work obtained inversion-to-multiplication ratios between 22 and 41 by implementing EEA in 64-bit mode [2], while the conventional ratios are between 5 and 10 [21,5]. While we cannot reach the small ratios with Itoh-Tsujii for the parameters considered here, we can hope to do better than applying the method from [2] which will give significantly larger ratios with the carry-less multiplier. Hence the cost of squarings and multi-squarings should be minimized to the lowest possible allowed by storage capacity.

To summarize, we use addition chains of 10, 10 and 11 steps for computing field inversion over the fields $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{251}}$ and $\mathbb{F}_{2^{409}}$, respectively.[6] We extensively used the multi-squaring approach described in the preceding section. For example, in the case of $\mathbb{F}_{2^{233}}$, we selected the addition chain $1{\to}2{\to}3{\to}6{\to}7{\to}14{\to}28{\to}29{\to}58{\to}116{\to}232$, and used 3 pre-computed tables for computing the iterated squarings $a^{2^{29}}$, $a^{2^{58}}$ and $a^{2^{116}}$. The rest of the field squaring operations were computed by executing consecutive squarings. We recall that each table stores a total of $16\lceil \frac{m}{4} \rceil$ field elements.

2.4 Half-trace

Half-trace plays a central role in point halving and performance is essential if halving is to be competitive against doubling. For an odd integer m, the half-trace function $H : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$ is defined by $H(c) = \sum_{i=0}^{(m-1)/2} c^{2^{2i}}$. The computation allows quadratics $\lambda^2 + \lambda = c$ to be solved, required in point halving. One efficient desktop-targeted implementation of the half-trace is described in [3], making extensive use of precomputations. This implementation is based on two main steps: the elimination of even power coefficients and the accumulation of half-trace precomputed values.

Step 5 in Algorithm 1, as shown in [21], consists in reducing the number of non-zero coefficients of c by removing the coefficients of even powers i via $H(z^i) = H(z^{i/2}) + z^{i/2} + \mathrm{Tr}(z^i)$. That will lead to memory and time savings during the last step of the half-trace computation, the accumulation (step 8). This is done by extraction of the odd and even bits and can benefit from the vectorization in the same way as it was done for the computation of square-root in [2]. However in the case of half-trace there is a bottleneck caused by data dependencies. For efficiency reasons, the bank of 128-bit registers are used as much as possible, but at one point in the algorithm execution the number of available bits to process decreases. For 64-bit and 32-bit digits, the use of 128-bit registers is still beneficial, but for a smaller size, the conventional approach (not vectorized) becomes again competitive.

Once step 5 completed, the direction taken in [21] remains in reducing memory needs. However another approach is followed in [3] which does not attempt to minimize memory requirements but rather it greedily strives to speed the accumulation part (step 8). Precomputation is extended so as to reduce the number of expected accesses to the lookup table. The following values of the half-trace

[6] In the case of inversion over $\mathbb{F}_{2^{409}}$, the minimal length addition chain to reach $m - 1 = 408$ has 10 steps. However, we preferred to use an 11-step chain to save one look-up table.

are stored: $H(l_0c^{8i+1} + l_1c^{8i+3} + l_2c^{8i+5} + l_3c^{8i+7})$ for all $i \geq 0$ such that $8i + 1 < m - 2$ and $(l_0, l_1, l_2, l_3) \in \mathbb{F}_2^4$. The memory size in bytes taken by the precomputations follows the formula $16 \times n_{64} \times 8 \times \lfloor \frac{m}{8} \rfloor$ where m is the degree of the field and n_{64} is the number of 64-bit words needed to store a field element.

Algorithm 1 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 $\mathrm{Tr}(c) = 0$
Output: a solution $s$ of $x^2 + x = c$
1: precompute $H(l_0c^{8i+1} + l_1c^{8i+3} + l_2c^{8i+5} + l_3c^{8i+7})$ for $i \in I = \{0, 1, \ldots, \lfloor \frac{m-3}{8} \rfloor\}$ and $(l_0, l_1, l_2, l_3) \in \mathbb{F}_2^4$
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 $s \leftarrow s + \sum_{i \in I} c^{8i+1}H(z^{8i+1}) + c^{8i+3}H(z^{8i+3}) + c^{8i+5}H(z^{8i+5}) + c^{8i+7}H(z^{8i+7})$
```

While considering different organizations of the half trace `C` code, we made the following serendipitious observation: when we inserted as many `xor` operations as the data dependencies permitted from the accumulation stage (step 8) into the first part (step 5), where the even and odd bits are extracted, we obtained a substantial speed-up of 20% to 25% compared with a code written in the same order as described in Algorithm 1. This empirical improvement, which probably is compiler and processor depending, is hard to explain. Plausible explanations are compiler optimization and instruction pipelining within the processor. Thus we reach a half-trace-to-multiplication ratio near of 1, and this ratio can be reduced if memory can be consumed more aggressively.

3 Random binary elliptic curves

Given a finite field $\mathbb{F}_q$ for $q = 2^m$, a non-supersingular elliptic curve $E(\mathbb{F}_q)$ is defined to be the set of points $(x, y) \in GF(2^m) \times GF(2^m)$ that satisfy the affine equation

$$y^2 + xy = x^3 + ax^2 + b, \quad (1)$$

where a and $0 \neq b \in \mathbb{F}_q$, together with the point at infinity denoted by $\mathcal{O}$. It is known that $E(\mathbb{F}_q)$ forms an additive Abelian group with respect to the elliptic point addition operation.

Let k be a positive integer and P a point on an elliptic curve. Then *elliptic curve scalar multiplication* is the operation that computes the multiple $Q = kP$, defined as the point resulting of adding P to itself k times. One of the most basic methods used for computing a scalar multiplication is based on a double-and-add variant of Horner's rule. As the name suggests, the two most prominent building blocks of this method are the *point doubling* and *point addition* primitives. By using the non-adjacent form (NAF) representation of the scalar k, the addition-subtraction method computes a scalar multiplication in about m doubles and $m/3$ additions [21]. The method can be extended to a *width-ω NAF* $k = \sum_{i=0}^{t-1} k_i 2^i$ where $k_i \in \{0, \pm 1, \ldots, \pm 2^m - 1\}$, $k_{t-1} \neq 0$, and at most one of any ω consecutive digits is nonzero. The length t is at most one bit larger than the bitsize of k, and the density is approximately $1/(\omega + 1)$; for $\omega = 2$, this is NAF.

3.1 Sequential algorithms for random binary curves

The traditional left-to-right double-and-add method is illustrated in Algorithm 2 where $n = 0$ (that is, the computation corresponds to the left column) and the *width-ω NAF* $k = \sum_{i=0}^{t-1} k_i 2^i$ expression is computed from left to right, i.e., it starts processing k_{t-1} first, then k_{t-2} until it ends with the coefficient k_0. Step 1 computes $2^{\omega-2}-1$ multiples of the point P. Based on the Montgomery trick, authors in [13] suggested a method to precompute the affine points in large-characteristic fields $\mathbb{F}_p$, employing only one inversion. Exporting that approach to $\mathbb{F}_{2^m}$, we obtained formulae that offer a saving of 4 multiplications and 15 squarings for $\omega = 4$ when compared with a naive method that would make use of the Montgomery trick in a trivial way (see Table 1 for a summary of the computational effort associated to this phase).

Algorithm 2 Double-and-add, halve-and-add scalar multiplication: parallel

```
Input: $\omega$, scalar $k$, $P \in E(\mathbb{F}_{2^m})$ of odd order $r$, constant $n$ (e.g., from Table 1(b))
Output: $kP$
 1: Compute $P_i = iP$ for
    $i \in I = \{1, 3, \ldots, 2^{\omega-1}-1\}$
 2: $Q_0 \leftarrow \mathcal{O}$
    {Barrier}
 3: Recode: $k' = 2^n k \bmod r$ and obtain rep
    $\omega\mathrm{NAF}(k')/2^n = \sum_{i=0}^{t} k'_i 2^{i-n}$
 4: Initialize $Q_i \leftarrow \mathcal{O}$ for $i \in I$
 5: for $i = t$ downto $n$ do
 6:     $Q_0 \leftarrow 2Q_0$
 7:     if $k'_i > 0$ then
 8:         $Q_0 \leftarrow Q_0 + P_{k'_i}$
 9:     else if $k'_i < 0$ then
10:         $Q_0 \leftarrow Q_0 - P_{-k'_i}$
11:     end if
12: end for
13: for $i = n-1$ downto $0$ do
14:     $P \leftarrow P/2$
15:     if $k'_i > 0$ then
16:         $Q_{k'_i} \leftarrow Q_{k'_i} + P$
17:     else if $k'_i < 0$ then
18:         $Q_{-k'_i} \leftarrow Q_{-k'_i} - P$
19:     end if
20: end for
    {Barrier}
21: return $Q \leftarrow Q_0 + \sum_{i \in I} iQ_i$
```

For a given ω, the evaluation stage of the algorithm has approximately $m/(\omega+1)$ point additions, and hence increasing ω has diminishing returns. For the curves given by NIST [34] and with on-line precomputation, $\omega \leq 6$ is optimal in the sense that total point additions are minimized. In many cases, the recoding in $\omega\mathrm{NAF}(k)$ is performed on-line and can be considered as part of the precomputation step.

The most popular way to represent points in binary curves is López-Dahab projective coordinates that yield an effective cost for a mixed point addition and point doubling operation of about $8M + 5S \approx 9M$ and $4M + 5S \approx 5M$, respectively (see Tables 2 and 3). Kim and Kim [26] report alternate formulas for point doubling requiring four multiplications and five squarings, but two of the four multiplications are by the constant b, and these have the same cost as general multiplication with the native carry-less multiplier. For mixed addition, Kim and Kim require eight multiplications but save two field reductions when compared with López-Dahab, giving their method the edge. Hence, in this work we use López-Dahab for point doubling and Kim and Kim for point addition.

Right-to-left halve-and-add Scalar multiplication based on point halving replaces point doubling by a potentially faster *halving* operation that produces Q from P with $P = 2Q$. The method was proposed independently by Knudsen [28] and Schroeppel [35] for curves $y^2 + xy = x^3 + ax^2 + b$ over $\mathbb{F}_{2^m}$. The method is simpler if the trace of a is 1, and this is the only case we consider. The expensive computations in halving are a field multiplication, solving a quadratic $z^2 + z = c$, and finding a square root. On the NIST random curves studied in this work, we found that the cost of halving is approximately $3M$, where M denotes the cost of a field multiplication.

Let the base point P have odd order r, and let t be the number of bits to represent r. For $0 < n \leq t$, let $\sum_{i=0}^{t} k'_i 2^i$ be the width-ω NAF of $2^n k \bmod r$.[7] Then $k \equiv k'/2^n \equiv \sum_{i=0}^{t} k'_i 2^{i-n} \pmod{r}$ and the scalar multiplication can be split as

$$kP = (k'_t 2^{t-n} + \cdots + k'_n)P + (k'_{n-1}2^{-1} + \cdots + k'_0 2^{-n})P. \tag{2}$$

When $n = t$, this gives the usual representation for point multiplication via halving, illustrated in Algorithm 2 (that is, the computation is essentially the right column). The cost for postcomputation appears in Table 1.

3.2 Parallel scalar multiplication on random binary curves

For parallelization, choose $n < t$ in (2) and process the first portion by a double-and-add method and the second portion by a method based on halve-and-add. Algorithm 2 illustrates a parallel approach suitable for two processors. Recommended values for n to balance cost between processors appear in Table 1.

Table 1. Costs and parameter recommendations for $\omega \in \{3, 4, 5\}$.

ω	Algorithm 2		Alg 4	Alg 5
	Precomp	Postcomp	Precomp	Postcomp
3	$14M+11S+I$	$43M+26S$	$2M+3S+I$	$26M+13S$
4	$38M+15S+I$	$116M+79S$	$9M+9S+I$	$79M+45S$
5	N/A	N/A	$23M+19S+2I$	$200M+117S$

(a) Pre- and post-computation costs in Alg 2, 4 and 5.

ω	Algorithm 2		Algorithm 3	
	B-233	B-409	K-233	K-409
3	128	242	131	207
4	132	240	135	210
5	N/A	N/A	136	213

(b) Recommended value for n in Alg 2 and 3.

3.3 Side-channel resistant multiplication on random binary curves

Another approach for scalar multiplication offering some resistance to side-channel attacks was proposed by López and Dahab [31] based on the Montgomery laddering technique. This approach requires $6M$ in $\mathbb{F}_{2^m}$ per iteration independently of the bit pattern in the scalar, and one of these multiplications is by the curve coefficient b. The curve being lately used for benchmarking purposes [7] at the 128-bit security level is an Edwards curve (curve2251) corresponding to the Weierstraß curve $y^2 + xy = x^3 + (z^{13} + z^9 + z^8 + z^7 + z^2 + z + 1)$. It is clear that this curve is especially tailored for this method due to the short length of b, reducing the cost of the algorithm to approximately $6M + 5S$ per iteration. At the same time, halving-based approaches are non-optimal for this curve due to the penalties introduced by the 4-cofactor [27]. Considering this and to partially satisfy the side-channel resistance offered by a bitsliced implementation such as [6], we restricted the choices of scalar multiplication at this security level to the Montgomery laddering approach.

4 Koblitz elliptic curves

A Koblitz curve $E_a(\mathbb{F}_q)$, also known as an Anomalous Binary Curve [29], is a special case of (1) where $b = 1$ and $a \in \{0, 1\}$. In a binary field, the map taking x to x^2 is an automorphism known as the Frobenius map. Since Koblitz curves are defined over the binary field $\mathbb{F}_2$, the Frobenius map and its inverse naturally extend to automorphisms of the curve denoted τ and τ^{-1}, respectively, where

[7] To be precise, the expression is obtained from the length-l width-ω NAF by setting $k'_i = 0$ for $i \geq l$.

$\tau(x, y) = (x^2, y^2)$. Moreover, $(x^4, y^4) + 2(x, y) = \mu(x^2, y^2)$ for every (x, y) on E_a, where $\mu = (-1)^{1-a}$; that is, τ satisfies $\tau^2 + 2 = \mu\tau$.

By solving the quadratic, we can associate τ with the complex number $\tau = \frac{-1+\sqrt{-7}}{2}$. Solinas [36] presents a τ-adic analogue of the usual NAF as follows. Since short representations are desirable, an element $\rho \in \mathbb{Z}[\tau]$ is found with $\rho \equiv k \pmod{\delta}$ of as small norm as possible, where $\delta = (\tau^m - 1)/(\tau - 1)$. Then for the subgroup of interest, $kP = \rho P$ and a width-ω τ-adic NAF ($\omega\tau$NAF) for ρ is obtained in a fashion that parallels the usual ωNAF. As in [36], define $\alpha_i = i \bmod \tau^\omega$ for $i \in \{1, 3, \ldots, 2^{\omega-1} - 1\}$. A $\omega\tau$NAF of a nonzero element ρ is an expression $\rho = \sum_{i=0}^{l-1} u_i\tau^i$ where each $u_i \in \{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{\omega-1}-1}\}$, $u_{l-1} \neq 0$, and at most one of any consecutive ω coefficients is nonzero. Scalar multiplication kP can be performed with the $\omega\tau$NAF expansion of ρ as

$$u_{l-1}\tau^{l-1}P + \cdots + u_2\tau^2P + u_1\tau P + u_0P \tag{3}$$

with $l - 1$ applications of τ and approximately $l/(\omega + 1)$ additions.

The length of the representation is at most $m + a$, and Solinas presents an efficient technique to find an estimate for ρ, denoted $\rho' = k$ partmod δ with $\rho' \equiv \rho \pmod{\delta}$, having expansion of length at most $m + a + 3$ [36,9]. Under reasonable assumptions, the algorithm will usually produce an estimate giving length at most $m + 1$. For simplicity, we will assume that the recodings obtained have this as an upper bound on length; small adjustments are necessary to process longer representations. Under these assumptions and properties of τ, scalars may be written

$$\begin{aligned} k &= \sum_{i=0}^{m} u_i\tau^i = u_0 + u_1\tau^1 + \cdots + u_m\tau^m \\ &= u_0 + u_1\tau^{-(m-1)} + u_2\tau^{-(m-2)} + \cdots + u_{m-1}\tau^{-1} + u_m = \sum_{i=0}^{m} u_i\tau^{-(m-i)} \end{aligned} \tag{4}$$

since $\tau^{-i} = \tau^{m-i}$ for all i.

4.1 Sequential algorithms for Koblitz curves

Algorithm 4 is a traditional left-to-right τ-and-add method, and expression (3) is computed from left to right, i.e., it starts processing u_{l-1} first, then u_{l-2} until it ends with the coefficient u_0. Step 1 computes $2^{\omega-2} - 1$ multiples of the point P, each at a cost of approximately one point addition (see Table 1 for a summary of the computational effort associated to this phase).

Alternatively, we can process right-to-left as shown in Algorithm 5. The multiple points of precomputation P_u in Algorithm 4 are exchanged for the same number of accumulators Q_u along with postcomputation. The cost of postcomputation is likely more than the precomputation of Algorithm 4; see Table 1 for a summary in the case where postcomputation is with projective additions. However, if the accumulator in Algorithm 4 is in projective coordinates, then Algorithm 5 has a less expensive evaluation phase since τ is applied to points in affine.

4.2 Parallel algorithm for Koblitz curves

The basic strategy in our parallel algorithm is to use (4) to reformulate the scalar multiplication in terms of both the τ and the τ^{-1} operators as

$$\begin{aligned} k &= \sum_{i=0}^{m} u_i \tau^i = u_0 + u_1\tau^1 + \cdots + u_n\tau^n + u_{n+1}\tau^{n+1} + \cdots + u_m\tau^m \\ &= u_0 + u_1\tau^1 + \cdots + u_n\tau^n + u_{n+1}\tau^{-(m-n-1)} + \cdots + u_m \\ &= \sum_{i=0}^{n} u_i\tau^i + \sum_{i=n+1}^{m} u_i\tau^{-(m-i)} \end{aligned} \tag{5}$$

where $0 < n < m$. Algorithm 3 illustrates a parallel approach suitable for two processors. Although similar in structure to Algorithm 2, a significant difference is the shared precomputation rather than the pre and postcomputation required in Algorithm 2.

The scalar representation is given by Solinas [36] and hence has an expected $m/(\omega + 1)$ point additions in the evaluation-stage, and an extra point addition at the end. There are also approximately m applications of τ or its inverse. If the field representation is such that these operators have similar cost or are sufficiently inexpensive relative to field multiplication, then the evaluation stage can be a factor 2 faster than a corresponding non-parallel algorithm.

Algorithm 3 $\omega\tau$NAF scalar multiplication: parallel

```
Input: $\omega, k \in [1, r-1]$, $P \in E_a(\mathbb{F}_{2^m})$ of order $r$, constant $n$ (e.g., from Table 1(b))
Output: $kP$
 1: $\rho \leftarrow k$ partmod $\delta$                          3: $P_u = \alpha_u P$,
 2: $\sum_{i=0}^{l-1} u_i\tau^i \leftarrow \omega\tau\mathrm{NAF}(\rho)$                 for $u \in \{1, 3, 5, \ldots, 2^{\omega-1} - 1\}$
    {Barrier}
 4: $Q_0 \leftarrow \mathcal{O}$                                 13: $Q_1 \leftarrow \mathcal{O}$
 5: for $i = n$ downto 0 do                        14: for $i = n + 1$ to $m$ do
 6:     $Q_0 \leftarrow \tau Q_0$                             15:     $Q_1 \leftarrow \tau^{-1} Q_1$
 7:     if $u_i = \alpha_j$ then                          16:     if $u_i = \alpha_j$ then
 8:         $Q_0 \leftarrow Q_0 + P_j$                        17:         $Q_1 \leftarrow Q_1 + P_j$
 9:     else if $u_i = -\alpha_j$ then                    18:     else if $u_i = -\alpha_j$ then
10:         $Q_0 \leftarrow Q_0 - P_j$                        19:         $Q_1 \leftarrow Q_1 - P_j$
11:     end if                                   20:     end if
12: end for                                      21: end for
    {Barrier}
22: return $Q \leftarrow Q_0 + Q_1$
```

As discussed before, unlike the ordinary width-ω NAF, the τ-adic version requires a relatively expensive calculation to find a short ρ with $\rho \equiv k \pmod{\delta}$. Hence, (a portion of) the precomputation is "free" in the sense that it occurs during scalar recoding. This can encourage the use of a larger window size ω.

Parallel variants on Koblitz curves The essential features exploited by Algorithm 3 are that the scalar can be efficiently represented in terms of the Frobenius map and that the map and its inverse can be efficiently applied, and hence the algorithm adapts to curves defined over small fields.

Algorithm 3 is attractive in the sense that two processors are directly supported without "extra" computations. However, if multiple applications of the "doubling step" are sufficiently inexpensive, then more processors and additional curves can be accommodated in a straightforward fashion without sacrificing the high-level parallelism of Algorithm 3. As an example for Koblitz curves, a variant on Algorithm 3 discards the applications of τ^{-1} (which may be more expensive than τ) and finds $kP = k^1(\tau^j P) + k^0 P = \tau^j(k^1 P) + k^0 P$ for suitable k^i and $j \approx m/2$ with traditional methods to calculate $k^i P$. The application of τ^j is low cost if there is storage for a per-field matrix as it was first discussed in [1].

5 Experimental results

We consider example fields $\mathbb{F}_{2^m}$ for $m \in \{233, 409\}$. These were chosen to address 112-bit and 192-bit security levels, according to the NIST recommendation. Moreover, we also address the 251-bit binary Edwards elliptic curve presented in [6]. The field $\mathbb{F}_{2^{233}}$ was also chosen as more likely to expose any overhead penalty in the parallelization compared with larger fields from NIST. Our C library coded all the algorithms using the GNU C 4.5.2 (gcc) and Intel 12 (icc) compilers, and the timings were obtained on a 3.326 GHz 32nm Intel *Westmere* processor i5 660.

Timings for field arithmetic appear in Table 2. For the most part, the times for gcc and icc are similar, although half-trace in $\mathbb{F}_{2^{233}}$ and square in $\mathbb{F}_{2^{251}}$ are exceptions. The difference in multiplication times between $\mathbb{F}_{2^{233}} = \mathbb{F}_2[z]/(z^{233}+z^{74}+1)$ and $\mathbb{F}_{2^{251}} = \mathbb{F}_2[z]/(z^{251}+z^7+z^4+z^2+1)$ is in reduction. The López-Dahab multiplier described in [2] was implemented as a baseline to quantify the speedup due to the native multiplier. The relatively expensive root in $\mathbb{F}_{2^{251}}$ is due to the representation chosen; if roots are of interest, then there are reduction polynomials giving faster root and similar times for other operations. Inversion via exponentiation (§2) gives I/M similar to that in [2] where an Euclidean algorithm variant was used with similar hardware but without the carry-less multiplier.

Table 2. Timings in clock cycles for field arithmetic operations. "op/M" denotes ratio to multiplication obtained from icc.

Base field	$\mathbb{F}_{2^{233}}$			$\mathbb{F}_{2^{251}}$			$\mathbb{F}_{2^{409}}$		
operation	gcc	icc	op/M	gcc	icc	op/M	gcc	icc	op/M
Multiplication	120	122	1.00	149	131	1.00	319	325	1.00
López-Dahab Mult.	234	289	2.37	335	350	2.67	599	643	1.98
Square root	59	55	0.45	132	122	0.93	54	48	0.15
Squaring	28	28	0.23	68	49	0.37	40	44	0.14
Half trace	156	126	1.03	181	171	1.31	301	291	0.90
Multi-Squaring	174	165	1.35	183	174	1.33	462	460	1.42
Inversion	2,764	2,577	21.12	3,511	3,305	25.42	9,381	9,789	30.12
4-τNAF	8,500	10,284	84.30	-	-	-	22,346	24,270	74.68
3-NAF	4,698	4,816	39.48	-	-	-	12,507	12,991	39.97
4-NAF	3,906	4,150	34.02	-	-	-	10,948	10,747	33.07
Recoding (halving)	1,260	1,380	11.31	-	-	-	3,308	2,980	9.17
Recoding (parallel)	800	850	6.97	-	-	-	2,050	1,850	5.69

Table 4 shows timings obtained for different variants of sequential and parallel scalar multiplication. We observe that for ωNAF recoding with $\omega = 3, 4$, the halve-and-add algorithm is always faster than its double-and-add counterpart. This performance is a direct consequence of the timings reported in Table 3, where the cost of one point doubling is roughly 5.1 and 4.6 multiplications whereas the cost of a point halving is of only 3.0 and 2.5 multiplications in the fields $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{409}}$, respectively. The parallel version that concurrently executes these algorithms in two threads computes one scalar multiplication with a latency that is roughly 32.9% and 34.8% smaller than that of the halve-and-add algorithm for the curves B-233 and B-409, respectively.

Table 3. Timings in clock cycles for curve arithmetic operations. "op/M" denotes ratio to multiplication obtained from icc.

Elliptic curve operations	B-233 gcc	B-233 icc	B-233 op/M	B-409 gcc	B-409 icc	B-409 op/M
Doubling (LD)	625	626	5.13	1,484	1,508	4.64
Addition (KIM)	1,116	1,093	8.96	2,771	2,801	8.61
Addition (LD Mixed)	1,167	1,142	9.36	2,852	2,888	8.88
Addition (LD General)	1,834	1,817	14.89	4,543	4,606	14.17
Halving	413	371	3.04	842	830	2.55

Table 4. Timings in 10^3 clock cycles for scalar multiplication.

ω	Scalar mult random curves	B-233 gcc	B-233 icc	B-409 gcc	B-409 icc
3	Double-and-add	224.6	220.5	911.5	923.5
3	Halve-and-add	182.8	176.1	706.2	705.9
3	(Dbl,Halve)-and-add	117.9	113.8	452.3	448.9
4	Double-and-add	216.7	212.0	871.5	886.8
4	Halve-and-add	176.1	165.8	659.8	656.3
4	(Dbl,Halve)-and-add	116.5	111.3	432.4	428.5

	Side-channel resistant scalar multiplication	curve2251 gcc	curve2251 icc
	Montgomery laddering	304.0	264.0

ω	Scalar mult Koblitz curves	K-233 gcc	K-233 icc	K-409 gcc	K-409 icc
3	Alg. 4	103.8	101.3	393.8	388.9
3	Alg. 5	**92.3**	92.4	**353.6**	363.2
3	(τ,τ)-and-add	72.7	**70.4**	252.9	236.6
3	Alg. 3	78.4	75.7	252.9	**235.3**
4	Alg. 4	90.8	88.3	338.2	331,9
4	Alg. 5	84.6	**83.8**	**309.0**	319.0
4	(τ,τ)-and-add	65.7	**61.6**	221.5	204.8
4	Alg. 3	70.1	67.1	218.8	**202.6**
5	Alg. 4	87.5	**83.2**	312.4	306.6
5	Alg. 5	88.3	86.8	**302.0**	310.9
5	(τ,τ)-and-add	61.6	**55.9**	199.0	**181.0**
5	Alg. 3	64.8	61.2	195.6	184.1

The bold entries for Koblitz curves identify fastest timings per category (i.e., considering the compiler, curve, and the specific value of ω used in the ωNAF recoding). For smaller ω, Algorithm 5 has an edge over Algorithm 4 because τ is applied to points in affine; this advantage diminishes with increasing ω due to postcomputation cost. "(τ,τ)-and-add" denotes the parallel variant described in §4.2. There is a storage penalty for a linear map, but applications of τ^{-1} are eliminated (of interest when τ is significantly less expensive). Given the modest cost of the multi-squaring operation (with an equivalent cost of less than 1.42 field multiplications, see Table 2), the (τ,τ)-and-add parallel variant is usually faster than Algorithm 3. When using $\omega = 5$, the parallel (τ,τ)-and-add algorithm computes one scalar multiplication with a latency that is roughly 32.8% and 40% smaller than that of the best sequential algorithm for the curves K-233 and K-409, respectively.

Per-field storage and coding techniques compute half-trace at cost comparable to field multiplication, and methods based on halving continue to be fastest for suitable random curves. However, the hardware multiplier and squaring (via shuffle) give a factor 2 advantage to Koblitz curves in the examples from NIST. This is larger than in [16,21], where a 32-bit processor in the same general family as the i5 has half-trace at approximately half the cost of a field multiplication for B-233 and a factor 1.7 advantage to K-163 over B-163 (and the factor would have been smaller for K-233 and B-233). It is worth remarking that the parallel scalar multiplications versions shown in Table 4 look best for bigger curves and larger ω.

6 Conclusion and future work

In this work we achieve the fastest timings reported in the open literature for software computation of scalar multiplication in NIST and Edwards binary elliptic curves defined at the 112-bit, 128-bit and 192-bit security levels. The fastest curve implemented, namely NIST K-233, can compute one scalar multiplication in less than 17μs, a result that is not only much faster than previous software implementations of that curve, but is also quite competitive with the computation time achieved by state-of-the-art hardware accelerators working on similar or smaller curves [24,1].

These fast timings were obtained through the usage of the native carry-less multiplier available in the newest Intel processors. At the same time, we strive to use the best algorithmic techniques, and the most efficient elliptic curve and finite field arithmetic formulae. Further, we proposed effective parallel formulations of scalar multiplication algorithms suitable for deployment in multi-core platforms.

The curves over binary fields permit relatively elegant parallelization with low synchronization cost, mainly due to the efficient halving or τ^{-1} operations. Parallelizing at lower levels in the arithmetic would be desirable, especially for curves over prime fields. Grabher et al.[17] apply parallelization for extension field multiplication, but times for a base field multiplication in a 256-bit prime field are relatively slow compared with Beuchat et al.[8]. On the other hand, a strategy that applies to all curves performs point doubles in one thread and point additions in another. The doubling thread stores intermediate values corresponding to nonzero digits of the NAF; the addition thread processes these points as they become available. Experimentally, synchronization cost is low, but so is the expected acceleration. Against the fastest times in Longa and Gebotys [30] for a curve over a 256-bit prime field, the technique would offer roughly 17% improvement, a disappointing return on processor investment.

The new native support for binary field multiplication allowed our implementation to improve by 16% the previous speed record for side-channel resistant scalar multiplication in random elliptic curves. It is hard to predict what will be the superior strategy between a conventional non-bitsliced or a bitsliced implementation on future revisions of the target platform: the latency of the carry-less multiplier instruction has clear room for improvement, while the new AVX instruction set has 256-bit registers. An issue with the current Sandy Bridge version of AVX is that `xor` throughput for operations with register operands was decreased significantly from 3 operations per cycle in SSE to 1 operation in AVX. The resulting performance of a bitsliced implementation will ultimately rely on the amount of work which can be scheduled to be done mostly in registers.

Acknowledgments

We wish to thank the University of Waterloo and especially, Professor Alfred Menezes for useful discussions related to this work during a visit by three of the authors, where the idea of this project was discussed, planned and a portion of the development phase of this work was done.

References

1. O. Ahmadi, D. Hankerson, and F. Rodríguez-Henríquez. Parallel formulations of scalar multiplication on Koblitz curves. *J. UCS*, 14(3):481–504, 2008.
2. D. F. Aranha, J. López, and D. Hankerson. Efficient software implementation of binary field arithmetic using vector instruction sets. In M. Abdalla and P. S. L. M. Barreto, editors, *The First International Conference on Cryptology and Information Security (LATINCRYPT 2010)*, volume 6212 of *Lecture Notes in Computer Science*, pages 144–161, 2010.
3. R. M. Avanzi. Another look at square roots (and other less common operations) in fields of even characteristic. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *14th International Workshop on Selected Areas in Cryptography (SAC 2007)*, volume 4876 of *Lecture Notes in Computer Science*, pages 138–154. Springer, 2007.
4. M. Bellare, editor. *Advances in Cryptology—CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*. 20th Annual International Cryptology Conference, Santa Barbara, California, August 2000, Springer-Verlag, 2000.
5. D. Bernstein and T. Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. In *Proceedings 8th International Conference on Finite Fields and Applications (Fq8)*, volume 461, pages 1–20. AMS, 2008.

6. D. J. Bernstein. Batch binary Edwards. In S. Halevi, editor, *Advances in Cryptology—CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16–20, 2009, Springer, 2009.
7. D. J. Bernstein and T. Lange, editors. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. http://bench.cr.yp.to, accessed 30 Mar 2011.
8. J.-L. Beuchat, J. Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. In M. Joye, A. Miyaji, and A. Otsuka, editors, *Pairing-Based Cryptography – Pairing 2010*, volume 6487 of *Lecture Notes in Computer Science*, pages 21–39, 2010.
9. I. F. Blake, V. K. Murty, and G. Xu. A note on window τ-NAF algorithm. *Inf. Process. Lett.*, 95(5):496–502, 2005.
10. M. Bodrato. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In C. Carlet and B. Sunar, editors, *Arithmetic of Finite Fields (WAIFI 2007)*, volume 4547 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2007.
11. J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. ECC2K-130 on Cell CPUs. In D. J. Bernstein and T. Lange, editors, *3rd International Conference on Cryptology in Africa (AFRICACRYPT 2010)*, volume 6055 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2010.
12. P. G. Comba. Exponentiation Cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.
13. E. Dahmen, K. Okeya, and D. Schepers. Affine precomputation with sole inversion in elliptic curve cryptography. In J. Pieprzyk, H. Ghodosi, and E. Dawson, editors, *Information Security and Privacy (ACISP 2007)*, volume 4586 of *Lecture Notes in Computer Science*, pages 245–258. Springer-Verlag, 2007.
14. N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New frontiers in performance improvement and energy efficiency. White paper. http://software.intel.com/.
15. A. Fog. Instruction tables: List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. http://www.agner.org/optimize/instruction_tables.pdf, accessed 01 Mar 2011.
16. K. Fong, D. Hankerson, J. López, and A. Menezes. Field inversion and point halving revisited. *IEEE Transactions on Computers*, 53(8):1047–1059, 2004.
17. P. Grabher, J. Großschädl, and D. Page. On software parallel implementation of cryptographic pairings. Cryptology ePrint Archive, Report 2008/205, 2008. http://eprint.iacr.org/.
18. J. Guajardo and C. Paar. Itoh-Tsujii inversion in standard basis and its application in cryptography and codes. *Designs, Codes and Cryptography*, 25(2):207–216, 2002.
19. S. Gueron. Intel Advanced Encryption Standard (AES) Instructions Set. White paper. http://software.intel.com/.
20. S. Gueron and M. E. Kounavis. Carry-less multiplication and its usage for computing the GCM mode. White paper. http://software.intel.com/.
21. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Secaucus, NJ, USA, 2003.
22. Intel. Intel SSE4 Programming Reference. Technical Report. http://software.intel.com/.
23. T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.
24. K. Järvinen. Optimized FPGA-based elliptic curve cryptography processor for high-speed applications. *Integration, the VLSI Journal*, to appear.
25. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics-Doklady 7, 595-596, 1963.
26. K. H. Kim and S. I. Kim. A new method for speeding up arithmetic on elliptic curves over binary fields. Cryptology ePrint Archive, Report 2007/181, 2007. http://eprint.iacr.org/.
27. B. King and B. Rubin. Improvements to the point halving algorithm. In H. Wang, J. Pieprzyk, and V. Varadharajan, editors, *9th Australasian Conference on Information Security and Privacy (ACISP 2004)*, volume 3108 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2004.
28. E. Knudsen. Elliptic scalar multiplication using point halving. In K. Lam and E. Okamoto, editors, *Advances in Cryptology—ASIACRYPT '99*, volume 1716 of *Lecture Notes in Computer Science*, pages 135–149. International Conference on the Theory and Application of Cryptology and Information Security, Singapore, November 1999, Springer-Verlag, 1999.
29. N. Koblitz. CM-curves with good cryptographic properties. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 279–287. Springer-Verlag, 1992.
30. P. Longa and C. H. Gebotys. Efficient techniques for high-speed elliptic curve cryptography. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems, (CHES 2010)*, volume 6225 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2010.
31. J. López and R. Dahab. Fast multiplication on elliptic curves over GF(2^m) without precomputation. In Ç. K. Koç and C. Paar, editors, *First International Workshop on Cryptographic Hardware and Embedded Systems (CHES 99)*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 1999.

32. J. López and R. Dahab. High-speed software multiplication in GF(2^m). In B. K. Roy and E. Okamoto, editors, *1st International Conference in Cryptology in India (INDOCRYPT 2000)*, volume 1977 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 2000.
33. P. L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.
34. National Institute of Standards and Technology (NIST). *Recommended Elliptic Curves for Federal Government Use*. NIST Special Publication, July 1999. http://csrc.nist.gov/csrc/fedstandards.html.
35. R. Schroeppel. Elliptic curves: Twice as fast! Presentation at the CRYPTO 2000 [4] Rump Session, 2000.
36. J. A. Solinas. Efficient arithmetic on Koblitz curves. *Designs, Codes and Cryptography*, 19(2-3):195–249, 2000.
37. D. W. Wall. Limits of instruction-level parallelism. In *4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS 91)*, pages 176–188, New York, NY, 1991. ACM.
38. W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

A Algorithms

Algorithm 4 Left-to-right τ-and-add scalar multiplication

```
Input: $\omega, k \in [1, r-1]$, $P \in E_a(\mathbb{F}_{2^m})$ of order $r$
Output: $kP$
 1: Compute $P_u = \alpha_u P$ for $u \in \{1, 3, 5, \ldots, 2^{\omega-1}-1\}$ where $\alpha_i = i \bmod \tau^{\omega}$
 2: Compute $\rho = k$ partmod $\delta$ and $\omega\tau$NAF($\rho$) $= \sum_{i=0}^{l-1} u_i \tau^i$; $Q \leftarrow \mathcal{O}$
 3: for $i = l-1$ downto $0$ do
 4:     $Q \leftarrow \tau Q$
 5:     if $u_i = \alpha_j$ then
 6:         $Q \leftarrow Q + P_j$
 7:     else if $u_i = -\alpha_j$ then
 8:         $Q \leftarrow Q - P_j$
 9:     end if
10: end for
11: return $Q$
```

Algorithm 5 Right-to-left τ-and-add scalar multiplication

```
Input: $\omega, k \in [1, r-1]$, $P \in E_a(\mathbb{F}_{2^m})$ of order $r$
Output: $kP$
 1: $\rho = k$ partmod $\delta$, $\omega\tau$NAF($\rho$) $= \sum_{i=0}^{l-1} u_i \tau^i$ $Q_u = \mathcal{O}$ for $u \in I = \{1, 3, \ldots, 2^{\omega-1}-1\}$
 2: for $i = 0$ to $l-1$ do
 3:     if $u_i = \alpha_j$ then
 4:         $Q_j \leftarrow Q_j + P$
 5:     else if $u_i = -\alpha_j$ then
 6:         $Q_j \leftarrow Q_j - P$
 7:     end if
 8:     $P \leftarrow \tau P$
 9: end for
10: return $Q \leftarrow \sum_{u \in I} \alpha_u Q_u$
```