Journal of Universal Computer Science, vol. 14, no. 3 (2008), 481-504
submitted: 2/6/07, accepted: 1/11/07, appeared: 1/2/08 © *J.UCS*

Parallel Formulations of Scalar Multiplication on Koblitz Curves

Omran Ahmadi
(University of Waterloo, Canada
oahmadid@uwaterloo.ca)

Darrel Hankerson
(Auburn University, USA
hankedr@auburn.edu)

Francisco Rodríguez-Henríquez†
(CINVESTAV-IPN, Mexico
francisco@cs.cinvestav.mx)

Abstract: We present an algorithm that by using the τ and τ^{-1} Frobenius operators concurrently allows us to obtain a parallelized version of the classical τ-and-add scalar multiplication algorithm for Koblitz elliptic curves. Furthermore, we report suitable irreducible polynomials that lead to efficient implementations of both τ and τ^{-1}, thus showing that our algorithm can be effectively applied on all the NIST-recommended curves. We also present design details of software and hardware implementations of our procedure. In a two-processor workstation software implementation, we report experimental data showing that our parallel algorithm is able to achieve a speedup factor of almost 2 when compared with the standard sequential point multiplication. In our hardware implementation, the parallel version yields a more modest acceleration of 17% when compared with the traditional point multiplication algorithm. Although the focus is on Koblitz curves, analogous strategies are discussed for other curves, in particular for random curves over binary fields.
Key Words: Elliptic Curve Cryptography; Koblitz Curves; Finite Field Arithmetic; Fast Cryptographic Algorithms.
Category: E.3, B.2.4

1 Introduction

First proposed in 1991 by N. Koblitz [Koblitz 1992], *Koblitz Elliptic Curves* have been subject of extensive analysis and study. Given a finite field $\mathbb{F}_q$ for $q = 2^m$, a Koblitz curve $E_a(\mathbb{F}_q)$, also known as an Anomalous Binary Curve (ABC), is defined as the set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy the equation

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \quad a \in \{0, 1\}, \tag{1}$$

together with a point at infinity denoted by $\mathcal{O}$. It is known that $E_a(\mathbb{F}_q)$ 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

† Corresponding author.

point resulting of adding $P+P+\cdots+P$, 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. It is known that the computational cost of the double-and-add algorithm is $m-1$ point doublings plus an average of $\frac{m-1}{2}$ point additions.

Most works published in this area have strived for reducing the cost associated to the double-and-add method by following two main strategies: reducing the computational complexity of point addition and point doubling primitives, and reducing the number of times that the point addition primitive is invoked during the algorithm execution. A significant improvement in performance can be obtained when the point P is known in advance by using precomputation (if memory space permits) and techniques such as the comb method [Hankerson et al. 2004].

However those methods are not very practical when dealing with the so-called *unknown point* case, where the elliptic curve point to be processed is not known in advance, thus precluding off-line precomputation. In the rest of this paper we will assume that we are working in the unknown point scenario, with sufficient memory space for storing a few multiples of the point to be processed. In this scenario, windowing methods reduce the number of point additions, but not point doubles, and hence the savings are typically modest. An exceptional case is that of Koblitz curves.

Koblitz curves can significantly speed scalar multiplication by substituting the traditional double-and-add algorithm with a "τ-and-add" procedure as it was presented in [Koblitz 1992, Solinas 2000]. In the context of Koblitz curves, the τ operator is defined as the mapping of a point $P=(x,y)\in E_a$ to the point $\tau P := (x^2,y^2)$, which also belongs to the curve E_a. Since field squaring is a linear operation in binary extension fields, the τ operator can be implemented far more efficiently than point double or point halving primitives. For this reason, the cost of scalar multiplication is determined largely by the number of point additions, although the applications of τ are typically not completely free.

Several efforts for speeding elliptic curve scalar multiplication on Koblitz curves have been reported both in hardware and software platforms [Avanzi and Sica 2006, Wong et al. 2006, Dimitrov et al. 2006, Hankerson et al. 2004, Lutz 2004, Solinas 2000, Hankerson et al. 2000]. Authors in [Avanzi et al. 2006] proposed to reduce the number of group operations by using a wide-double non-adjacent form of the scalar k combined with applications of the point halving primitive. Compared with the τ-NAF method of [Solinas 2000], a savings of 25% in point additions are obtained. However, the effective savings are somewhat less since the wide-double non-adjacent form leads to an increment in the number of Frobenius applications (the scalar multiplication algorithm must execute two loops of about m iterations each). Further, the savings are in the case where there is no room for a few points of (on-line) precomputation. The same savings in point additions (and without the extra applications of τ)

can be obtained if an extra point of storage is permitted.

[Dimitrov et al. 2005, Avanzi and Sica 2006, Wong et al. 2006, Dimitrov et al. 2006] proposed the idea of representing the scalar k using a double base rather than the traditional binary form. Point doublings can be partially substituted with advantage by tripling, quadrupling and even halving a point. In [Dimitrov et al. 2005], k is represented as the sum or difference of numbers of the form 2^i3^j, and integrated operations such as $4Q + P$ are exploited to speed scalar multiplication. On random binary curves, the savings over the use of NAF is modest; further, the analysis is for affine coordinate arithmetic, which may be of less interest due to the relatively high cost of field inversion. In [Avanzi and Sica 2006], a double-base expansion using τ and 3 is shown to give a sublinear scalar multiplication algorithm on Koblitz curves. The result is asymptotic, and the practical applicability to curves of cryptographic interest requires more examination. In [Dimitrov et al. 2006], a provably sublinear point multiplication algorithm on Koblitz curves where the scalar k was represented using multiple-base expansions was presented. The reconfigurable hardware implementation of that algorithm was reported to require 35.75μs on a Xilinx Virtex-II device in order to compute a scalar multiplication operation on the Koblitz elliptic curve $E_1(\mathbb{F}_{2^{163}})$.

In this work we discuss yet another approach for accelerating scalar multiplication on Koblitz curves: the use of parallel strategies. The parallelization occurs at a high level, an advantage in implementation. First, we show that the τ^{-1} Frobenius operator can be successfully applied in the domain of Koblitz elliptic curves giving extra flexibility and potential speedup to known elliptic curve scalar multiplication procedures. Secondly, we present architectures that utilizing τ and τ^{-1} blocks offer significant speedups. Provided that the costs of computing the τ and the τ^{-1} operators are approximately equal, the performance increments attained by our schemes become maximal.

Unfortunately, several reduction polynomials recommended by NIST lead to expensive computations of the τ^{-1} operator, thus precluding the effectiveness of our architecture for providing major performance improvements. In an effort to overcome this difficulty, we present a list of alternative reduction polynomials that yield efficient computations of the Frobenius inverse operator. We also indicate the exact hardware cost of computing the τ^{-1} operator when the proposed alternative polynomials are used.

Additionally, we show how multiple applications of the Frobenius operator of the form $\tau^{\pm n}$ can be applied for producing an *interleaved* version of the basic τ and τ^{-1} scheme. That interleaved version provides a further improvement in the performance gain. We also discuss an analogous parallelization strategy for random curves over binary fields when point halving methods apply. This strategy along with the architectures discussed for Koblitz curves broaden the applicability of our approach to virtually all relevant elliptic curves defined over GF(2^m).

Finally, we present actual implementations of the proposed architectures in both hardware and software platforms. In hardware, a single point addition block may provide a speedup of up to 17% when compared with the traditional sequential version. Furthermore, for those designs where two point addition units are affordable, we show that our point multiplication formulation can speed up the operation by a factor close to 2. Experimental data on a two-processor workstation is presented that approaches this ceiling.

The rest of this paper is organized as follows. In §2 some relevant mathematical concepts are briefly outlined. Section 3 presents a parallel formulation of scalar multiplication on Koblitz curves. Algorithm variants and analogues for other curves appear in §4. Section 5 discusses relevant implementation aspects of the proposed parallel algorithms for software and hardware environments. Finally, in §6 some conclusions are drawn.

2 Mathematical background

In a binary field, the map taking x to x^2 is an automorphism of the field called Frobenius map whose inverse, the square root map, takes x to $\sqrt{x}$. Since Koblitz curves are defined over the binary field $GF(2)$, the Frobenius map and its inverse naturally extend to two automorphisms of the curve denoted τ and τ^{-1}, respectively. The τ map takes (x, y) to (x^2, y^2) and $\mathcal{O}$ to $\mathcal{O}$. Similarly, the τ^{-1} map takes (x, y) to $(\sqrt{x}, \sqrt{y})$ and $\mathcal{O}$ to $\mathcal{O}$. Given the above definitions, it is easy to show that in $E_a(\mathbb{F}_{2^m})$, $\tau^{-i} = \tau^{m-i}$ for all i. Moreover, it has been shown that $(x^4, y^4) + 2(x, y) = \mu(x^2, y^2)$ 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 [Solinas 2000] presents a τ-adic analogue of the usual non-adjacent form (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 τ-adic NAF (τNAF) for ρ is obtained in a fashion that parallels the usual NAF. For simplicity, we write this τNAF (of ρ) as $k = \sum_{i=0}^{l-1} u_i \tau^i$, where each $u_i \in \{0, \pm 1\}$ and l is the expansion's length. The scalar multiplication kP can then be computed with a corresponding addition-subtraction method with the τNAF.

Standard (NAF) addition-subtraction method computes a scalar multiplication in about m doubles and $m/3$ additions [Hankerson et al. 2004]. Similarly, the corresponding τNAF method implies the computation of l τ mappings (field squarings) and approximately $l/3$ additions. On the other hand, it is possible to process $\omega \geq 2$ digits of the scalar k at a time. As in [Solinas 2000], define $\alpha_i = i \bmod \tau^\omega$ for $i \in \{1, 3, 5, \ldots, 2^{\omega-1} - 1\}$. A width-$\omega$ τNAF of a nonzero element k is an expression $k = \sum_{i=0}^{l-1} u_i \tau^i$ where each $u_i \in \{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{\omega-1}-1}\}$ and $u_{l-1} \neq 0$, and at most one of any consecutive ω coefficients is nonzero. Therefore, the scalar multiplication kP can be performed with the $\omega\tau$NAF expansion of k as

$$u_{l-1}\tau^{l-1}P + \cdots + u_2\tau^2 P + u_1\tau P + u_0 P. \tag{2}$$

Algorithm 1 $\omega\tau$NAF Scalar Multiplication [Solinas 2000]

Require: $\omega\tau\text{NAF}(k) = \sum_{i=0}^{l-1} u_i\tau^i$, $P \in E_a(\mathbb{F}_{2^m})$.
Ensure: kP

Pre-compute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, \dots, 2^{\omega-1} - 1\}$ where $\alpha_i = i \bmod \tau^\omega$;
$Q \leftarrow \mathcal{O}$;
for i from $l - 1$ **downto** 0 **do**
    $Q \leftarrow \tau Q$;
    if $u_i \neq 0$
        Find u such that $\alpha_{\pm u} = \pm u_i$;
        if $u > 0$
            $Q \leftarrow Q + P_u$;
        else
            $Q \leftarrow Q - P_{-u}$;
Return Q;

The length of ρ is at most $m + a$, and Solinas presents an efficient technique to find an estimate for ρ that is of length at most $m + a + 3$ [Solinas 2000, Blake et al. 2005]. Under reasonable assumptions, the algorithm will usually produce an estimate with 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

$$\begin{aligned} k &= \sum_{i=0}^{m} u_i\tau^i = u_0 + u_1\tau^1 + \cdots + u_m\tau^m \qquad (3) \\ &= u_0 + u_1\tau^{-(m-1)} + u_2\tau^{-(m-2)} + \cdots + u_{m-1}\tau^{-1} + u_m = \sum_{i=0}^{m} u_i\tau^{-(m-i)} \end{aligned}$$

Summarizing, Koblitz elliptic curve scalar multiplication can be accomplished by processing elliptic point additions and τ and/or τ^{-1} mappings.

3 Parallel scalar multiplication on Koblitz curves

Often, a point multiplication algorithm on a Koblitz curve is divided into two main phases: a $\omega\tau$NAF expansion of the scalar k, and the scalar multiplication itself based on the τ Frobenius operator and elliptic curve addition sequences. If projective coordinates are in use, then the traditional τ-and-add method is Algorithm 1, and expression (2) 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.

The basic strategy in our parallel algorithm is to use (3) to reformulate the scalar multiplication in terms of both the τ and the τ^{-1} operators as

Algorithm 2 $\omega\tau$NAF Scalar Multiplication: Parallel Version

Require: $\omega\tau\text{NAF}(k) = \sum_{i=0}^{m} u_i \tau^i$, $P \in E_a(\mathbb{F}_{2^m})$, integer $n < m$.
Ensure: kP

Precompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, \ldots, 2^{\omega-1} - 1\}$ where $\alpha_i = i \bmod \tau^\omega$;
$Q = R = \mathcal{O}$;
Find the largest n_1 such that $u_{n_1} \neq 0$ and $n_1 \leq n$.
Find the smallest n_2 such that $u_{n_2} \neq 0$ and $n_2 > n$.

for i from n_1 **downto** 0 **do**	**for** j from n_2 **to** m **do**
$Q \leftarrow \tau Q$;	$R \leftarrow \tau^{-1} R$
if $u_i \neq 0$ **then**	**if** $u_j \neq 0$
Find u such that $\alpha_{\pm u} = \pm u_i$;	Find u such that $\alpha_{\pm u} = \pm u_j$;
if $u > 0$ **then**	**if** $u > 0$ **then**
$Q \leftarrow Q + P_u$;	$R \leftarrow R + P_u$;
else	**else**
$Q \leftarrow Q - P_{-u}$;	$R \leftarrow R - P_{-u}$;

$Q \leftarrow Q + R$;
Return Q;

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

where $0 < n < m$. This formulation leads to a parallel version of Algorithm 1 which is shown in Algorithm 2.

It is worth mentioning that the joint use of the Frobenius map and its inverse has appeared in other contexts. For example, taking advantage of the low cost of inverting τ when using normal basis representations, Solinas [Solinas 2000, Algorithm 5] proposed a right-to-left windowing method for kP that uses both τ and τ^{-1}. Furthermore, Hasan in [Hasan 2001] proposed the parallel application of the Frobenius operator and its inverse in order to instrument countermeasures for side-channel attacks. The main idea was to *randomize* the scalar k, so that an opponent could not easily guess the actual value of the coefficient being processed. This procedure, however, has the associated performance penalty of computing more point addition operations.

We stress that the aim of Algorithm 2 is radically different than the two aforementioned works. Indeed, by means of the expression (4), we effectively split the m-bit computation of kP into two subsequences of length $m/2$ bits each, which essentially

Table 1: Computing kP over $GF(2^{17})$, $k = 195$ with Algorithm 2.

i	u_i	τ-and-Add	j	u_j	τ^{-1}-and-Add
6	0	$Q = \tau P$	11	0	$R = \tau^{-1}P = \tau^{16}P$
5	-1	$Q = (\tau^2 - 1) \cdot P$	12	0	$R = \tau^{-2}P = \tau^{15} \cdot P$
4	0	$Q = (\tau^3 - \tau) \cdot P$	13	0	$R = \tau^{-3}P = \tau^{14} \cdot P$
3	0	$Q = (\tau^4 - \tau^2) \cdot P$	14	1	$R = (\tau^{-4} + 1) \cdot P = (\tau^{13} + 1) \cdot P$
2	1	$Q = (\tau^5 - \tau^3 + 1) \cdot P$	15	0	$R = (\tau^{-5} + \tau^{-1}) \cdot P = (\tau^{12} + \tau^{16}) \cdot P$
1	0	$Q = (\tau^6 - \tau^4 + \tau) \cdot P$	16	1	$R = (\tau^{-6} + \tau^{-2} + 1) \cdot P = (\tau^{11} + \tau^{15} + 1) \cdot P$
0	-1	$Q = (\tau^7 - \tau^5 + \tau^2 - 1) \cdot P$	17	0	$R = (\tau^{-7} + \tau^{-3} + \tau^{-1}) \cdot P = (\tau^{10} + \tau^{14} + \tau^{16}) \cdot P$

do not share any common operations (an obvious advantage for computing in parallel). Hence, provided that two independent processing units are available, Algorithm 2 can speed the computation time of Algorithm 1 by a factor of almost 2, as will be discussed in more detail in the rest of this section.

3.1 An example

Let us consider the binary extension field $GF(2^m)$ with $m = 17$, generated with the irreducible trinomial $f(x) = x^{17} + x^3 + 1$. Referring to the Koblitz elliptic curve of Eq. (1) with $a = 1$, suppose we want to compute point multiplication using the positive integer scalar $k = 195$. Then, the τNAF expansion of k is given as $-1 + \tau^2 - \tau^5 + \tau^7 + \tau^{10} + \tau^{14} + \tau^{16}$, which can be rewritten as

$$\begin{aligned}\tau\text{NAF}(k) &= -1 + \tau^2 - \tau^5 + \tau^7 + \tau^{10} + \tau^{14} + \tau^{16} \\ &= -1 + \tau^2 - \tau^5 + \tau^7 + \tau^{-(m-10)} + \tau^{-(m-14)} + \tau^{-(m-16)} \\ &= -1 + \tau^2 - \tau^5 + \tau^7 + \tau^{-7} + \tau^{-3} + \tau^{-1}.\end{aligned}$$

We can compute in parallel above scalar multiplication using Algorithm 2 with $n = \lfloor \frac{m}{2} \rfloor = 8$. Indeed, initialization sets $n_1 = 7$ and $n_2 = 10$, and $Q = R = P$ after the first iteration. Notice also that the τNAF expansion of k can be written in vector form as $\tau\text{NAF}(k) = [\bar{1}0100\bar{1}01001000101]$, where $\bar{1} = -1$. Then, the algorithm dataflow is as shown in Table 1. The required result is found in step 13 of Algorithm 2 as

$$Q + R = (\tau^7 - \tau^5 + \tau^2 - 1) \cdot P + (\tau^{10} + \tau^{14} + \tau^{16}) \cdot P = kP.$$

Provided that the computational cost of the τ and the τ^{-1} are about the same,[1] and that two independent point addition units are available, Algorithm 2 can perform the scalar multiplication $195P$, as described above, at a time cost equivalent to the computation of four point additions plus seven applications of the τ operator, as opposed to six point additions and sixteen applications of the τ operator required by Algorithm 1.

[1] See Appendix A for details of how to compute the τ and the τ^{-1} operators efficiently.

3.2 Computational costs in Algorithm 2

We analyze cost issues in Algorithm 2 in the case where the precomputation (Step 1) must occur on-line (the point-not-known-in-advance case). However, the step corresponding to "doubling" in traditional algorithms is inexpensive in Koblitz curves (in contrast to the case for random curves). This is similar to situation for supersingular curves over fields of characteristic 2 or 3 where point doublings (characteristic 2) or triplings (characteristic 3) are inexpensive. In such cases, off-line precomputation may offer only modest acceleration if relatively few (e.g., 8) points of precomputation are used, and hence the algorithm may also be appropriate in the known-point case.

3.2.1 Precomputation

Step 1 of Algorithm 2 computes $2^{\omega-2} - 1$ multiples of the point P, each at a cost of approximately one point addition. 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 [NIST, 1999] and with on-line precomputation, $\omega \leq 6$ is optimal in the sense that total point additions are minimized. A valuable feature of Algorithm 2 is the *shared* use of the pre-computed points by the pair of parallel loops in steps 5–12.

In many cases, the recoding in $\omega\tau\mathrm{NAF}(k)$ is performed on-line and can be considered as part of the precomputation step. Unlike the ordinary width-ω NAF, the τ-adic version requires a relatively expensive calculation of a short ρ with $\rho \equiv k \pmod{\delta}$ where $\delta = (\tau^m - 1)/(\tau - 1)$ [Solinas 2000]. This calculation can be done in parallel with the computation of P_u.

3.2.2 Square and square root

The choice of the input parameter n in Algorithm 2 depends on the relative cost difference between τ and τ^{-1}. In a normal basis representation for the field, these operators are of very low and similar cost. However, polynomial basis multipliers are typically preferred (due to performance), and so we shall assume a polynomial basis representation with a NIST-like reduction polynomial (that is, a trinomial or pentanomial $x^m + p(x)$ where $\deg p$ is small relative to m).

Under the given scenario, applying τ is typically inexpensive and can be done via linearity by "thinning" the coefficients followed by reduction. However, the cost of τ^{-1} can be significantly higher. A generic method splits the computation of the square root of a field element c as

$$\sqrt{c} = \left(\sum c_i x^i\right)^{1/2} = \sum_{i \text{ even}} c_i x^{i/2} + \sqrt{x} \sum_{i \text{ odd}} c_i x^{(i-1)/2}$$

where $\sqrt{x}$ is a per-field precomputation. The computation is less expensive than a field multiplication, but can be significantly more than the cost of a squaring.

Table 2: Candidate reduction polynomials for $\mathbb{F}_{2^m}$, $m \in \{163, 233, 283\}$, giving low cost squaring and/or square roots. The value $N(M)$ is listed where N is the total number of XORs required and M is the number of T_x delays needed for computing the operation, where T_x is the time delay of one XOR gate.

Reduction polynomial	$\sqrt{x}$	$N(M)$ c^2	$N(M)$ $\sqrt{c}$
$m = 163$			
$x^{163} + x^7 + x^6 + x^3 + 1$	(79 terms)	252(2)	1619(6)
$x^{163} + x^{65} + x^{35} + x^{33} + 1$	$x^{18}(x^{64} + 1) + x^{33} + x^{17}$	272(3)	243(2)
$x^{163} + x^{99} + x^{97} + x^3 + 1$	$x^{50}(x^{32} + 1) + x^{49} + x^2$	362(3)	243(2)
$m = 233$			
$x^{233} + x^{74} + 1$	$(x^{32} + x^{117} + x^{191})(x^{37} + 1)$	153(1)	153(2)
$x^{233} + x^{159} + 1$	$x^{117} + x^{80}$	116(2)	116(1)
$m = 283$			
$x^{283} + x^{12} + x^7 + x^5 + 1$	(68 terms)	430(2)	2677(6)
$x^{283} + x^{97} + x^{89} + x^{87} + 1$	$x^{142} + x^{49} + x^{45} + x^{44}$	514(3)	423(2)
$x^{283} + x^{105} + x^{73} + x^{41} + 1$	$x^{142} + x^{21}(x^{32} + 1) + x^{37}$	432(3)	359(2)

Authors in [Fong et al. 2004] noted that for fields represented with a trinomial reduction polynomial, $\sqrt{x}$ may be obtained at low cost directly from the reduction polynomial. Further, $\sqrt{x}$ is sparse for the NIST reduction polynomials for the fields $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{409}}$. The $m = 409$ case is especially pleasant since $\sqrt{x}$ is of degree not exceeding $(m + 1)/2$ (and so there is no reduction in the multiplication).

If the given reduction polynomial does not produce sparse $\sqrt{x}$, then we can consider mapping to another representation where roots are less expensive. This can be done at the beginning and end of the point multiplication, at a cost comparable to a few field multiplications (an insignificant amount in the overall point multiplication).

As an example, the NIST-recommend pentanomial for $m = 163$ yields a 79-term $\sqrt{x}$. This appears to demand significant effort in both hardware and software (compared with squaring). The "best" alternate representation depends on environment. Table 2 gives examples for the NIST-recommended fields with $m \in \{163, 233, 283\}$. The first line for each m is the NIST reduction polynomial, which gives low-cost reduction. In software environments, the difference in reduction cost between the polynomials in the table (for a fixed m) may be insignificant, and one can choose the representation giving the fastest square root. For example, the table entries for $m = 163$ giving 4-term $\sqrt{x}$ are both reasonably attractive in software.[2] Similarly, there are pentanomials for

[2] There are other possibilities giving 4-term roots; these where selected because the polynomial and $\sqrt{x}$ have some powers that differ by a multiple of 32 (a possible implementation advantage for software platforms).

$m = 571$ giving fast reduction and 4-term $\sqrt{x}$; in particular, the parallel algorithm can be applied effectively to all the NIST-recommended Koblitz curves (via the inexpensive conversion described earlier). In hardware platforms, the costs listed in Table 2 for $m = 233$ (where irreducible trinomials can be found) have already been reported in [Wu 2001, Rodríguez-Henríquez et al. 2006].

3.2.3 Evaluation stage

Algorithm 2 uses the scalar representation given by Solinas [Solinas 2000] and hence has the same evaluation-stage cost in terms of total point additions ($m/(\omega + 1)$ expected) as the Solinas algorithm, 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 up to a factor 2 faster than the corresponding non-parallel algorithm.

Overall acceleration will depend on overhead of parallelization, the cost of precomputation and scalar recoding, and the amount of hardware shared between the parallel threads. For example, if two point addition modules are too expensive in hardware then it is possible to share some of the units and still obtain a performance improvement. We illustrate via specific software and hardware implementations in §5.

4 Extensions

The essential features exploited by Algorithm 2 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 directly to curves defined over small fields $\mathbb{F}_q$. For example, Smart [Smart 1999] showed that there are suitable expansions in the case of (small) odd characteristic fields, and the parallelization applies provided qth roots are efficiently calculated.

Algorithm 2 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 2. As an example for Koblitz curves, a variant on Algorithm 2 discards the applications of τ^{-1} 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.[3] An analogous approach applies (although less efficiently) to supersingular curves over fields of characteristic 2 or 3, where repeated point doublings or triplings, respectively, are inexpensive.

[3] See Appendix A for a specific example of how to compute multiple instances of the Frobenius τ operator.

4.1 Interleaving

A variant of Algorithm 2 can be obtained by *interleaving* the applications of the Frobenious and inverse operators. For simplicity, we shall assume in the following that $k = \sum_{i=0}^{m-1} u_i\tau^i$ and $n = m/4$ is an integer (small adjustments are necessary for m not a multiple of 4). Write

$$kP = u^0P + u^1\tau^nP + u^2\tau^{2n}P + u^3\tau^{3n}P$$

where $u^j = \sum_{i=0}^{n-1} u_{jn+i}\tau^i$. Using properties of τ, we may write

$$kP = (u^0P + u^2\tau^{2n}P) + (\tilde{u}^3P + \tilde{u}^1\tau^{-2n}P)$$

where $\tilde{u}^j = \tau^{-n}u^j$. If our architecture possesses two independent processing units, then we can advantageously use the computational partition described next.

Let us suppose that the first processor computes the portions involving u^0 and u^2 using a "τ and add" approach as

$$R = u^0P + u^2\tau^{2n}P = u^0P + u^2Q, \tag{5}$$

whereas the second processor concurrently computes the portions involving u^3 and u^1 using a "τ^{-1} and add" procedure as

$$S = \tilde{u}^3P + \tilde{u}^1\tau^{-2n}P = \tilde{u}^3P + \tilde{u}^1Q \tag{6}$$

where the point $Q = \tau^{2n}P = \tau^{-2n}P$. Then, the scalar multiplication kP can be obtained by performing one final point addition, namely $R + S = kP$. Compared with Algorithm 2, the extra cost is the precomputation of $2^{\omega-2} - 1$ multiples of the point Q, which may be obtained at relatively low cost from the precomputation involving P.[4] As was mentioned in §3, that calculation can be done in parallel with the recoding $\omega\tau$NAF(k). In return, the strategy may provide a speedup in the evaluation stage, by calculating (5) and (6) via interleaving techniques for multi-exponentiation such as those proposed in [Möller 2001]. However, we anticipate that the performance acceleration using interleaving techniques will be modest for Koblitz curves since the computational savings will come only from performing fewer of the relatively inexpensive τ and τ^{-1} applications—the total number of point addition calculations remains the same in both approaches.

[4] Using notation as in Algorithm 2, the precomputation Q_u may be obtained as $Q_u = \tau^{2n}P_u$. Appendix A estimates that the calculation in hardware has cost less than that of two field multiplications. As an example in software, the straightforward vector-matrix multiplication in [Dahab et al. 2006, Table 2] (on a Pentium III) has cost equivalent to 1.8 to 2.4 field multiplications. Unless inversion is unusually inexpensive, these estimates imply that the precomputation for Q is obtained at less than half the cost of the precomputation for P.

4.2 Parallelization for random binary curves

Scalar multiplication based on point halving shares strategy with the τ-adic method on Koblitz curves in the sense that point doubling is replaced by a potentially faster *halving* operation that produces Q from P with $P = 2Q$. The method was proposed independently by Knudsen [Knudsen 1999] and Schroeppel [Schroeppel 2000] 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, for example, the cost of halving was estimated in the software implementation of [Fong et al. 2004] to be roughly $2M$ where M denotes the cost of a field multiplication. In hardware, that computation was also reported at a time cost of about 2 field multiplications in [Hernández-Rodríguez and Rodríguez-Henríquez 2005]. On the other hand, in López-Dahab projective coordinates the cost of a point addition and a point double is approximately $8M$ and $4M$, respectively.[5] Let the base point P have odd order N, and let t be the number of bits to represent N. For parallelization, choose $0 < n < t$ and let $\sum_{i=0}^{t} k'_i 2^i$ be the width-ω NAF of $2^n k \mod N$. Then $k \equiv \sum_{i=0}^{t} k'_i 2^{i-n} \pmod N$ and the scalar multiplication can be split as

$$kP = (k'_t 2^{t-n} + \cdots + k'_n)P + (k'_{n-1}2^{-1} + \cdots + k'_0 2^{-n})P. \tag{7}$$

The parallelization then processes the first portion of this expression by a double-and-add method, and the second portion by a halve-and-add procedure, with shared precomputation.

Let us now consider the cost for the NIST random curves using mixed affine-projective coordinates, and let us recall that halving produces a result in "lambda coordinates" that requires an extra multiplication whenever a point addition is performed. By choosing $\omega = 2$, the width-ω NAF recoding of the scalar k yields an average cost of $\frac{t}{\omega+1} = \frac{t}{3}$ point addition and t point doublings when using the standard double-and-add method for computing the scalar multiplication kP. This implies a total cost of $(4 + 8/3)t \approx 6.7t$ field multiplications. If we use a halve-and-add procedure then the computational cost is of about t point halving computations plus $\frac{t}{3}$ point additions, which is equivalent to $(2 + 9/3)t \approx 5t$ field multiplications.

On the other hand, we can use a parallel approach by choosing n to balance the costs of each portion indicated in (7), namely $(4+8/3)(t-n+1)M = (2+9/3)nM$ or $n \approx 4t/7$. In other words, the time cost of the parallel approach is equivalent to $(2 + 9/3)n = \frac{20}{7}t \approx 2.86t$ field multiplications. This is a 57% and 43% reduction in time compared with the non-parallel method based on the double-and-add method and

[5] One of four multiplications is by b. Kim and Kim [Kim and Kim 2007] provide alternate formulas with two (of four) multiplications by b, and suggest a cost estimate of $3M$ under the assumption that multiplication by b can be done at half the cost of a general multiplication. Faster doubling favors the parallelization discussed here, but we will use the more conservative estimate of $4M$.

point halving, respectively. We stress that the parallelization has been instrumented at the same high level as Algorithm 2 (so that synchronization has very low cost).

5 Implementation and timings

This section presents implementation notes and timings for Algorithm 2 in two environments. We first consider a software implementation on a common workstation with two processors. Then we examine the situation on dedicated hardware, where design constraints may limit the amount of parallel hardware that can be deployed for point multiplication.

5.1 Software implementation

In software implementations on common platforms, it is generally the case that field squarings and additions are sufficiently inexpensive relative to multiplication that their cost is ignored in rough operation counts for point multiplication. Field inversion is not prohibitive, but typically sufficiently expensive that point arithmetic is done with projective coordinates to avoid most field inversions.

The "generic method" for a square root discussed in the preceding section is less expensive than a field multiplication, but can be significant relative to a field squaring. Since the parallel algorithm uses both squaring and square roots, the algorithm looks best when these are of comparable cost or are sufficiently inexpensive that point multiplication does not heavily favor one or the other.

We consider example fields $\mathbb{F}_{2^m}$ for $m \in \{163, 233\}$. These were chosen because they are the two smallest fields in the NIST recommendation, and further because $m = 163$ illustrates the case where there is no irreducible trinomial for the field and the NIST choice of pentanomial yields a non-sparse $\sqrt{x}$. A root in this case can be roughly half the cost of a field multiplication, substantially more than a squaring. [6]

Fortunately, the "workstation" software environment is not especially sensitive to the precise form of the reduction polynomial $x^m + p(x)$, subject to the practical requirement that $\deg p$ is not too close to m. The NIST polynomials are chosen so that the powers of p are smallest in a lexicographic ordering. The candidate replacement polynomials in Table 2 are not as attractive in this sense; on the positive side, it is sometimes possible to choose polynomials with other nice features, e.g., that more powers in the polynomial and $\sqrt{x}$ differ by a multiple of the processor wordsize, and so that $\deg \sqrt{x} \leq (m+1)/2$. For our implementation, we chose the last polynomial in Table 2 for $m = 163$ and we used the NIST polynomial for $m = 233$.

[6] The smallest of the NIST fields were also chosen because they are more likely to expose any overhead penalty in the parallelization compared with the larger fields from NIST.

Table 3: Timings (in μs) for point multiplication on Koblitz curves on a Sun X4200 (dual processor Opteron 2.8 GHz). Times include cost of scalar recoding and precomputation. Methods "τ" and "τ^{-1}" indicate the operation applied at each step of the evaluation phase. The acceleration for the parallel algorithm is relative to the best time from the other methods.

	$E(\mathbb{F}_{2^{163}})$		$E(\mathbb{F}_{2^{233}})$	
Method	ω	Time	ω	Time
τ	4	166	4	349
τ^{-1}	4	168	4	345
Algorithm 2	4	103	5	187
Acceleration		38%		46%

Platform notes and timings

Algorithm 2 was implemented on a Sun X4200, a system based on AMD Opteron processors.[7] Roughly speaking, these processors are similar to those in the common Intel Pentium family, and the instruction set in the Opteron is an extension of that in the Pentium.

The particular test machine has two (single-core) Opteron running at 2.8 GHz. Coding was almost entirely in C, and the basic field and curve code was adapted directly from [Hankerson et al. 2000]. Compilation and timings were done under Solaris 10 using Sun's "Studio" compilers to produce 32-bit executables, although nothing in the code is Solaris-specific. We used POSIX threads, a portable standard in threading, and synchronization is accomplished via "spin locks" [Garg and Sharapov 2001]. A feature of Algorithm 2 is that the parallelization is at a very high level, and so it is likely that synchronization can be done with methods that are more elegant than spin locks.

Timings for Koblitz curves over $\mathbb{F}_{2^m}$ for $m \in \{163, 233\}$ are given in Table 3. These are for kP in the online-precomputation (point not known in advance) case, and the times include the cost of finding the width-ω τ-adic NAF of k. As discussed in §3, unlike the ordinary width-ω NAF, the τ-adic version requires a relatively expensive calculation to find a short ρ with $\rho \equiv k \pmod{\delta}$. We implemented Solinas' algorithms [Solinas 2000] to find ρ using the OpenSSL libraries.[8]

Computation of P_u in Algorithm 2 is done in parallel with the calculation of the τ-adic NAF. 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. Times are shown for the corresponding sequential algorithm based on the use of τ and τ^{-1}, respectively, along with the time for the parallel algorithm. The widths ω were chosen

[7] See Sun Microsystems, `http://www.sun.com`, and Advanced Micro Devices, `http://www.amd.com`.
[8] See `http://www.openssl.org`.

so that they minimize each of the timings. The savings are substantial, although less than 50%, in part because of the cost of precomputation and scalar recoding.

5.2 Hardware implementation considerations

In an effort to minimize the number of clock cycles required by Algorithm 1 when implemented in a hardware platform, we first pre-process the width-ω τNAF expansion of coefficient k as described below.

Let us recall that it is guaranteed that at most one of any consecutive ω coefficients of an $\omega\tau$NAF expansion is nonzero. Let $w_i \in \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1}-1)\}$, $1 \leq i < n$, denote each one of an average of $\lceil \frac{m}{\omega+1} \rceil$ nonzero $\omega\tau$NAF expansion coefficients. Then, the expansion would have the following structure:

$$w_0, 0 \ldots 0, w_1, 0 \ldots 0, w_2, 0, \ldots, 0, w_{i-1}, 0 \ldots 0, w_{n-1}.$$

Let $z_i \geq \omega - 1$ denote the length of the zero run that appears to the right of w_i, $1 \leq i < n$. Then, the proposed compact representation has the following form,

$$w_0, z_0, w_1, z_1, \ldots, z_{n-1}, w_{n-1}. \tag{8}$$

In this new format we just need to store in memory an average of $2\lceil \frac{m}{\omega+1} \rceil$ expansion coefficients. Given the relatively cheap cost of the field squaring operation, we may compute up to $\omega - 1$ applications of the τ Frobenius operator per cycle. This will render a valuable saving of system clock cycles as is discussed next.

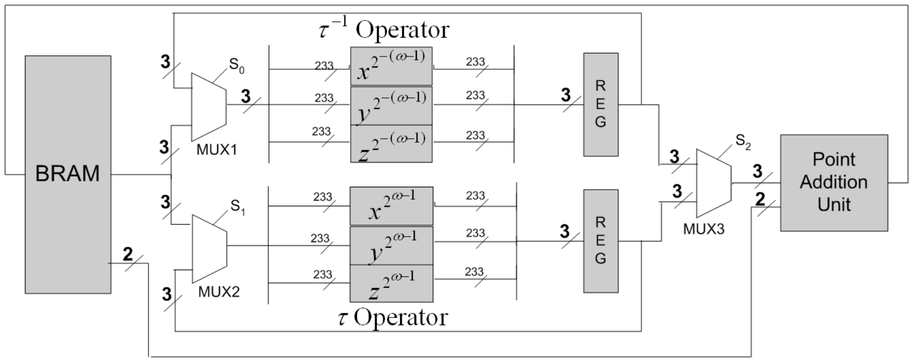

Figure 1: A hardware architecture for scalar multiplication on the NIST Koblitz curve K-233.

Hardware implementation

Ideally and as we did in the software implementation, we could implement Algorithm 2 using two point addition units with τ and τ^{-1} blocks operating separately. However, this approach may have a prohibited price due to the expensive cost of the the point addition block. A more austere strategy that uses a single point addition unit is shown in Fig. 1. That architecture is able to compute $\omega - 1$ τ and τ^{-1} applications *concurrently* with the calculations of the point addition unit. At every clock cycle our architecture computes a point addition operation and at the same time can compute $\omega - 1$ applications of τ and τ^{-1}. Intermediate results required for next stages of the algorithm are read/written in a RAM block.

The architecture shown in Fig. 1 was coded in VHDL and the basic field and curve code was adapted directly from [Cruz-Alcaraz et al. 2006]. The design was synthesized in a Virtex 2 Xilinx XC2V4000 device. The architecture was optimized for the elliptic curve NIST-233 and its corresponding $GF(2^{233})$ field arithmetic generated using the irreducible trinomial $x^{233} + x^{74} + 1$. We used $\omega = 4$; therefore we precomputed and stored in the BRAM $2^{4-2} = 4$ elliptic curve points. Table 4 shows the hardware resources required by the main modules in the architecture.

Table 4: Hardware resources required by the main design blocks.

Block	Slices	Clock
array of three $\omega - 1$ square blocks	1392	
array of three $\omega - 1$ square root blocks	1425	
Mult via Karatsuba-Offman	9588	
ECC ALU single	14091	51.7 MHz
ECC ALU parallel	15916	51.7 MHz

In order to compute one point multiplication we need to perform an average of $\frac{m}{\omega+1}$ and m point additions and τ applications, respectively. Therefore, a single point multiplication requires a total of

$$\text{\# of clock cycles} = d \cdot \frac{m}{\omega + 1} + \frac{m}{\omega - 1} \tag{9}$$

where d is the number of clock cycles required per point addition operation, ω is the window size of the expansion, m is the length of the $\omega\tau$NAF(k) expansion.[9] Notice that the cost per point addition is given by d clock cycles whereas the cost per τ application is just $1/(\omega - 1)$ clock cycles, due to the fact that our architecture is able to process up to $\omega - 1$ tau operators in a row.

[9] In the following, we do not take into consideration the cost of generating the $\omega\tau$NAF(k) expansion and precomputation.

In the case of the architecture shown in Fig. 1, since the τ or τ^{-1} operations will be computed at the same time that the point addition computation is taking place, the total number of clock cycles is of just

$$\text{\# of clock cycles} = d\frac{m}{\omega+1}. \tag{10}$$

In this work, we utilized the point addition unit design that was presented in [Cruz-Alcaraz et al. 2006], which accomplishes a point addition computation in $d = 8$ clock cycles. Hence, according with Eq. (9) and using $m = 233$ and $\omega = 4$, a straightforward implementation of Algorithm 1 yields

$$\text{\# of clock cycles} = 8 \cdot \frac{233}{4+1} + \frac{233}{4-1} = 451.$$

In the case of the parallel version of Algorithm 2, we obtain

$$\text{\# of clock cycles} = 8 \cdot \frac{233}{4+1} = 373.$$

The design can run at a clock speed of 51.7 MHz, therefore the latency on computing the scalar multiplication operation is of 8.72μs and 7.22μs, respectively, which implies a speedup of about 17%.

6 Conclusion

In this contribution we presented parallel formulations of scalar multiplication on curves over binary fields. For Koblitz curves, the main idea proposed here consists of using concurrently the τ and τ^{-1} Frobenius operators in order to parallelize known procedures for the computation of scalar multiplication under the so-called unknown point case. The method is flexible in terms of the amount of hardware dedicated, and can be implemented in hardware or software environments. For those random curves where point halving methods apply, we proposed the parallelization of the scalar multiplication computation via a procedure that concurrently applies the double-and-add and the halve-and-add procedures. Hence, our strategy can be applied to virtually all cryptographically relevant elliptic curves defined over GF(2^m).

The parallelization occurs at a very high level, an attractive feature for software implementations where the overhead of parallelization is a significant obstacle. The method for Koblitz curves is most effective when the cost of squaring and square root are comparable or sufficiently inexpensive relative to field multiplication. In particular, the method is effective on all the NIST-recommended curves [NIST, 1999] since suitable (polynomial basis) field representations can be found. Our timings on the Koblitz curves over $\mathbb{F}_{2^m}$ for $m \in \{163, 233\}$ show a 38–46% acceleration compared with a corresponding non-parallel version on a two-processor workstation, where the cost of finding the width-ω τNAF and precomputation are included.

A compact format of the $\omega\tau$NAF expansion especially tailored for hardware implementations was introduced. In this new format, typically $2\lceil \frac{m}{\omega+1} \rceil$ expansion coefficients need to be stored and processed. Furthermore, it was shown that by using as building blocks the τ and τ^{-1} Frobenius operators along with a single point addition unit, a portion of the classical double-and-add scalar multiplication algorithm can be parallelized at low cost, with an estimated acceleration of around 17% when compared with the traditional sequential version. For those designs where two point addition units are affordable, our point multiplication formulation may potentially speed the operation by a factor close to 2.

Future work therefore includes implementing Algorithm 2 using two separated point addition units in a reconfigurable hardware platform. In this way, we will be able to assess the exact impact in the performance that the parallel formulations of the Koblitz scalar multiplication presented here can yield. On the theoretical side, there are open questions of interest on the existence of reduction polynomials with special properties. In the context of the parallel algorithm presented here, we would like to choose reduction polynomials leading to both fast reduction and fast square roots, and the optimal choice will depend to some extent on environment.[10]

Our focus has been on the on-line precomputation case. The results in §5.1 show that we can parallelize at minimal speed penalty, and the techniques could be applied to common methods for known-point cases. It would be of interest to determine feasibility of parallelism much lower in the arithmetic, and the degradation in using more elegant synchronization techniques (that cooperate better with other processes).

Acknowledgments

Second and third authors wish to thank Alfred Menezes for a productive and enjoyable visit to Waterloo in August 2006. The third author acknowledges support from CONACyT through the CONACyT project number 60240.

References

[Ahmadi et al. 2007] Ahmadi, O., Hankerson, D. and Menezes, A.: "Software implementation of arithmetic in $\mathbb{F}_{3^m}$"; In Claude Carlet and Berk Sunar, editors, *International Workshop on the Arithmetic of Finite Fields (WAIFI 2007)*, volume 4547 of *Lecture Notes in Computer Science*, pages 85–102. Springer, 2007.

[Avanzi et al. 2006] Avanzi, R. M., Heuberger, C., and Prodinger, H.: "Minimality of the hamming weight of the τ-NAF for Koblitz curves and improved combination with point halving"; In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography—SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 332–344. Springer, 2006.

[Avanzi and Sica 2006] Avanzi, R. M. and Sica, F.: "Scalar multiplication on Koblitz curves using double bases"; Cryptology ePrint Archive, Report 2006/067, 2006. `http://eprint.iacr.org/`.

[10] See [Ahmadi et al. 2007] for an example in characteristic 3. Avanzi discusses the characteristic 2 case in more generality in [Avanzi 2007].

[Avanzi 2007] Avanzi, R. M.: "Another look at square roots (and other less common operations) in fields of even characteristic"; In *Selected Areas in Cryptography—SAC 2007*, Lecture Notes in Computer Science. Springer, to appear.

[Bellare 2000] Bellare, M. editor: *Advances in Cryptology—CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Blake et al. 2005] Blake, I. F., Murty, V. K. and Xu, G.: "A note on window τ-NAF algorithm"; *Inf. Process. Lett.*, 95(5):496–502, 2005.

[Cruz-Alcaraz et al. 2006] Cruz-Alcaraz, J. and Rodríguez-Henríquez, F.: "Multiplicación Escalar en Curvas de Koblitz: Arquitectura en Hardware Reconfigurable"; In *XII-IBERCHIP Workshop, IWS-2006*, pages 1–10. Iberoamerican Development Program of Science and Technology (CYTED), March 2006. In Spanish.

[Dahab et al. 2006] Dahab, R., Hankerson, D., Hu, F., Long, M., López, J. and Menezes, A.: "Software multiplication using Gaussian normal bases"; *IEEE Transactions on Computers*, 55(8):974–984, August 2006.

[Dimitrov et al. 2005] Dimitrov, V. S., Imbert, L., and Mishra, P. K.: "Efficient and secure elliptic curve point multiplication using double-base chains"; In Bimal K. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 59–78. Springer, 2005.

[Dimitrov et al. 2006] Dimitrov, V. S., Järvinen, K. U., Jacobson, M. J., Chan, W. F. and Huang, Z.: "FPGA implementation of point multiplication on Koblitz curves using Kleinian integers"; In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems—CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 445–459. Springer, 2006. ISBN 3-540-46559-6.

[Fong et al. 2004] Fong, K., Hankerson, D., López, J. and Menezes, A.: "Field inversion and point halving revisited"; *IEEE Transactions on Computers*, 53(8):1047–1059, 2004.

[Garg and Sharapov 2001] Garg, R. P. and Sharapov, I.: *"Techniques for Optimizing Applications: High Performance Computing"*; Prentice Hall PTR, 2001.

[Hasan 2001] Hasan, M. A.: "Power analysis attacks and algorithmic approaches to their countermeasures for Koblitz curve cryptosystems"; *IEEE Transactions on Computers*, 50(10): 1071–1083, 2001.

[Hankerson et al. 2000] Hankerson, D., López, J. and Menezes, A.: "Software implementation of elliptic curve cryptography over binary fields"; In Ç. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems—CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 2000.

[Hankerson et al. 2004] Hankerson, D., Menezes, A. and Vanstone, S.: *"Guide to Elliptic Cryptography"*; Springer-Verlag, New York, 2004.

[Hernández-Rodríguez and Rodríguez-Henríquez 2005] Hernández-Rodríguez, S. M. and Rodríguez-Henríquez, F.: "An FPGA arithmetic logic unit for computing scalar multiplication using the half-and-add method"; In *Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*, pages 1–7. IEEE Computer Society, 2005.

[Kim and Kim 2007] Kim, K. H. and Kim, S.I.: "A new method for speeding up arithmetic on elliptic curves over binary fields"; Cryptology ePrint Archive, Report 2007/181, 2007. `http://eprint.iacr.org/`.

[Knudsen 1999] Knudsen, E.: "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. Springer-Verlag, 1999.

[Koblitz 1992] Koblitz, N.: "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.

[Lutz 2004] Lutz, J.: "High Performance Elliptic Curve Cryptographic Co-processor"; Master's thesis, University of Waterloo, 2004.

[Möller 2001] Möller, B.: "Algorithms for multi-exponentiation"; In S. Vaudenay and A. Youssef, editors, *Selected Areas in Cryptography—SAC 2001*, volume 2259 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2001.

[NIST, 1999] 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`.

[Rodríguez-Henríquez et al. 2006] Rodríguez-Henríquez, F. Morales-Luna, G., and López-Hernández, J.: "Low Complexity Bit-Parallel Square Root Computation over GF(2^m) for all Trinomials"; Cryptology ePrint Archive, Report 2006/133, 2006. http://eprint.iacr.org/.

[Schroeppel 2000] Schroeppel, R.: "Elliptic curves: Twice as fast!"; Presentation at the CRYPTO 2000[Bellare 2000] Rump Session, 2000.

[Smart 1999] Smart, N.: "Elliptic curve cryptosystems over small fields of odd characteristic"; *Journal of Cryptology*, 12:141–151, 1999.

[Solinas 2000] Solinas, J. A.: "Efficient arithmetic on Koblitz curves"; *Designs, Codes and Cryptography*, 19(2-3):195–249, 2000.

[Wong et al. 2006] Wong, K.W., Lee, E.C.W., Cheng, L.M., and Liao, X.: "Fast elliptic scalar multiplication using new double-base chain and point halving"; Cryptology ePrint Archive, Report 2006/124, 2006. `http://eprint.iacr.org/`.

[Wu 2001] Wu, H.: "On complexity of squaring using polynomial basis in GF(2^m)"; In D. Stinson and S. Tavares, editors, *Selected Areas in Cryptography—SAC 2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 118–129. Springer-Verlag, 2001.

Appendix A: Efficient Computation of the τ and τ^{-1} Operators

The methods discussed in this paper require efficient computation of $\tau(x, y) := (x^2, y^2)$ and $\tau^{-1}(x, y) := (\sqrt{x}, \sqrt{y})$ for curve points (x, y). We assume a field representation $GF(2^m) = GF(2)[x]/(f(x))$; that is, a polynomial basis representation. In this context, field squaring and field square root are linear operations. They are particularly simple to implement when the irreducible polynomial f happens to be a trinomial of the form $x^m + x^n + 1$, with $n < m/2$.

In the rest of this Appendix, we briefly describe how to compute efficiently the τ and τ^{-1} operators as defined above, with a focus on hardware implementations.

The Frobenius τ operator

Let $A(x) = \sum_{i=0}^{m-1} a_i x^i$ for $a_i \in \{0, 1\}$ be be an arbitrary element of the field $GF(2^m)$. Then the square $C = A^2 \bmod f(x)$ in $GF(2^m)$ may be obtained by computing first the polynomial product of A by itself, followed by a reduction step modulo $f(x)$. In fact, since the characteristic of the field is 2, the square map is linear, thus the polynomial square of A is

$$A^2(x) = \left(\sum_{i=0}^{m-1} a_i x^i\right) \cdot \left(\sum_{i=0}^{m-1} a_i x^i\right) = \sum_{i=0}^{m-1} a_i x^{2i}$$

which is followed by modular reduction. For a field representation generated by a trinomial, the reduction rule is of the form $x^k = x^{k-m}x^m = x^{k-m}(1 + x^n) = x^{k-m} + x^{k-m+n}$ for $k \geq m$; in other words, the power x^k can be expressed as the addition of two lower powers whose exponents differ by n. We give next two concrete examples.

Example 1. Once again, let us consider the binary extension field $GF(2^{17})$ generated with the irreducible trinomial $f(x) = x^{17} + x^3 + 1$ used as an example in §3.1. Then, given an arbitrary field element $A \in GF(2^{17})$, the field element C satisfying $C = A^2 \bmod f(x)$ can be computed as

$$c_i = \begin{cases} a_{\frac{i}{2}} & i \text{ even}, i < 3, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+7} + a_{\frac{i}{2}+14} & i \text{ even}, 3 < i < 6, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+7} & i \text{ even}, i \geq 6, \\ a_{\frac{17+i}{2}} + a_{\frac{17+i}{2}+7} & i \text{ odd}, i < 3, \\ a_{\frac{17+i}{2}} & i \text{ odd}, i \geq 3, \end{cases} \tag{11}$$

for $i = 0, 1, \ldots, 16$. It can be verified that Eq. (11) has an associated cost of $\frac{17-1}{2} = 8$ XOR gates and $2T_x$ delays.

Example 2. Consider the binary extension field $GF(2^{233})$ generated using the irreducible trinomial $f(x) = x^{233} + x^{74} + 1$. Then, given an arbitrary field element $A \in GF(2^{233})$, the field element C satisfying $C = A^2 \bmod f(x)$ can be computed as

$$c_i = \begin{cases} a_{\frac{i}{2}} + a_{\frac{i}{2}+196} & i \text{ even}, i < 74, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+159} & i \text{ even}, 74 \leq i < 148, \\ a_{\frac{i}{2}} & i \text{ even}, i \geq 148, \\ a_{\frac{233+i}{2}} & i \text{ odd}, i < 74, \\ a_{\frac{233+i}{2}} + a_{\frac{233+i}{2}-37} & i \text{ odd}, i > 74, \end{cases} \tag{12}$$

for $i = 0, 1, \ldots, 232$. It can be verified that Eq. (12) has an associated cost of $\frac{m+n-1}{2} = 153$ XOR gates and one T_x delay.

The Frobenius τ^{-1} operator

As it has been already mentioned, the Frobenius τ^{-1} operator can be computed using field square-roots. An efficient way of computing square roots can be found by computing first field squaring. To see this, since squaring is linear, there exists a square matrix M of order $m \times m$ with coefficients in $GF(2)$ such that

$$C = A^2 = MA \tag{13}$$

which implies that computing the square root of an arbitrary field element A means finding a field element $D = \sqrt{A}$ such that $D^2 = MD = A$. Hence,

$$D = M^{-1}A \tag{14}$$

We give next three concrete examples.

Example 3. Once again, consider the binary extension field $GF(2^{17})$ generated using the irreducible trinomial $f(x) = x^{17} + x^3 + 1$ used as example in §3.1. Then, given an arbitrary field element $A \in GF(2^{17})$, the field element D satisfying $A = D^2 \bmod f(x)$ can be computed as

$$d_i = \begin{cases} a_{2i} & i < 2, \\ a_{2i} + a_{2i-3} & 2 \le i < 9, \\ a_{2i-3} + a_{2i-17} & 9 \le i < 10, \\ a_{2i-17} & 10 \le i < 17, \end{cases} \tag{15}$$

for $i = 0, 1, \ldots, 16$. It can be verified that Eq. (15) has an associated cost of $\frac{17-1}{2} = 8$ XOR gates and one T_x delay.

Example 4. Let us consider the binary extension field $GF(2^{233})$ generated using the irreducible trinomial $f(x) = x^{233} + x^{74} + 1$. Then, given an arbitrary field element $A \in GF(2^{233})$, the field element D satisfying $A = D^2 \bmod f(x)$ can be computed as

$$d_i = \begin{cases} a_{2i} + a_{2i+159} + a_{2i+85} + a_{2i+11} & i < 32, \\ a_{2i} + a_{2i+159} + a_{2i+85} + a_{2i+11} + a_{2i-63} & 32 \le i < 37, \\ a_{2i} + a_{2i+85} + a_{2i+11} + a_{2i-63} & 37 \le i < 69, \\ a_{2i} + a_{2i+85} + a_{2i+11} + a_{2i-63} + a_{2i-137} & 69 \le i < 74, \\ a_{2i} & 74 \le i < 117, \\ a_{2i-233} & 117 \le i < 154, \\ a_{2i-233} + a_{2i-307} & 154 \le i < 191, \\ a_{2i-233} + a_{2i-307} + a_{2i-381} & 191 \le i < 228, \\ a_{2i-233} + a_{2i-307} + a_{2i-381} + a_{2i-455} & 228 \le i < 233, \end{cases} \tag{16}$$

for $i = 0, 1, \ldots, 232$. Eq. (16) can be implemented with an XOR gate cost of $\frac{m+n-1}{2} = 153$ XOR gates with a $3T_x$ gate delay.

Eq. (16) can be efficiently implemented in software [Fong et al. 2004] by extracting two half-length vectors organized as, $A_{\text{even}} = (a_{m-1}, a_{m-3}, \ldots, a_2, a_0)$ and $A_{\text{odd}} = (a_{m-2}, a_{m-4}, \ldots, a_3, a_1)$ followed by the computation of

$$A^{1/2} = A_{\text{even}} + A_{\text{odd}} \left(x^{32} + x^{69} + x^{117} + x^{154} + x^{191} + x^{228} \right) \bmod f(x),$$

a relatively inexpensive operation.

Example 5. Let us consider the binary extension field $GF(2^{163})$ generated using The irreducible pentanomial $f(x) = x^{163} + x^{99} + x^{97} + x^3 + 1$. Then, given an arbitrary field element $A \in GF(2^{163})$, the field element D satisfying $A = D^2 \bmod f(x)$ can be computed as

$$d_i = \begin{cases} a_{2i} & i < 2, \\ a_{2i} + a_{2i-3} & 2 \leq i < 48, \\ a_{2i} + a_{2i-3} + a_{2i-97} & i = 49, \\ a_{2i} + a_{2i-3} + a_{2i-97} + a_{2i-99} & 50 \leq i < 82, \\ a_{2i-3} + a_{2i-97} + a_{2i-99} + a_{2i-163} & i = 82, \\ a_{2i-97} + a_{2i-99} + a_{2i-163} & 83 \leq i < 130, \\ a_{2i-99} + a_{2i-163} & i = 130, \\ a_{2i-163} & 131 \leq i < 163, \end{cases} \tag{17}$$

for $i = 0, 1, \ldots, 162$. Eq. (17) can be implemented with an XOR gate cost of 243 two-input XOR gates with a $2T_x$ gate delay.

Multiple applications of the Frobenius τ and τ^{-1} operators

Using Equations (13) and (14) one can readily compute multiple applications of the Frobenius τ and τ^{-1} operators. In the following we illustrate how to compute $n = \lceil \frac{m}{2} \rceil = 8$ Frobenius operators in the binary extension field $\mathbb{F} = GF(2^{17})$. The discussion for multiple computations of the τ^{-1} operator is omitted as it is completely analogous to the one of the τ operator.

Example 6. Once more, let us consider the binary extension field $\mathbb{F} = GF(2^{17})$ generated with the irreducible trinomial $f(x) = x^{17} + x^3 + 1$. Then, given an arbitrary field element $A \in GF(2^{17})$, the field element C satisfying $C = A^2 \bmod f(x)$ can be computed via (13) as $C = A^2 = MA$, where M is square matrix of order 17×17 given by

$$M = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1 \\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\
0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&1 \\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0 \\
0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\
0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0 \\
0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\
0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0 \\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0
\end{bmatrix}.$$

Let us say that one is interested in applying $n = \lceil \frac{m}{2} \rceil = 8$ Frobenius τ operations in a row, i.e., we want to compute τ^8. Then the matrix $M' = M^8$ can be computed as

$$M' = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&0&1&0&0&0&1&0&0&1&1&1&0\\
0&1&0&0&1&0&1&0&0&0&0&1&1&1&0&0&0\\
0&0&1&0&0&1&1&0&1&1&1&0&0&1&0&1&0\\
0&0&0&1&0&0&1&1&1&0&1&0&1&1&1&1&0\\
0&0&0&1&0&0&0&1&1&0&0&1&1&0&1&0&1\\
0&0&1&1&1&1&1&1&1&1&0&0&1&1&0&0&1\\
0&1&0&0&1&0&1&1&0&0&1&0&1&0&1&0&0\\
0&0&0&1&0&0&1&0&1&0&0&1&0&0&0&0&0\\
0&0&1&0&1&0&0&1&1&0&0&1&1&0&0&0&1\\
0&1&0&1&1&1&0&0&0&0&1&0&1&1&0&1&1\\
0&0&0&0&1&0&1&1&1&0&1&1&1&1&0&1&0\\
0&0&1&1&0&1&1&1&0&0&1&0&0&1&1&0&0\\
0&1&1&0&1&0&1&0&1&1&0&1&1&1&1&1&1\\
0&0&1&1&1&1&1&0&0&1&1&1&0&1&1&1&0\\
0&1&1&1&1&0&1&1&1&0&1&1&0&0&0&0&1\\
0&1&1&1&1&1&0&0&1&0&0&1&0&0&1&1&0
\end{bmatrix}.$$

Thus, according to (13), the field element $C = M^8 A = M'A = A^{2^8}$ can be obtained at a hardware cost of 68 two-input XOR gates with a $4T_x$ gate delay.

It is noted that the proportion of ones in the matrix M' is about $\frac{135}{17^2} \approx .47$. If the typical matrix M^n with n sufficiently large has a uniform distribution of ones and zeros, then, in a first order estimation, the hardware cost of the field arithmetic operation A^{2^n} for n large can be estimated as about $m \cdot \frac{m-1}{2}$ two-input XOR gates (less than the cost of one field multiplication).