There's something about m-ary
Fixed-Point Scalar Multiplication Protected Against Physical Attacks

Benoit Feix[1] and Vincent Verneuil[2,⋆]

[1] Underwriters Laboratories, Security Lab, Basingstoke, England
benoit.feix@ul.com
[2] NXP Semiconductors, Hamburg, Germany
vincent.verneuil@nxp.com

Abstract. In this paper, we study the fixed-point scalar multiplication operation on elliptic curves in the context of embedded devices prone to physical attacks. We propose efficient algorithms based on Yao and BGMW algorithms that are suited for embedded computing, with various storage-efficiency trade-offs. In particular, we study their security towards side-channel and fault analysis and propose a set of low-cost yet efficient countermeasures against these attacks.

Keywords: elliptic curve cryptography, scalar multiplication, embedded devices, side-channel analysis, fault analysis.

1 Introduction

Elliptic Curve Cryptography (ECC) is involved in many cryptographic protocols for signature (ECDSA), key exchange (ECDH), encryption (ECIES), etc. These protocols are used in many applications such as payment, pay-TV, transport and identity. It is thus of strong interest for the industry to improve the computation efficiency of the point scalar multiplication — the most time-consuming operation in ECC protocols.

Implementations must withstand *Side-Channel Analysis* (SCA) and *Fault Analysis* (FA). In general, the *Simple Side-Channel Analysis* (SSCA) operates the secret recovery through a single side-channel execution trace. Alternatively, *Advanced Side-Channel Analysis* (ASCA) uses multiple traces and associated data (messages, ciphertexts...) to recover the secret value through statistical processing [27]. On the other hand, FA consists in perturbing the chip activity and infer information from the possibly faulty results returned by the device [9].

The fixed base-point property of protocols relying on ElGamal or Diffie-Hellman schemes can be exploited to speed up the scalar multiplication computation. For instance, an ECDSA signature generation requires a single scalar multiplication involving a fixed base point. Yao and Pippenger showed first that

⋆ This work was initiated when both authors were with Inside Secure.

P. Goutam and P. Vaudenay (Eds.): INDOCRYPT 2013, LNCS 8250, pp. 197–214, 2013.
The final publication is available at link.springer.com.

values depending on the generator only can be precomputed to save computations during a group exponentiation or scalar multiplication [5,34].

Subsequent works have improved these methods to fit implementers needs. First, Brickell, Gordon, McCurley, and Wilson have proposed a method based on Yao's algorithm, often referred to as BGMW algorithm [11]. In an other direction, Lim and Lee presented a fixed-base comb technique [28], later improved by Tsaur and Chou [32] and by Mohamed, Hashim, and Hutter [29] in the context of elliptic curve cryptography.

In this paper, we study how simple algorithms inspired by Yao's method can be protected against physical attacks, including SSCA, ASCA and FA. All of them are suited to the context of embedded devices, since nowadays microprocessors have storage capabilities from many hundreds of kilobytes to a few megabytes, which was not the case a decade ago. Moreover, our methods are designed to ensure reasonable RAM requirements, they are thus very practical on embedded devices. In particular, we show that the point additions performed in our algorithms can be computed in a random sequence, hence providing a novel and cheap countermeasure. In addition, internal point blinding can be applied for a negligible extra cost. Finally, we propose an adaptation of a FA countermeasure proposed by Boscher, Naciri and Prouff [10] and improved by Baek [3] and Joye and Karroumi [26].

Roadmap. The paper is organized as follows. Section 2 reminds the reader of the necessary background on elliptic curve scalar multiplication and embedded security. In Section 3 we present fast fixed-point scalar multiplication algorithms and compare their cost with classical methods. In Section 4 we devise side-channel and fault analysis countermeasures for our algorithms. We also discuss their cost and compare the performances obtained with our techniques with classical ones. Finally we conclude our paper in Section 5.

2 Scalar Multiplication Background

2.1 Elliptic Curves Background

We focus on general elliptic curves defined over fields of large characteristic as they are the most used in practice. However, most of our study applies also to curves of specific shape such as Montgomery and Edwards curves, or to elliptic curves defined over binary fields.

An elliptic curve over a field $\mathbb{F}_q$ of characteristic greater than 3 is defined by an affine equation of the form:

$$y^2 = x^3 + ax + b \tag{1}$$

where a, b are elements of $\mathbb{F}_q$ such that $4a^3 + 27b^2 \neq 0$.

The set of the affine points of $\mathcal{E}$ with coordinates in $\mathbb{F}_q$, together with the *point at infinity* $\mathcal{O}$ is denoted $\mathcal{E}(\mathbb{F}_q)$. It has an abelian group structure considering

the well-known *chord and tangent* group law denoted $[+]$. The addition of two same points is generally computed using a specific *doubling* formula and is thus denoted $P\,[+]\,P = [2]\,P$ in the following.

Homogeneous ($\mathcal{H}$) and Jacobian ($\mathcal{J}$) projective coordinates are generally used to implement elliptic curve arithmetic to avoid the costly field inversion of affine ($\mathcal{A}$) point addition formulas. Jacobian coordinates offer a faster doubling than homogeneous coordinates, but the addition is more expensive.

Table 1 recalls the cost of additions, mixed affine-projective additions and doublings using homogeneous projective coordinates and Jacobian projective coordinates on prime fields of large characteristic [6]. Throughout the rest of this paper, M denotes the cost of a field multiplication, S the cost of a field squaring and A the cost of a field addition or subtraction.

Operation	Cost
$\mathcal{H} \leftarrow \mathcal{H}\,[+]\,\mathcal{H}$	$12M + 2S + 7A$
$\mathcal{H} \leftarrow \mathcal{H}\,[+]\,\mathcal{A}$	$9M + 2S + 7A$
$\mathcal{H} \leftarrow [2]\,\mathcal{H}$	$6M + 6S + 12A$
$\mathcal{H} \leftarrow [2]\,\mathcal{H}$ $(a = -3)$	$7M + 3S + 11A$

Operation	Cost
$\mathcal{J} \leftarrow \mathcal{J}\,[+]\,\mathcal{J}$	$11M + 5S + 13A$
$\mathcal{J} \leftarrow \mathcal{J}\,[+]\,\mathcal{A}$	$7M + 4S + 14A$
$\mathcal{J} \leftarrow [2]\,\mathcal{J}$	$2M + 8S + 17A$
$\mathcal{J} \leftarrow [2]\,\mathcal{J}$ $(a = -3)$	$3M + 5S + 18A$

Table 1. Cost of point operation in homogeneous and Jacobian coordinates over large-characteristic fields

2.2 Scalar Multiplication

The addition of a point P to itself d times is called the *scalar multiplication* of P by d and denoted $[d]\,P$. A well-known method to compute the scalar multiplication is the *double-and-add* algorithm. Considering $(d_{\ell-1}d_{\ell-2}\ldots d_0)_2$, the binary representation of d, the *right-to-left* version of this method relies on the following decomposition:

$$[d]\,P = \sum_{i=0}^{\ell-1} d_i \left[2^i\right] P \tag{2}$$

We focus in this study on the right-to-left version of this algorithm on which Yao's algorithm is based, whereas fixed-base comb algorithms are based on its left-to-right counterpart. This algorithm requires on average ℓ doublings and $\ell/2$ point additions.

Because a point addition $P\,[+]\,Q$ and a subtraction $P\,[-]\,Q$ have the same cost in $\mathcal{E}(\mathbb{F}_q)$, a common option to speed up the scalar multiplication consists in using a signed representation in order to decrease the number of additions to be computed.

A base b signed representation of k is $(k_{\ell_b-1}k_{\ell_b-2}\ldots k_0)$ such that:

$$k = \sum_{i=0}^{\ell_b-1} k_i b^i \quad \text{with} \quad |k_i| < b \tag{3}$$

Among them the binary *Non-Adjacent Form* (NAF) is defined as follows. The NAF representation of a positive non-zero integer k is $(k_{\ell-1}k_{\ell-2}\dots k_0)_{\mathrm{NAF}}$ with $k_i \in \{-1, 0, 1\}$, $0 \leq i < \ell - 1$ and $k_{\ell-1} = 1$, such that for all pairs of consecutive digits, at least one of them is zero. As a consequence, the number of non-zero digits of ℓ-digit NAF representations is approximately $\ell/3$.

The right-to-left double-and-add scalar multiplication algorithm using the NAF representation is presented in Alg. 2.1. Compared to the binary algorithm, it requires only ℓ doublings and $\ell/3$ point additions on average, thus saving $\ell/6$ point additions.

Alg. 2.1 Right-to-left NAF double-and-add scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, ℓ-NAF-digit scalar $d = (d_{\ell-1}d_{\ell-2}\dots d_0)_{\mathrm{NAF}}$
Output: $Q = [d]\,P$
1: $Q \leftarrow \mathcal{O}$
2: $R \leftarrow P$
3: **for** $i = 0$ **to** $\ell - 1$ **do**
4: **if** $d_i = 1$ **then**
5: $Q \leftarrow Q\,[+]\,R$
6: **if** $d_i = -1$ **then**
7: $Q \leftarrow Q\,[-]\,R$
8: $R \leftarrow [2]\,R$
9: **return** Q

Remark. For a sake of simplicity, the binary length of d and the length of its signed representation — which may differ by 1 — are both denoted ℓ through the rest of this paper.

2.3 Side-Channel Analysis and Countermeasures

Countering SSCA on scalar multiplication can be achieved using regular implementations [25]. Considering ASCA, most countermeasures stem from Coron's propositions [14]: scalar blinding, projective coordinates randomization and input point blinding. Among them, the multiplicative blinding of the projective coordinates requires the lowest computational overhead but does not protect against chosen input-point attacks [19,1]. It is then necessary to use extra countermeasures such as additive point randomization [14,1] or scalar blinding, which are much more expensive.

Another category of attacks inspired by the Big Mac attack from Walter [33] has been recently extended to several other attacks referred to as *horizontal* techniques [13,4]. They are more difficult to mount in practice than classical ASCA but necessitate only one side-channel trace, contrary to classical ASCA.

To thwart FA on scalar multiplication, Biehl, Meyer, and Müller proposed that implementations verify that the output point of a computation belongs to

the curve [7]. Ciet and Joye advise to check the curves parameters also [12]. Nevertheless Blömer, Otto, and Seifert have proven that such countermeasures are circumvented by a so-called *sign change* attack which takes advantage of the signed NAF representation [8]. To provide a robust protection against fault attacks, *self-secure* algorithms [18,10,3,26] detect a fault injected during the execution of the scalar multiplication by checking an invariant property on the manipulated variables during or at the end of the computation.

Recently, Fan, Gierlichs, and Vercauteren presented a combined FA and SSCA [16]. A well-chosen input point and a single fault injection lead to the manipulation of the point at infinity during the scalar multiplication, which can be observed by SSCA. Input point blinding is thus necessary to thwart this attack.

2.4 Fixed-Point Scalar Multiplication Methods

Assuming that $(d_{v-1}d_{v-2}\dots d_0)_h$ is the base-h representation of d, the BGMW [11] algorithm relies on the following decomposition:

$$[d]\,P = \sum_{i=1}^{h-1} i \sum_{d_j=i} \left[h^j\right] P \tag{4}$$

If all $\left[h^j\right] P$, such that $0 \leq j \leq v-1$, are precomputed, Alg. 2.2 computes $[d]\,P$ using in average $v(h-1)/h - 1$ point additions[3] in the inner loop (step 5) and $h-2$ additions in the main loop (step 6).

Alg. 2.2 BGMW fixed-point scalar multiplication

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $d = (d_{v-1}d_{v-2}\dots d_0)_h$, $\left\{P_j = \left[h^j\right] P,\ \text{for } j \in [0, v-1]\right\}$
Output: $Q = [d]\,P$

1: $Q, R \leftarrow \mathcal{O}$
2: **for** $i = h-1$ **to** 1 **by** -1 **do**
3: **for** $j = 0$ **to** $v-1$ **do**
4: **if** $d_j = i$ **then**
5: $Q \leftarrow Q\,[+]\,P_j$
6: $R \leftarrow R\,[+]\,Q$
7: **return** R

Other fixed-point scalar multiplication techniques are generally derived from the Lim and Lee comb method [28] which uses more precomputations but provides a better efficiency. For instance, Tsaur and Chou [32] and Mohamed et al. [29] improve this method by using signed representations. Besides, Hedabou, Pinel and Bénéteau [21] show that comb algorithms can be rendered regular to counter SSCA by recoding the scalar and propose a point blinding method

[3] The initial point addition with $\mathcal{O}$ is free.

against ASCA. Unfortunately, this method does not allow to refresh the blinding point without recomputing the whole array of precomputed points, which is an issue in practice. Finally, we are not aware of any work addressing the protection of these algorithms against FA.

3 Revisiting the BGMW Algorithm

Following Brickell et al. observation [11], the sequence of intermediate results stored in R in Alg. 2.1 does not depend on the scalar:

$$R \leftarrow P \leftarrow [2]\,P \leftarrow \cdots \leftarrow \left[2^{\ell-1}\right] P$$

We propose in this section to rewrite the BGMW algorithm to make use of signed representations of the scalar and benefit of the lower number of non-zero digits thereof.

3.1 NAF Scalar Multiplication

Let T refer to an ℓ-point pre-computation table such that:

$$T_i = \left[2^i\right] P \qquad \text{with} \qquad 0 \leq i \leq \ell - 1$$

With d_i the binary or NAF digits of d, the scalar multiplication can then be written as:

$$[d]\,P = \sum_{i=0}^{\ell-1} [d_i]\,T_i$$

In this manner, the computation complexity drops to $\ell/3$ point additions, see below Alg. 3.1. On the other hand, $2(\ell-1)\lceil \ell/8 \rceil$ bytes in memory are required if the additional $\ell - 1$ points are stored in affine coordinates.

Alg. 3.1 Add-only NAF scalar multiplication using precomputations

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, ℓ-NAF-digit scalar $d = (d_{\ell-1} d_{\ell-2} \dots d_0)_{\text{NAF}}$, $T = \left(\left[2^i\right] P\right)_{0 \leq i \leq \ell-1}$
Output: $Q = [d]\,P$

```
1: Q ← O
2: for i = 0 to ℓ − 1 do
3:     if d_i = 1 then
4:         Q ← Q [+] T_i
5:     if d_i = −1 then
6:         Q ← Q [−] T_i
7: return Q
```

The storage requirements of this method amounts to 16 kB for a 256-bit elliptic curve and 64 kB for a 512-bit curve. In the next section, we discuss using window methods to improve both efficiency and memory requirements.

3.2 Application to m-ary NAF Scalar Multiplication

Window algorithms are well-known to improve scalar multiplication efficiency by reducing the number of point additions in left-to-right algorithms [20]. Yet less widely known among implementers, window methods apply to right-to-left algorithms also [30,34]. Considering an m-ary scalar multiplication algorithm, with $m = 2^t$, this strategy generally requires $m-1$ point registers instead of one register Q to perform a general m-ary scalar multiplication — thus it requires more RAM —, and an additional computation phase at the end of the main loop known as *aggregation*.

On the other hand, only one precomputed point out of t is required, therefore the size of T drops ℓ to $\ell' = \lceil \ell/t \rceil$. Consequently, the precomputed values are:

$$T_i = \left[2^{ti}\right] P \quad \text{with} \quad 0 \leq i \leq \ell' - 1$$

We consider in this section an ℓ-NAF-digit integer which digits are scanned using a fixed window of size t. As two consecutive NAF digits cannot be both non-zero, the set of possible integers coded by t NAF digits is:

$$\begin{aligned}
&\{0, \pm 1\} \text{ for } t = 1,\\
&\{0, \pm 1, \pm 2\} \text{ for } t = 2,\\
&\{0, \pm 1, \pm 2, \pm 3, \pm 4, \pm 5\} \text{ for } t = 3,\\
&\{0, \pm 1, \pm 2, \ldots, \pm \nu(t)\} \text{ for any } t \geq 1,\\
&\text{with } \nu(t) = \frac{2^{t+1} - \frac{3+(-1)^t}{2}}{3}.
\end{aligned}$$

The m-ary NAF scalar multiplication strategy requires that $\nu(t)$ registers R_j, $j \in \{1, 2, \ldots, \nu(t)\}$ are available to store the internal sums of points $\left[2^{ti}\right] P$ such that the i^{th} scanned window holds the value j. We obtain the final result by computing the aggregation:

$$\sum_{i=1}^{\nu(t)} [i] R_i$$

Our fixed-point scalar multiplication using the m-ary NAF scalar scanning is presented in Alg. 3.2. The average number of point additions performed in the main loop is:

$$\left\lceil \frac{\ell}{t} \right\rceil \left(1 - \left(\frac{2}{3}\right)^t\right)$$

The aggregation is computed using the efficient technique from Joye and Karroumi [26]. It requires $2(\nu(t) - 1)$ additional additions.

It is worth noticing that the storage requirement and the number of additions performed in the main loop decrease when t increases. On the other hand, the cost of the aggregation rises rapidly with t. The optimal value for t thus depends on ℓ, as discussed in Section 3.4.

Alg. 3.2 Add-only m-ary NAF scalar multiplication using precomputations

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, ℓ-NAF-digit scalar $d = (d_{\ell-1} d_{\ell-2} \dots d_0)_{\mathrm{NAF}}$, $T = \left(\left[2^{ti} \right] P \right)_{0 \le i \le \ell'-1}$
Output: $Q = [d]\, P$

`Initialization`
1: $R_1 \leftarrow \mathcal{O}$, $R_2 \leftarrow \mathcal{O}$, $\dots$, $R_{\nu(t)} \leftarrow \mathcal{O}$
`Main loop`
2: **for** $i = 0$ **to** $\ell' - 1$ **do**
3: $\quad u = \sum_{j=0}^{t-1} 2^j d_{ti+j}$ [Assume $d_i = 0$ if $i \ge \ell$]
4: $\quad$ **if** $u > 0$ **then**
5: $\quad\quad R_u \leftarrow R_u\, [+]\, T_i$
6: $\quad$ **if** $u < 0$ **then**
7: $\quad\quad R_{-u} \leftarrow R_{-u}\, [-]\, T_i$
`Aggregation`
8: **for** $i = \nu(t) - 1$ **to** 1 **by** -1 **do**
9: $\quad R_i \leftarrow R_i\, [+]\, R_{i+1}$
10: $\quad R_{\nu(t)} \leftarrow R_{\nu(t)}\, [+]\, R_i$
11: **return** $R_{\nu(t)}$

On-the-fly Table Computation. Although we focus on the context where the base point is known in advance and the precomputations table can be calculated off-line — this is the context of ECDSA signature, where the input point is a generator defined in the public parameters —, we can also consider the case of an unknown base point that will be used for several scalar multiplications. This may be the case in applications such as ECDSA verification, ECDH key exchange, ECIES decryption, etc.

In such a case, one may run a first scalar multiplication using the classical right-to-left algorithm, and populates on-the-fly the table T. However, points T_i will likely be computed in projective coordinates which involves general additions during further scalar multiplications instead of mixed affine-projective additions.

An option is thus to convert these points to affine coordinates at the end of the first scalar multiplication using Montgomery's trick for multiple inversions [31]. It requires expensive computations to be put in balance with the gain obtained in further scalar multiplications.

3.3 Application to Width-w NAF Scalar Multiplication

We recall the *width-w* NAF representation [20]: given an integer $w \ge 2$, the width-w NAF representation of a non-zero positive integer k, denoted $(k)_{\mathrm{NAF}_w}$ is a base 2^{w-1} signed representation[4], cf. expression (3) and the aggregation consists in computing:

$$\sum_{i=0}^{2^{w-2}-1} [2i+1]\, R_{2i+1}$$

[4] In particular, $(k)_{\mathrm{NAF}} = (k)_{\mathrm{NAF}_2}$ for any positive integer k.

Algorithm 3.3 presents our fixed-point scalar multiplication method using the width-w representation. The number of point additions performed in the main loop is $\ell/(w+1)$ in the average. The aggregation is computed using an efficient technique inspired by Joye and Karroumi [26] and requires $3(2^{w-2}-1)$ additions.

Alg. 3.3 Add-only width-w NAF scalar multiplication using precomputations

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, $d = (d_{\ell-1} d_{\ell-2} \dots d_0)_{\mathrm{NAF}_w}$, $T = \left([2^i] P \right)_{0 \le i \le \ell-1}$
Output: $Q = [d] P$

`Initialization`
1: $R_1 \leftarrow \mathcal{O}$, $R_3 \leftarrow \mathcal{O}$, $\dots$, $R_{2^{w-1}-1} \leftarrow \mathcal{O}$
`Main loop`
2: **for** $i = 0$ **to** $\ell - 1$ **do**
3: **if** $d_i > 0$ **then**
4: $R_{d_i} \leftarrow R_{d_i} [+] T_i$
5: **if** $d_i < 0$ **then**
6: $R_{-d_i} \leftarrow R_{-d_i} [-] T_i$
`Aggregation`
7: **for** $i = 2^{w-1} - 3$ **to** 1 **by** -2 **do**
8: $R_i \leftarrow R_i [+] R_{i+2}$
9: **if** $i = 2^{w-1} - 3$ **and** $i \neq 1$ **then** [First loop iteration, $w \neq 3$]
10: $R_{2^{w-1}-1} \leftarrow [2] R_{2^{w-1}-1} [+] [2] R_i$
11: **if** $i \neq 2^{w-1} - 3$ **and** $i \neq 1$ **then** [Not first/last loop iteration]
12: $R_{2^{w-1}-1} \leftarrow R_{2^{w-1}-1} [+] [2] R_i$
13: **if** $i \neq 2^{w-1} - 3$ **and** $i = 1$ **then** [Last loop iteration, $w \neq 3$]
14: $R_{2^{w-1}-1} \leftarrow R_{2^{w-1}-1} [+] R_i$
15: **if** $i = 2^{w-1} - 3$ **and** $i = 1$ **then** [Single loop iteration ($w = 3$)]
16: $R_{2^{w-1}-1} \leftarrow [2] R_{2^{w-1}-1} [+] R_i$
17: **return** $R_{2^{w-1}-1}$

This methods requires less registers than Alg. 3.2 as only odd digits appear in the width-w NAF representation. On the other hand, it requires the full ℓ-point precomputed table T.

Remark. The width parameter w allow to control the aggregation cost vs. main loop speed-up trade-off. Intermediate trade-offs can be obtained using sliding-window NAF techniques [20] for instance, or, more generally, *fractional* window techniques [30].

3.4 Efficiency and Storage Requirements Analysis

We study hereafter the efficiency and storage (RAM & non-volatile memory) requirements of the previous fixed-point scalar multiplication methods.

Coordinates Choice. Since our algorithms involve point additions only, the best representation for registers R_i is the homogeneous projective coordinates. All additions performed in the main loop use a mixed affine-projective formula, as precomputed points T_i are stored in affine coordinates. Only the aggregation stage requires general additions.

As a matter of reference, we provide the cost of a classical left-to-right NAF double-and-add algorithm, denoted *LR-NAF-DA*. It is computed assuming the use of Jacobian coordinates, mixed affine-projective additions and $a = -3$ to provide a fair comparison. We compare our algorithms with BGMW Alg. 2.2 with $h = 2^t$ for t=2, 3, 4, 5, assuming the use of homogeneous projective coordinates and mixed affine-projective additions when possible.

Efficiency Comparison. Costs are expressed in field multiplications M, assuming $S/M = 0.8$ and $A/M = 0$. The storage requirement corresponds to the size of table T for our algorithms and to the size of the array $\{P_j\}$ for the BGMW algorithm. The RAM estimation is based on the number of registers used in algorithms — i.e. it does not include the RAM required to compute point operations.

We provide in Fig. 1 a graphical comparison of the cost per scalar bit of our fixed base-point scalar multiplication algorithms depending on ℓ for common key lengths.

Algorithm	$\ell = 256$			$\ell = 512$		
	Cost	Storage	RAM	Cost	Storage	RAM
LR-NAF-DA	2662	0	160	5324	0	320
BGMW Alg. 2.2 $h = 4$	1034	8	256	2052	32	512
BGMW Alg. 2.2 $h = 8$	862	5.3	256	1654	21.3	512
BGMW Alg. 2.2 $h = 16$	**816**	4	256	1452	16	512
BGMW Alg. 2.2 $h = 32$	923	3.2	256	**1449**	12.8	512
Alg. 3.1	901	16	160	1806	64	320
Alg. 3.2, $t = 2$	775	8	256	1529	32	512
Alg. 3.2, $t = 3$	**743**	5.3	544	1377	21.3	1088
Alg. 3.2, $t = 4$	781	4	1024	**1325**	16	2048
Alg. 3.3, $w = 3$	717	16	256	1395	64	512
Alg. 3.3, $w = 4$	**663**	16	448	1206	64	896
Alg. 3.3, $w = 5$	736	16	832	**1188**	64	1664

Table 2. Comparison of scalar multiplication algorithms cost (in M), storage requirement (in kB), and used RAM (in B) for 256 and 512-bit elliptic curves over large characteristic fields, assuming $a = -3$

Non-Volatile Memory Transfers. We consider in Table 2 arithmetic operations only. On embedded devices, the comparison must also takes into account

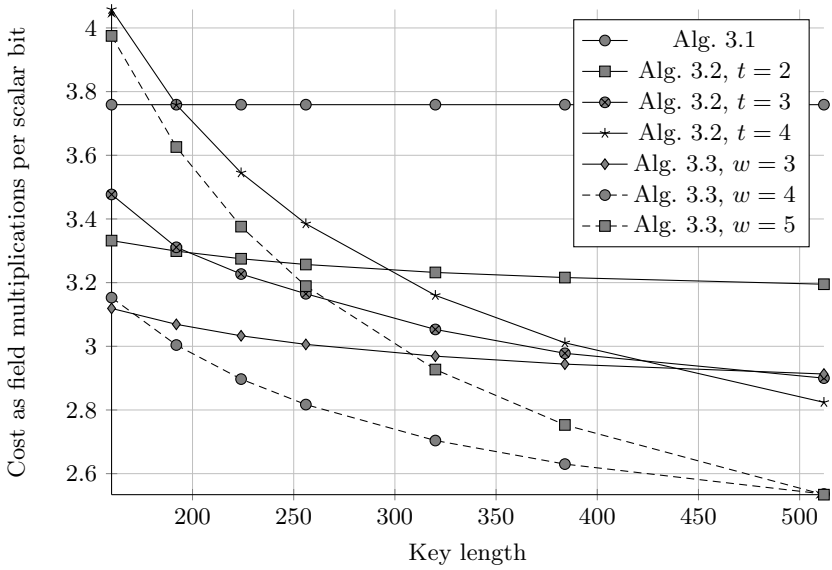

Fig. 1. Comparison of fixed-point scalar multiplication algorithms cost depending on the key length

the numerous transfers from non-volatile memory to RAM performed in algorithms, but such costs highly depends on devices (frequency, bus width, etc.) To estimate the practical expected overhead, we take as an example an 8-bit AVR chip provided with the Ad-X coprocessor, both running at 30 MHz. Considering Alg. 3.2 with $t = 2$ and $\ell = 256$, the transfer of 8kB of EEPROM to RAM takes about 5 % of the full computation time. This is far from negligible, but makes our method still very attractive. In addition, the overhead would be much smaller for higher window sizes, for 16 or 32-bit devices, or if transfers can be performed while the coprocessor is running.

Results Analysis. Considering Fig. 1, the best choice for an efficient and practical implementation is the m-ary NAF add-only scalar multiplication with $t = 3$ or 4, depending on the key length.

The better efficiency of our algorithms over the classical BGMW method, cf. Table 2, is due to the use of signed representations. On the opposite, the strategy of using separate registers and a final aggregation requires a few more computations and more RAM than the strategy applied in the BGMW algorithm. However, we will see in the next section that our choices allow very efficient countermeasures.

4 Side-Channel Analysis Countermeasures

We discuss in this section the means to protect our fixed-point algorithms against side-channel and fault attacks.

4.1 Simple Analysis Countermeasure

Algorithms 3.1, 3.2, and 3.3 are obviously subject to the simple analysis if the conditional branches of the main loop can be observed by an attacker. We show in the following how this source of leakage can be removed using an highly regular algorithm.

Highly Regular Addition Loop. Highly regular algorithms such as the Montgomery ladder [24] are known to perform the scalar multiplication — or the exponentiation in a multiplicative group — without any dummy operation and thus provide a protection against a wide range of attacks [25,26].

A straightforward adaptation of the m-ary method can be obtained as described in Alg. 4.1. It requires m registers instead of $\nu(t)$ compared to Alg. 3.2.

Alg. 4.1 Regular add-only m-ary scalar multiplication using precomputations

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, ℓ-bit scalar $d = (d_{\ell-1} d_{\ell-2} \dots d_0)_2$, $T = \left([2^{ti}] P \right)_{0 \le i \le \ell'-1}$
Output: $Q = [d] P$

`Initialization`
1: $R_0 \leftarrow \mathcal{O}$, $R_1 \leftarrow \mathcal{O}$, $\dots$, $R_{m-1} \leftarrow \mathcal{O}$
`Main loop`
2: **for** $i = 0$ **to** $\ell' - 1$ **do**
3: $\quad u = \sum_{j=0}^{t-1} d_{ti+j} 2^j$ [Assume $d_i = 0$ if $i \ge \ell$]
4: $\quad R_u \leftarrow R_u \,[+]\, T_i$
`Aggregation`
5: **for** $i = m - 2$ **to** 1 **by** -1 **do**
6: $\quad R_i \leftarrow R_i \,[+]\, R_{i+1}$
7: $\quad R_{m-1} \leftarrow R_{m-1} \,[+]\, R_i$
8: **return** R_{m-1}

This algorithm requires exactly ℓ' point additions in the main loop. It is worth noticing that the overhead introduced by this method over Alg. 3.2 tends to 0 as t grows. Note also that the NAF representation does not help reducing the number of performed additions here, as zero digits are now treated in the same way as other digits.

Efficiency Analysis. Table 3 gives the precise cost of Alg.4.1 assuming the use of homogeneous coordinates and mixed affine-projective additions in the main loop. Under our assumptions, the window width minimizing the number of field operations is $t = 3$ for $\ell \le 224$, and $t = 4$ for $\ell \ge 256$.

Algorithm	$\ell = 256$			$\ell = 512$		
	Cost	Storage	RAM	Cost	Storage	RAM
Alg. 4.1, $t = 1$	2713	16	256	5427	64	512
Alg. 4.1, $t = 2$	1411	8	448	2768	32	896
Alg. 4.1, $t = 3$	1075	5.3	832	1976	21.3	1664
Alg. 4.1, $t = 4$	**1059**	4	1600	**1738**	16	3200
Alg. 4.1, $t = 5$	1367	3.2	3136	1908	12.8	6272

Table 3. Cost of Alg. 4.1 (in M), storage requirement (in kB), and used RAM (in B) for 256 and 512-bit elliptic curves over large characteristic fields

4.2 Advanced Analysis Countermeasures

Let us now focus on the protection of our algorithms towards ASCA. Commonly used countermeasures consist in randomizing the projective coordinates of points, blinding the scalar with a random multiple of the subgroup order and input point blinding.

While the projective coordinates randomization is generally a low-cost countermeasure, it would imply a non-negligible overhead here. Indeed, each point T_i of the precomputed table should have its coordinates randomized, which in turn requires a general addition, instead of a mixed affine-projective one.

The scalar blinding countermeasure $d^* \leftarrow d + rn$, with r a random nonce and n the order of the generator point P, has two drawbacks: first it induces an overhead of $(|r| + \ell)/\ell$, second its efficiency is uncertain when n has a particular form as with NIST prime curves [17].

We present in the rest of this section two ASCA countermeasures ensuring high protection and little overhead for the fixed-point scalar multiplication algorithms.

Shuffling the Sequence of Point Additions. As the main loop of our algorithms performs additions only, and as points to be added are all stored in a table, the sequence of additions can be performed in any order. Not only it can be processed in left-to-right direction as well as right-to-left, but it is even possible to pick the points T_i to be added in a random order[5].

Considering our proposed highly regular algorithm, a random permutation σ of $\{0, 1, \ldots, \ell' - 1\}$ can be generated and used in a similar way to shuffle the iterations of the main loop. This solution is detailed at the end of the section in Alg. 4.2.

The extra cost brought by this countermeasure lies principally in the generation of the random permutation. Generating efficiently a random permutation is an issue that merits a whole study by itself. As it is not the core subject of this paper, we suggest to use the method proposed by Coron [15,4] for generating a pseudo-random permutation.

[5] Permutations of the points added in Yao's algorithm already appear in a paper by Avanzi [2] for efficiency purpose.

Internal Point Blinding. Assuming a random point R lying on the curve, a naïve blinding of the form $[d](P[+]R)[-][d]R$ requires two scalar multiplications instead of one, which is an overkill for most applications. In specific cases such as fixed-scalar multiplication, efficient point blinding is possible as shown by Coron [14]. Itoh, Izu, and Takenaka have also proposed a modified left-to-right algorithm using internal point blinding [22,23].

Due to the specific structure of our algorithms the countermeasure is straightforward to apply. For instance, considering Alg. 3.1, initialize Q to some random point R from $\mathcal{E}(\mathbb{F}_q)$ at step 1 and subtract this point from Q before the return statement. Thus the point blinding requires only an extra point addition.

For m-ary algorithms using accumulators R_1, R_2,..., R_q, $q > 1$, we propose the following strategy to keep a low overhead. Find a sequence of integers $(e_i)_{1 \le i \le q} \in \{-1, 1\}^q$ such that:

$$\sum_{i=1}^{q} i \cdot e_i \in \{0, 1\}$$

Then, modifying the initialization step of the algorithm to $R_i \leftarrow [e_i]R$, for $1 \le i \le q$, yields $R_q = [d]P$ or $R_q = [d]P[+]R$ at the end of the aggregation step. In the first case no extra operation is required to remove the mask, and a single subtraction by R has to be performed in the second case.

Our experimentations show that a sequence (e_i) with $i \cdot e_i$ summing to 1 can be easily found if $q = \nu(t)$ — i.e. using the m-ary NAF algorithm — or summing to 0 if $q = 2^t - 1$ — i.e. using the highly regular m-ary algorithm[6]. For instance:

$$(e_i) = (-1, 1) \text{ for } q = 2,$$
$$(e_i) = (-1, -1, 1) \text{ for } q = 3,$$
$$(e_i) = (1, -1, 1, 1, -1) \text{ for } q = 5.$$
$$(e_i) = (-1, 1, -1, -1, 1, -1, 1) \text{ for } q = 7.$$

Random Point Generation. Generating a random point on the curve may be an overkill in practice as it requires a square root computation in $\mathbb{F}_q$. The following strategy can thus be applied: a random point R is generated once for all offline before any computation and stored in non-volatile memory together with T. After each scalar multiplication, R is updated as follows:

for $i = 1$ **to** r **do**
  Pick at random $j \in \{0, 1, \ldots, \ell' - 1\}$
  Pick a random bit b
  $R \leftarrow R[+]\left[(-1)^b\right]T_j$

where r is a security parameter depending on the size of T and on the required security level.

Randomizing the projective coordinates of R before each scalar multiplication provides an additional level of blinding at a very low cost.

[6] In this case, R_0 can be initialized to R as it has no consequence on the result.

4.3 Fault Analysis Countermeasures

We finally consider the protection of our scalar multiplication algorithms against FA.

Following an observation by Joye and Karroumi [26], register R_1 holds the value $\left[\sum_{i=1}^{m-1}\right] R_i^*$ at the end of the aggregation, where R_i^* is the value stored in R_i prior to the aggregation. It follows that, after the aggregation, $R_0 + R_1 = \left[\sum_{i=0}^{\ell'-1}\right] T_i$. We can use this invariant to check the computation consistency.

Therefore, we propose to store an extra point S, together with the table T, equal to the sum of all the points in T. The fault detection stage then requires only a point addition $R_0 \,[+]\, R_1$ after the aggregation and a comparison with S.

If the point blinding countermeasure presented in the last section is used, an extra subtraction is required to unmask $R_0 \,[+]\, R_1$. Alternatively, one may initialize R_0 to $\left[-\sum_{i=1}^{m-1} e_i\right] R$, such that no extra addition at all is required in the fault verification stage.

We present in Alg. 4.2 an updated version of the highly regular m-ary algorithm with the main loop shuffling and point blinding countermeasures presented in the previous section, and the present fault detection method.

Alg. 4.2 Regular Add-only m-ary scalar multiplication using precomputations with ASCA and FA countermeasures

---

Input: $P \in \mathcal{E}(\mathbb{F}_q)$, ℓ-bit scalar $d = (d_{\ell-1} d_{\ell-2} \dots d_0)_2$, $T = \left(\left[2^{ti}\right] P\right)_{0 \le i \le \ell'-1}$, $S = \sum_{i=0}^{\ell'-1} T_i$, $(e_i)_{1 \le i \le t-1}$ s.t. $\sum_{i=1}^{m-1} i \cdot e_i = 0$ and $\sum_{i=1}^{m-1} e_i = -1$
Output: $Q = [d]\, P$

`Initialization`
1: Generate a random permutation $\sigma = (\sigma_0 \dots \sigma_{\ell'-1})$ of $[0, \dots, \ell'-1]$
2: Generate a random point $R \in \mathcal{E}(\mathbb{F}_q) \setminus \mathcal{O}$
3: $R_0 \leftarrow R$, $R_1 \leftarrow [e_1]\, R$, …, $R_{m-1} \leftarrow [e_{m-1}]\, R$
`Main loop`
4: **for** $i = 0$ **to** $\ell' - 1$ **do**
5: $\quad u = \sum_{j=0}^{t-1} 2^j d_{t\sigma_i + j}$ [Assume $d_i = 0$ if $i \ge \ell$]
6: $\quad R_u \leftarrow R_u \,[+]\, T_{\sigma_i}$
`Aggregation`
7: **for** $i = m - 2$ **to** 1 **by** -1 **do**
8: $\quad R_i \leftarrow R_i \,[+]\, R_{i+1}$
9: $\quad R_{m-1} \leftarrow R_{m-1} \,[+]\, R_i$
`Fault detection`
10: $R_0 \leftarrow R_0 \,[+]\, R_1$
11: **if** $R_0 \neq S$ **then**
12: $\quad$ **return** FAULT DETECTED
13: **return** R_{m-1}

4.4 Security Analysis

The regular structure of Alg. 4.2 provides a standard protection against classical SSCA, i.e. against attacks targeting conditional branches leakages. The additive blinding of internal point registers thwarts classical ASCA that requires the knowledge of manipulated data. It also counteracts chosen input-point attacks when the attacker controls the value of the internal registers. Shuffling the sequence of point additions adds an additional layer of protection against ASCA by increasing the number of required traces by a factor of ℓ'.

Let us now consider the attacks based on potential noticeable additions involving the point at infinity in the main loop of the scalar multiplication. Obviously no point in T should be the point at infinity and its integrity should further be checked. As specified in Alg. 4.2, the pseudo-random generation of R should also verify that $R \neq \mathcal{O}$. The possibility that the point at infinity appears in one computation of the main loop is thus very unlikely considering the size of the groups. Due to the masking and shuffling countermeasures, even in this case, an attacker would not be able to identify the addition or the register involved.

Finally, as no value nor any result is ever discarded during the computation, the fault countermeasure ensures that a perturbation introduced in any point addition or any point register would be detected and no result returned.

5 Conclusion

We propose in this paper fixed-point scalar multiplication algorithms derived from Yao and BGMW algorithms. These algorithms benefit from the use of signed representations for the scalar and have a small code size footprint. They can be used in embedded devices provided with a few dozen kilobytes of storage and a few kilobytes of RAM. We propose a novel countermeasure towards side-channel analysis with a very low cost and adapt some others to our algorithms. We present a combination of them and hence provide our regular m-ary algorithm with state-of-the-art protection against SSCA, ASCA and FA. The question of comparing the efficiency, storage requirement, and protection against physical attacks of our methods with fixed-base comb algorithms is left open for further research.

Acknowledgements

The authors would like to thank Christophe Clavier for his support and fruitful discussions. We also thank the anonymous referees for their valuable comments.

References

1. T. Akishita and T. Takagi. Zero-Value Point Attacks on Elliptic Curve Cryptosystem. In C. Boyd and W. Mao, editors, *ISC*, volume 2851 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2003.

2. Roberto Maria Avanzi. Delaying and Merging Operations in Scalar Multiplication: Applications to Curve-Based Cryptosystems In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 203–219. Springer, 2006.
3. Yoo-Jin Baek. Regular 2^w-ary right-to-left exponentiation algorithm with very efficient DPA and FA countermeasures. *Int. J. Inf. Sec.*, 9(5):363–370, 2010.
4. Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, and Justine Wild. Horizontal and Vertical Side-Channel Attacks against Secure RSA Implementations. In Ed Dawson, editor, *CT-RSA*, volume 7779 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.
5. Daniel J. Bernstein. Pippenger's exponentiation algorithm, 01 2002. To be incorporated into author's High-speed cryptography book. `http://cr.yp.to/papers.html#pippenger`.
6. Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. `http://www.hyperelliptic.org/EFD`.
7. I. Biehl, B. Meyer, and V. Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In Mihir Bellare, editor, *CRYPTO*, volume 1880 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2000.
8. J. Blömer, M. Otto, and J-P. Seifert. Sign Change Fault Attacks on Elliptic Curve Cryptosystems. In L. Breveglieri, I. Koren, D. Naccache, and J-P. Seifert, editors, *FDTC*, volume 4236 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2006.
9. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *EUROCRYPT*, pages 37–51, 1997.
10. A. Boscher, R. Naciri, and E. Prouff. CRT RSA Algorithm Protected Against Fault Attacks. In D. Sauveron, C. Markantonakis, A. Bilas, and J-J. Quisquater, editors, *WISTP*, volume 4462 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2007.
11. Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast Exponentiation with Precomputation (Extended Abstract). In Rainer A. Rueppel, editor, *EUROCRYPT*, volume 658 of *Lecture Notes in Computer Science*, pages 200–207. Springer, 1992.
12. M. Ciet and M. Joye. Elliptic Curve Cryptosystems in the Presence of Permanent and Transient Faults. *Des. Codes Cryptography*, 36(1):33–43, 2005.
13. C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. Horizontal Correlation Analysis on Exponentiation. In Miguel Soriano, Sihan Qing, and Javier López, editors, *ICICS*, volume 6476 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2010.
14. J-S Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 1999*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.
15. J-S. Coron. A New DPA Countermeasure Based on Permutation Tables. In R. Ostrovsky, R. De Prisco, and I. Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2008.
16. J. Fan, B. Gierlichs, and F. Vercauteren. To Infinity and Beyond: Combined Attack on ECC Using Points of Low Order. In B. Preneel and T. Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2011.
17. FIPS PUB 186-3. *Digital Signature Standard*. National Institute of Standards and Technology, october 2009.

18. C. Giraud. An RSA Implementation Resistant to Fault Attacks and to Simple Power Analysis. *IEEE Trans. Computers*, 55(9):1116–1120, 2006.
19. L. Goubin. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In Yvo Desmedt, editor, *Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2003.
20. D. Hankerson, A.J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Professional Computing Series, 01 2003.
21. Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. Countermeasures for Preventing Comb Method Against SCA Attacks. In Robert H. Deng, Feng Bao, Hwee-Hwa Pang, and Jianying Zhou, editors, *ISPEC*, volume 3439 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2005.
22. K. Itoh, T. Izu, and M. Takenaka. Efficient Countermeasures against Power Analysis for Elliptic Curve Cryptosystems. In J-J. Quisquater, P. Paradinas, Y. Deswarte, and A. A. El Kalam, editors, *CARDIS*, pages 99–114. Kluwer, 2004.
23. K. Itoh, T. Izu, and M. Takenaka. Improving the Randomized Initial Point Countermeasure against DPA. In J. Zhou, M. Yung, and F. Bao, editors, *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006*, volume 3989 of *Lecture Notes in Computer Science*, pages 459–469, 2006.
24. M. Joye and S-M. Yen. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 291–302, 2002.
25. Marc Joye. Highly Regular m-ary Powering Ladders. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2009.
26. Marc Joye and Mohamed Karroumi. Memory-Efficient Fault Countermeasures. In Emmanuel Prouff, editor, *CARDIS*, volume 7079 of *Lecture Notes in Computer Science*, pages 84–101. Springer, 2011.
27. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
28. Chae Hoon Lim and Pil Joong Lee. More Flexible Exponentiation with Precomputation. In Yvo Desmedt, editor, *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.
29. Nashwa A. F. Mohamed, Mohsin H. A. Hashim, and Michael Hutter. Improved Fixed-Base Comb Method for Fast Scalar Multiplication. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *AFRICACRYPT*, volume 7374 of *Lecture Notes in Computer Science*, pages 342–359. Springer, 2012.
30. Bodo Möller. Improved Techniques for Fast Exponentiation. In Pil Joong Lee and Chae Hoon Lim, editors, *ICISC 2002*, volume 2587 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2003.
31. P.L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.
32. Woei-Jiunn Tsaur and Chih-Ho Chou. Efficient algorithms for speeding up the computations of elliptic curve cryptosystems. *Applied Mathematics and Computation*, 168(2):1045–1064, 2005.
33. C. D. Walter. Sliding Windows Succumbs to Big Mac Attack. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 286–299. Springer, 2001.
34. A. C. Yao. On the evaluation of powers. *SIAM J. Comput.*, 5(1):100–103, 1976.