Countermeasures against Side-Channel Attacks for Elliptic Curve Cryptosystems*

Antonio Bellezza**
November 2001

1 : Overview

In recent years, some attacks on cryptographic systems have been deviced, exploiting the leakage of information through so-called "side channels". When a real-life device is performing a coding or decoding procedure, one can measure quantities such as the time employed, the profile of power consumption, the contents of a particular memory cell. If the algorithm is known, this information can get to the knowledge of part or all of the secret hidden in the device. This is the case when an exponentiation with secret exponent is performed according to a known deterministic algorithm, in case one can detect the order of squarings and products performed. This setting is common to several crypto-systems, involving computations in $(\mathbf{Z}/pq\mathbf{Z})^*$, in $\mathbf{F}_q^*$ or in

* An extended version of this paper may be submitted to the CHES 2002 workshop.
** abellezza@beautylabs.net
The author thanks Elena Trichina of Gemplus for introducing him to the subject of side-channel attacks and for precious help during the preparation of this paper.

the set of $\mathbf{F}_q$-rational points of an elliptic curve.

Some countermeasures are possible. We show that, in the case of elliptic curves over a binary finite field, one can split point addition into two blocks which, through the addition of a little overhead, can be made undistinguishable from a point doubling. Thus, the whole exponentiation process is performed as a sequence of homogeneous steps.

Another measure involves adding a degree of randomization to the algorithm, so that statistical attacks become unfeasible. For elliptic curve systems, for instance, one can move to a random isomorphic elliptic curve, or change the field representation ([J–T]) or add a random point to the input and subtract a suitable multiple after the exponentiation is performed ([Cor]). This last method is very efficient if one stores a perturbation point and the final correction once and then updates them by doubling both. This presents the disadvantage that one has to keep two points stored in non-volatile RAM. We suggest as an alternative the use of points of small order, that allow on the fly computation of arbitrary multiples with reduced overhead.

Another technique ([Cor]) uses a so called "exponent blinding", by substituting an exponent e by $e+kN$, with k a small random number and N the order of the group. We suggest a multiplicative variation of this, involving a two-phase exponentiation.

We describe and analyze computer experiments implementing some of these ideas.

2 : Introduction

In some of the most widely used cryptographic systems, an exponentiation with a secret exponent is required at some stage. "Exponentiation" in this context is the repeated application of a group operation to the same element. Thus in additive notation it is multiplication by an integer. In RSA, the group is the multiplicative group of integers modulo the product of two big primes, while in discrete log systems it can be the multiplicative group of a finite field or the group of points of an elliptic curve with point addition. Some general purpose exponentiation algorithms exist, which work for all such settings and involve a sequence of squarings, multiplications and possibly divisions (respectively doublings, additions and possibly subtractions in additive notation). Knowing the sequence of operations can give information on the exponent and in some cases uniquely determine it. For real life implementations, an attacker can be able to perform measurements on the device. If the different operations present different characteristics detectable from the outside (like power consumption profile or duration), the sequence of operations can be discovered by the attacker (see [K]). In order to avoid this, it is important that the sequence of operations in the exponentiation process either be independent of the secret exponent or present homogeneous computational characteristics. For our case of interest, namely elliptic curve cryptosystems, the operations involved are point doubling, addition and subtraction. The computational characteristics depend on the characteristic of the base field and on the representation chosen for points. One is often interested in finite fields of characteristic 2, for which doublings in projective coordinates and additions in mixed affine-projective coordinates with projective result allow to avoid inversions in the field. For this situation, one can split point addition and subtraction into couples of subroutines, each of which is computationally similar to the doubling procedure. Modulo computing some unnecessary operations in the process, one can get the same kinds of field operations performed in the same order with just a little overhead.

Other kinds of side-channel attacks are possible, which employ a statistical analysis of a repeated number of runs. In order to prevent this, suggested techniques require adding a degree of randomness to the process. One proposed method ([Cor]) involves adding a random multiple of the order of the group to the exponent. This doesn't change the outcome, but does change the exponent and thus the order of operations. If e is the original exponent, N the order of the group (a big prime in the elliptic curve case, $(p-1)(q-1)$ in the RSA case, with p and q two big primes), then exponentiation is performed with exponent $e+kN$ with k varying at the end of each run. The disadvantage is that the bit-length of the exponent is increased by about the bit-length of k. We suggest a similar method, expressing e as the product of two numbers, which is always possible, again at the expense of some overhead.

Other techniques involve using an addition or addition/subtraction chain with some degree of randomness. For instance, $9P$ (additive notation) can be got with the two different sequences

$$\begin{array}{ll}
a_1 \leftarrow P & b_1 \leftarrow P \\
a_2 \leftarrow 2a_1 = 2P & b_2 \leftarrow 2b_1 = 2P \\
a_3 \leftarrow 2a_2 = 4P & b_3 \leftarrow 2b_2 = 4P \\
a_4 \leftarrow 2a_3 = 8P & b_4 \leftarrow b_3 + b_1 = 5P \\
a_5 \leftarrow a_1 + a_4 = 9P & b_5 \leftarrow b_3 + b_4 = 9P
\end{array}$$

The two latter techniques still allow the attacker to control on the input being processed in the exponentiation loop. This could be exploited somehow.

Other techniques mask the input by applying some bijective function for which it's easy to compute the final correction. For example, for an elliptic curve this can be done by moving to an isomorphic curve ([J–T]) or adding to the input a "perturbation point" ([Cor]). This latter technique involves storing a couple $(R, -eR)$ and updating it at each run by doubling both elements. Exponentiation of an input I with exponent e (again in additive notation) is performed with the following steps:

$$\begin{aligned} L &\leftarrow I + R \\ L &\leftarrow eL = eI + eR \\ L &\leftarrow L + (-eR) = eI \end{aligned}$$

As the argument $I + R$ of the exponentiation phase does not depend exclusively on the input, it is not possible to choose points with particular properties (such as a particularly fast or particularly expensive doubling).

We suggest a variation on this idea, choosing as perturbation point a point of small order on the curve. This will allow an independent computation of the point and the correction at each run with fewer curve operations.

3 : Preliminaries and notation

Since we are interested in elliptic curves, the notation will be additive. Thus the basic operations will be doubling, addition and subtraction. We will keep the denomination "exponent" for the integer factor.

We will focus on elliptic curves over finite fields of characteristic 2, which have an affine equation of the form

$$y^2 + xy = x^3 + ax^2 + b$$

for two elements a and b in the base field. A point on the curve is either identified by a couple of affine coordinates (P_x, P_y) or is the special point P_∞. Each point has also a projective expression of the form (P_X, P_Y, P_Z). If the point is P_∞, then its projective coordinates are $(1, 1, 0)$. Else they are related to the affine coordinates via the trasformation rules

$$\begin{aligned} (P_x, P_y) &\longmapsto (P_x, P_y, 1) \\ (P_X/P_Z^2, P_Y/P_Z^3) &\longleftarrow\!\!\dashv (P_X, P_Y, P_Z) \end{aligned}$$

If P and Q are points on the curve, then an addition $\oplus$ is defined so that $P \oplus Q$ is also a point on the curve and P_∞ is the neutral element. Thus one can also define a subtraction $P \ominus Q$.

Some of the algorithms we will describe will work with different exponentiation procedures. In order to be able to interchange them, we will rely on an algorithm that trasforms an integer into an array indicating the order of operations to perform. Possible contents of the resulting array cells are

D	Double the result
A	Add the input to the result
S	Subtract the input from the result

They can be coded in 1 bit if subtraction is never performed or in 2 bits if all three possibilities can appear.

Given such an algorithm ES (for Exponentiation Sequence), one can exponentiate with the following algorithm:

Input:
element to exponentiate I, array $a[1], \ldots, a[l]$ obtained by ES applied to exponent e.
Output:
eI.

- Set R as the zero element (P_∞ in the elliptic curve case);
- For $i \leftarrow 1, \ldots, l$ do:
  - If $a[i] = \mathrm{D}$:
    - $R \leftarrow 2R$;
  - Else if $a[i] = \mathrm{A}$:
    - $R \leftarrow R \oplus I$;
  - Else if $a[i] = \mathrm{S}$:
    - $R \leftarrow R \ominus I$;
- Return R.

Choosing an appropriate algorithm ES, one can reproduce the binary and signed-binary left-to-right algorithms.

In case the input is a fixed I, one can speed up the process by precomputing and storing the elements $2I, 4I, \ldots, 2^k I$ where 2^k is the biggest power of 2 not exceeding the order of I. In this case we will rely on an algorithm IES (from Indexed Exponentiation Sequence) returning an array $a[1], \ldots, a[l]$, where $a[i]$ is a structure made of a field $a[i].op$ which is either A or S and an integer field $a[i].d$ from 0 to k which indicates which of the 2^i-th multiples of I one should add or subtract. To implement the structure, one can just employ an additional bit for the field *op*, or even completely avoid it if one knows *a priori* that the operation is always an addition.

An IES procedure together with the following algorithm would perform exponentiation:

Settings:
Elements $I, 2I, \ldots, 2^k I$.
Input:
Array $a[1], \ldots, a[l]$ obtained by IES applied to exponent e.
Output:
eI.

- Set R as the zero element (P_∞ in the elliptic curve case);
- For $i \leftarrow 1, \ldots, l$ do:
  - If $a[i].op = \mathrm{A}$
    - $R \leftarrow R \oplus 2^{a[i].d} I$;
  - Else if $a[i].op = \mathrm{S}$
    - $R \leftarrow R \ominus 2^{a[i].d} I$;
- Return R.

4 : Homogeneous point operations in characteristic 2

In the choice of doubling and addition algorithms for elliptic curves, it is often advisable to work in projective coordinates, which allows us to avoid costly inversions in the field. For elliptic curves over fields of characteristic 2, complexities for doubling and addition are

Table 4.1

	General case			**Case $a = 1$**		
Curve operation	M	S	A	M	S	A
Doubling	5	5	4	5	5	4
Addition	11	3	7	10	3	7
Subtraction	11	3	8	10	3	8

The doubling is performed on projective coordinates and addition and subtraction on mixed affine-projective coordinates with projective result.

The problem we are trying to address is the relevant computational difference between addition/subtraction and doubling, which in implementations reflects in differences detectable by an attacker. Observing that the complexity for point doubling is about half of the complexity for addition/subtraction, one can think of splitting the complex subroutines into two parts, each one approximately equivalent to a doubling. This is actually possible. Before proceeding, we observe that computing the subtraction $P \ominus Q$ of two points just corresponds to computing $P \oplus (-Q)$. If Q has affine coordinates (Q_x, Q_y), then $-Q$ has affine coordinates $(Q_x, Q_y + Q_x)$. Thus, the algorithms for addition and subtraction only differ by a field addition performed in the subtraction and we will consider them at the same time. Furthermore, if the affine point to be subtracted is the same throughout a loop, one can just compute $Q_x + Q_y$ once.

Here are the subroutines:

Doubling

The input is given by the coordinates (A_X, A_Y, A_Z) of the point to double.

$\lambda_1 \leftarrow A_Z^2$
$\lambda_2 \leftarrow A_X \lambda_1$
$\lambda_3 \leftarrow A_Y A_Z$
$\lambda_4 \leftarrow A_X^2$
$\lambda_5 \leftarrow \lambda_2 + \lambda_4$
$\lambda_6 \leftarrow \widetilde{b} \lambda_1$
$\lambda_7 \leftarrow A_X + \lambda_6$
$\lambda_8 \leftarrow \lambda_4^2$
$\lambda_9 \leftarrow \lambda_8 \lambda_2$
$\lambda_{10} \leftarrow \lambda_5 + \lambda_3$
$\lambda_{11} \leftarrow \lambda_7^4$ obtained by two squarings
$\lambda_{12} \leftarrow \lambda_{10} \lambda_{11}$
$\lambda_{13} \leftarrow \lambda_9 + \lambda_{12}$

Here $\widetilde{b}$ is such that $\widetilde{b}^4 = b$. It can be computed once and for all when the curve is set. The ouput point has projective coordinates $(\lambda_{11}, \lambda_{13}, \lambda_2)$.

The sequence of operations is SMMSAMASMASSMA.

Addition/subtraction - first subroutine

The input is given by the projective coordinates (A_X, A_Y, A_Z) and the affine coordinates (B_x, B_y) of the points to add.

$\lambda_1 \leftarrow {A_Z}^2$

$$\begin{aligned}
\lambda_2 &\leftarrow \lambda_1 B_x \\
\lambda_3 &\leftarrow \lambda_1 A_Z \\
\lambda_4 &\leftarrow B_y \qquad\qquad \text{(addition)}
\end{aligned}$$

or

$$\begin{aligned}
\lambda_4 &\leftarrow B_y + B_x \qquad \text{(subtraction)} \\
\lambda_5 &\leftarrow \lambda_4 \lambda_3 \\
\lambda_6 &\leftarrow A_X + \lambda_2 \\
\lambda_7 &\leftarrow A_Z \lambda_6 \\
\lambda_8 &\leftarrow A_Y + \lambda_5 \\
\lambda_9 &\leftarrow \lambda_8 B_x \\
\lambda_{10} &\leftarrow \lambda_7 \lambda_4 \\
\lambda_{11} &\leftarrow \lambda_7 + \lambda_8
\end{aligned}$$

The sequence of operations is SMMAMAMAMMA.

Addition/subtraction - second subroutine

This soubroutine is executed right after the previous one, of which it uses some of the partial results.

$$\begin{aligned}
\mu_1 &\leftarrow \lambda_7^2 \\
\mu_2 &\leftarrow \mu_1 a \\
\mu_3 &\leftarrow \lambda_8 \lambda_{11} \\
\mu_4 &\leftarrow \lambda_6^2 \\
\mu_5 &\leftarrow \mu_2 + \mu_3 \\
\mu_6 &\leftarrow \lambda_6 \mu_4 \\
\mu_7 &\leftarrow \mu_5 + \mu_6 \\
\mu_8 &\leftarrow \mu_7 \lambda_{11} \\
\mu_9 &\leftarrow \lambda_9 + \lambda_{10} \\
\mu_{10} &\leftarrow \mu_9 \mu_1 \\
\mu_{11} &\leftarrow \mu_8 + \mu_{10}
\end{aligned}$$

The output point has projective coordinates $(\mu_7, \mu_{11}, \lambda_7)$.

The sequence of operations is SMMSAMAMAMA.

By adding unused operations, we can make the sequence of operations for the three subroutines equal to SMMSAMASMASSMMA, which amounts to 6 products, 5 squarings and 4 additions.

Observe that the addition and subtraction algorithms are not valid if one of the points is zero or the points are equal (resp. opposite for subtraction). One should add a check at the beginning of the routines, but during the use in an exponentiation algorithm one is usually sure not to be in one of these exceptional cases.

When choosing an elliptic curve, setting $a = 1$ allows us to skip the product yielding μ_2 in addition-subtraction-phase II subroutines. This makes the total number of products in addition and subtraction exactly twice the number for doubling and efficiency can be increased. To get the new subroutines, one has to move a product from phase I of addition and subtraction to addition-subtraction-phase II. This is achieved in the following subroutines, for which we keep as much as possible the numeration of the previous ones:

Addition/subtraction - first subroutine (case $a = 1$)

As above, the input is given by the projective coordinates (A_X, A_Y, A_Z) and the affine coordinates (B_x, B_y) of the points to add.

$\lambda_1 \leftarrow {A_Z}^2$
$\lambda_2 \leftarrow \lambda_1 B_x$
$\lambda_3 \leftarrow \lambda_1 A_Z$
$\lambda_4 \leftarrow B_y$ (addition)

or

$\lambda_4 \leftarrow B_y + B_x$ (subtraction)
$\lambda_5 \leftarrow \lambda_4 \lambda_3$
$\lambda_6 \leftarrow A_X + \lambda_2$
$\lambda_7 \leftarrow A_Z \lambda_6$
$\lambda_8 \leftarrow A_Y + \lambda_5$
$\lambda_9 \leftarrow \lambda_8 B_x$
$\lambda_{11} \leftarrow \lambda_7 + \lambda_8$

The sequence of operations is SMMAMAMAMA.

Addition/subtraction - second subroutine (case $a = 1$)

This soubroutine is executed right after the previous one, of which it uses some of the partial results.

$\mu_1 \leftarrow \lambda_7^2$
$\mu_2 \leftarrow \lambda_7 \lambda_4$
$\mu_3 \leftarrow \lambda_8 \lambda_{11}$
$\mu_4 \leftarrow \lambda_6^2$
$\mu_5 \leftarrow \mu_1 + \mu_3$
$\mu_6 \leftarrow \lambda_6 \mu_4$
$\mu_7 \leftarrow \mu_5 + \mu_6$
$\mu_8 \leftarrow \mu_7 \lambda_{11}$
$\mu_9 \leftarrow \lambda_9 + \mu_2$
$\mu_{10} \leftarrow \mu_9 \mu_1$
$\mu_{11} \leftarrow \mu_8 + \mu_{10}$

The output point has (again) projective coordinates $(\mu_7, \mu_{11}, \lambda_7)$.

The sequence of operations is SMMSAMAMAMA.

The shortest operation sequence including the two above and the sequence for doubling is now SMMSAMASMASSMA, using 5 products, 5 squarings and 4 additions.

The complexity table for various possibilities is the following

Table 4.2

	General case			**Case $a = 1$**		
Curve operation	M	S	A	M	S	A
Doubling (old)	5	5	4	5	5	4
Doubling (new)	6	5	4	5	5	4
Addition (old)	11	3	7	10	3	7
Addition (new)	12	10	8	10	10	8
Subtraction (old)	11	3	8	10	3	8
Subtraction (new)	12	10	8	10	10	8

Observe that (especially when working with normal basis), squarings in characteristic 2 are very cheap. Thus we are wasting little more than one field multiplication per curve operation in the general case, and only a few additions and squarings in the case $a = 1$.

5 : An exponentiation algorithm using homogeneous point operations

The preceding algorithms can be used to implement an exponentiation algorithm which shows to the outside a sequence of homogeneous rounds. The subroutines for doubling, addition-phase I, subtraction-phase I and addition/subtraction-phase II are implemented. In order to achieve homogeneity, they are invoked with the same kind of parameters, i.e. with 5 field elements representing the projective and affine coordinates of two curve points. The result is a point in projective coordinates, i.e. a triplet of field elements. We suppose that phase I of addition and subtraction return the first input point (the one in projective coordinates). The functions will be called F_i with i being

0 for doubling, 1 for addition/I, 2 for subtraction/I and 3 for addition-subtraction/II. The functions F_1 and F_2 also have side-effects, namely they store in memory partial results which will be used by F_3.

Suppose the exponent e is stored as an array of symbols D, A and S as output by an ES algorithm as described in Chapter 3. Then one creates an array where each D is coded as 0, each A as the two adjacent cells containing 1 and 2 and each S as two cells containing 1 and 3. This way we get an array coding the sequence of F_i's to use and we invoke them accordingly. This is resumed in the following algorithm. We suppose the curve parameters to be accessible to the subroutines and thus we do not specify them explicitly as input.

Input:
Point $I = (I_x, I_y)$ in affine coordinates
Exponent e

Output:
Point eI in affine coordinates

Precomputations:

- Store $ES(e)$ in array $a[1], \ldots, a[l]$ — Compute the sequence of operations via an ES-algorithm.
- Set $d \leftarrow 1$ — The destination index, updated as the destination array is filled
- For $i \leftarrow 1, \ldots, l$:
  - Switch($a[i]$):
    - Case D: Set $b[d] \leftarrow 0$; set $d \leftarrow d + 1$
    - Case A: Set $b[d] \leftarrow 1$; set $b[d+1] \leftarrow 3$; set $d \leftarrow d + 2$
    - Case S: Set $b[d] \leftarrow 2$; set $b[d+1] \leftarrow 3$; set $d \leftarrow d + 2$

Main loop:

- Set $(S_X, S_Y, S_Z) \leftarrow (1, 1, 0)$ — Projective coordinates of P_∞. This is going to contain the result in projective coordinates
- For $i \leftarrow 1, \ldots, d - 1$:
  - Set $(S_X, S_Y, S_Z) \leftarrow F_{b[i]}(S_X, S_Y, S_Z, I_x, I_y)$ — In case $b[i] = 1, 2$, this also stores in memory some values used in the following F_3.

Final conversion:

- Set $S_x \leftarrow S_X/S_Z^2$
- Set $S_y \leftarrow S_Y/S_Z^3$
- Return the point with affine coordinates (S_x, S_y)

In the loop in the precomputation phase, case D differs from cases A and S. If the exponent is costant through several runs, this phase can be performed only when the exponent is set on the device and one doesn't have to worry. If e varies at each run (for instance if random multiples of the group order are added to it), then care must be taken to mask this phase.

6 : Randomizing the input point

A component of some statistical attacks is the choice of inputs for which some quantity appearing in the exponentiation process is known to have particular properties ([Kocher]). For instance, one could choose a point such that its 2^i-th multiple has a zero coordinate and check whether a particularly efficient addition appears in the process. A suggested countermeasure ([Cor]) involves perturbating the input point by adding a random point R and adding $-eR$ at the end of the exponentiation loop. Computing a couple $(R, -eR)$ would increase too much the complexity of the algorithm (unless one exploits some kind of parallelism), so one stores such a couple and updates it by multiplying each of the two points by a same small factor at the end of each run. With this method, the attacker has no control whatsoever on the point processed, as the outcome of the perturbation can be any point in the group. Besides, this adds a degree of variability to the field operations involved and this constitutes a defense against a wide spectrum of statistical attacks. A disadvantage of this method is that it requires two supplementary points to be stored in non-volatile RAM. On devices with constraints, one would prefer to avoid this.

We suggest a variation using points of small order which presents some disadvantages but allows on-the-fly computation of the perturbation point and the final correction.

Elliptic curves of use for cryptographic purposes have a group of points with order a big prime N times a cofactor h. As the order of the total group is about the size of the base field (with difference being at most of the order of twice the square root of the size of the field), in order to exploit efficiently the field one usually requires the cofactor to be as small as possible. We suggest choosing as the perturbation point R a point of order h. We set the following hypotesis:

i. We call l_N and l_h the bit-lengths of N and h respectively.
ii. We require the subgroup of order h, which is either cyclic or the product of two cyclic groups, to be cyclic with a generator we will call Q.
iii. We suppose the cofactor not to be too small (a cofactor of 2 would be no good) but such that l_h is not bigger than, say, $l_N/5$. [ANSI] contains an example with $l_h = 16$ and $l_N = 161$.

We compute a random point of order h as the multiple of Q by a random factor r between 0 and $h-1$. The final correction to apply is $T := -erQ$. We can actually compute it as $(-er \bmod h)Q$. The factor is again between 0 and $h-1$. As the bit-length of h is considerably less than the bit-length of N, the exponentiations to get R and T are faster than the main exponentiation. We will examine the overhead later. One can speed up this phase by storing $2Q, 4Q, \ldots, 2^{l_h-1}Q$.

The computation of rQ and $(-er \bmod h)Q$ must be masked. If the attacker could get simultaneous information on r and $(-er \bmod h)$, with h being public, that would reveal information on the secret exponent e. As r changes after each run, one has to take care mostly of single-run analysis. Storing $2Q, 4Q, \ldots, 2^{l_h}Q$ allows us to perform only additions (or additions and subtractions) and in any order. We code the sequence of which additions to perform on which destination in an array and shuffle it according to another random register s.

The following algorithm relies on a procedure IES as described in Chapter 3. Apart from the fields *op* and d, the blocks computed also store in an additional bit-field c the point involved. This allows us to shuffle the blocks related to different points while keeping able to perform the desired operations.

Settings:
The curve parameters;
The affine coordinates of point Q generating the subgroup of order h;

Input:
A "random" integer r between 0 and h;
A "random" l_h-bit integer s;

Output:
The projective coordinates of rQ and $(-er \bmod h)Q = tQ$.

Computation of the index array:

- $t \leftarrow (-re) \bmod h$

- Store $IES(e)$ in the array $b[1], \ldots, b[l_1]$, setting the field $b[i].c$ to 0;
- Store $IES(t)$ in the array $b[l_1+1], \ldots, b[l_1+l_2]$, setting the field $b[i].c$ to 1;

Shuffling:

- Set $w \leftarrow \lceil \log_2(l_1+l_2) \rceil$; — 2^w is at least the dimension of array b
- For $i = 0, \ldots, l_h - w$ do:
  - Set $start \leftarrow s_i + 2s_{i+1} + \ldots + 2^{w-1}s_{i+w-1}$ — $start$ represents the slice of s from bit i to bit $i+w-1$.
  - Set $end \leftarrow s_{l_h-i-1} + 2s_{l_h-i-2} + \ldots + 2^{w-1}s_{l_h-i-w}$ — the reversed slice of s from bit l_h-i-1 to bit l_h-i-w.
  - For $j = 0, \ldots, \left\lfloor \frac{(end-start) \bmod (l_1+l_2)}{2} \right\rfloor$ do:
    - swap $b\big[\big((start+j) \bmod (l_1+l_2)\big)+1\big]$ and $b\big[\big((end-j) \bmod (l_1+l_2)\big)+1\big]$.

Computation:

- Set $(R_X, R_Y, R_Z) = (1, 1, 0)$ — Projective coordinate of P_∞. This point will evolve into rQ.
- Set $(T_X, T_Y, T_Z) = (1, 1, 0)$ — Projective coordinate of P_∞. This point will evolve into tQ.
- For $i = 1, \ldots, l_1 + l_2$ do:
  - If $b[i].c$ is set: — then we are working on T
    - If $b[i].op = A$
      - compute $(T_X, T_Y, T_Z) \leftarrow (T_X, T_Y, T_Z) \oplus (2^{b[i].d}Q)$
    - else $(b[i].op = S)$:
      - compute $(T_X, T_Y, T_Z) \leftarrow (T_X, T_Y, T_Z) \ominus (2^{b[i].d}Q)$
  - Else ($b[i].c$ not set):
    - If $b[i].op = A$:
      - compute $(R_X, R_Y, R_Z) \leftarrow (R_X, R_Y, R_Z) \oplus (2^{b[i].d}Q)$
    - else $(b[i].op = S)$:
      - compute $(R_X, R_Y, R_Z) \leftarrow (R_X, R_Y, R_Z) \ominus (2^{b[i].d}Q)$
- Return points (R_X, R_Y, R_Z) and (T_X, T_Y, T_Z).

The number of additions and subtractions required by this algorithm depends on the procedure chosen for IES. In case of ordinary binary exponentiation it equals the number of bits set, i.e. an average of l_h additions (twice $l_h/2$). In case of signed NAF exponentiation, this reduces to $\frac{2}{3}l_h$.

We have to point out that the choice of a curve with cofactor of adequate size has a further cost. In fact, $l_N + l_h$ is about the bit-size of the base field. Thus one is forced to use larger fields to achieve the same degree of security.

7 : Masking the exponent

One way to add randomness to the exponentiation process, regardless of the algorithm used, consists in masking the exponent. One known method consists in adding a multiple of the element order to the exponent (see [Cor]). We will refer to this as *additive blinding.* If N is the group order, then $ea = (e + kN)a$ for any integers e and k and for any element a in the group. As e usually is the size of the group order, choosing an l-bit integer k increases the size of the exponent by about l bits.

We suggest a multiplicative analogue of this idea, which we will call *multiplicative blinding.* Let N be the group order. If k is a unit in the multiplicative group $(\mathbf{Z}/N\mathbf{Z}, \times)$ and we call k^{-1} its inverse, then

$$e = k(k^{-1}e) \pmod{N}$$

for any exponent e. This means that calling $e' := k^{-1}e \bmod N$, for any group element a one has the following equivalent exponentiations:

$$\begin{array}{lllll}
1. & a & \xmapsto{e} & ea & \\
2. & a & \xmapsto{k} & ka & \xmapsto{e'} & (e'k)a \\
3. & a & \xmapsto{e'} & e'a & \xmapsto{k} & (ke')a \\
4. & a & \xmapsto{ke'} & (ke')a &
\end{array}$$

Sequence 1. is the straightforward exponentiation, sequence 4. is a subcase of the additive blinding technique. Sequences 2. and 3. are performed in two phases. We will focus on sequence 2. The overhead is given by the exponentiation with exponent k, since both e and e' are on average the size of N. To keep overhead low, one can choose k to yield a fast exponentiation, for instance by choosing it randomly among the elements of $(\mathbf{Z}/N\mathbf{Z})^*$ of at most l bits, with a small l. As in the additive blinding, one can trade-off the degree of randomization (and thus the security) for speed.

If sequence 2. is used, the element fed to the second phase is not controlled by an attacker. The first phase, on the other hand, doesn't leak any information whatsoever on the secret exponent, depending uniquely on the input element and the random integer k. Thus one gets at the same time a change in the sequence of point operations and, in the second phase, a randomization of the processed point.

8 : Computer experiments

We implemented some of the idea explained in the previous chapters. For finite fields, we implemented basic operations from scratch in C following [Impl]. For multiplication we implemented right to left and left to right comb methods, plus a base 16 left to right comb method. As the inversion algorithm we chose the extended euclidean algorithm. The resulting libraries are available for download at [BFFL]. Sample timings for a few fields are given below

Table 8.1(Timings for field operations in μs)

	$\mathbf{F}_{2^{163}}$	$\mathbf{F}'_{2^{163}}$	$\mathbf{F}_{2^{233}}$	$\mathbf{F}_{2^{283}}$
Operation				
Addition	0.16	0.16	0.17	0.16
Squaring	0.34	0.37	0.38	0.53
Products:				
RL comb	7.00	7.13	9.86	18.76
LR comb	10.04	10.10	14.93	18.72
Base 16 LR comb	2.78	2.80	3.61	4.59
Inversion	36.02	36.05	60.50	77.97

Timings where taken on an AMD Duron 600 Mhz running Linux. As polynomial basis, we followed [Impl] and [ANSI], choosing

$$\mathbf{F}_{2^{163}} = \mathbf{F}_2[x]/(x^{163} + x^7 + x^6 + x^3 + 1)$$
$$\mathbf{F}'_{2^{163}} = \mathbf{F}_2[x]/(x^{163} + x^8 + x^2 + x + 1)$$
$$\mathbf{F}_{2^{233}} = \mathbf{F}_2[x]/(x^{233} + x^{74} + 1)$$
$$\mathbf{F}_{2^{283}} = \mathbf{F}_2[x]/(x^{283} + x^{12} + x^7 + x^5 + 1)$$

We implemented both binary and signed NAF exponentiation for elliptic curves over $F_{2^{163}}$, choosing a curve E_1 with $a = 1$ and a general curve E_2. Point operations were performed using the standard sequence of operations and the homogeneous versions. Both binary and NAF exponentiations were used. The parameters were the following:

Table 8.2(Elliptic curve parameters)

E_1 (from [Impl])		
	F	$\mathbf{F}_2[x]/(x^{163}+x^7+x^6+x^3+1)$
	a	01
	b	02 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD
	N	04 00000000 00000000 000292FE 77E70C12 A4234C33
	h	02
	Q_x	03 FB02D922 0A5E7980 D9C7C192 AFC7EDC4 19B261E4
	Q_y	05 F8692B70 5F82AAF2 7E41D4D3 82D9E359 98979F99
E_2 (from [ANSI])		
	F	$\mathbf{F}_2[x]/(x^{163}+x^8+x^2+x+1)$
	a	07 2546B543 5234A422 E0789675 F432C894 35DE5242
	b	00 C9517D06 D5240D3C FF38C74B 20B6CD4D 6F9DD4D9
	N	04 00000000 00000000 000292FE 77E70C12 A4234C33
	h	02
	Q_x	07 AF699895 46103D79 329FCC3D 74880F33 BBE803CB
	Q_y	06 434AB98E 1F769093 2FA04BCA 9ED0479D 4B5FC954

To get the following timings, the generating points were multiplied by a random exponent both with standard and homogeneous operations. For E_1 the fact that $a = 1$ was taken into account in both versions.

Table 8.3(Exponentiation timings in μs)

	binary	**NAF**
E_1		
standard	5355	4699
homogeneous	5512	4847
overhead	2.9%	3.1%
E_2		
standard	5642	4932
homogeneous	6581	5770
overhead	16.6%	17.0%

The expected overhead for the general case, only taking into account field multiplications, would be

$$\frac{6 + 12/2}{5 + 11/2} - 1 \cong 14.3\%$$

in the case of binary exponentiation and

$$\frac{6 + 12/3}{5 + 11/3} - 1 \cong 15.4\%$$

for NAF exponentiation. Taking also squarings and additions into account, the experimental results closely reflect the expected increase in complexity. The case $a = 1$ as expected presents a very low computational cost with respect to standard implementations.

Bibliography

[ANSI] *ANSI working draft x9.62-1998*

[BFFL] Antonio Bellezza *Binary Finite Field Library* `http://www.beautylabs.net/software/finitefields.html`

[Cor] J. S. Coron *Resistance against differential power analysis attacks for elliptic curve cryptosystems* Proc. CHES 1999 *LNCS 1717 (1999)*

[Impl] Darrel Hankerson, Julio López Hernandez, Alfred Menezes *Software implementation of elliptic curve cryptography over binary fields* Proc. CHES 2000 *LNCS 1965 (2000)*

[J–T] Marc Joye, Cristophe Tymen *Protections against Differential Analysis for Elliptic Curve Cryptography – An Algebraic Approach* Proc. CHES 2001 *LNCS 2162 (2001)*

[K] Paul C. Kocher *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems* Proc. Advances in Cryptology–Crypto'96 *LNCS 1109 (1996)*

[O–A] Elisabeth Oswald, Manfred Aigner *Randomized Addition-Subtraction Chains as a Countermeasure against Power Attacks* Proc. CHES 2001 *LNCS 2162 (2001)*

Contents