Transient Fault Induction Attacks on XTR

Mathieu Ciet[1] and Christophe Giraud[2]

[1] Innova Card,
Avenue Coriandre, 13 600 La Ciotat, France
mathieu.ciet@innova-card.com
[2] Oberthur Card Systems,
25, rue Auguste Blanche, 92 800 Puteaux, France
c.giraud@oberthurcs.com

Abstract. At Crypto 2000, the public-key system XTR was introduced by Lenstra and Verheul. This system uses an efficient and compact method to represent subgroup elements. Application of XTR in cryptographic protocols, such as Diffie-Hellman key agreement, El Gamal encryption or DSA signature, greatly reduces the computational cost without compromising security. XTR in the presence of a fault, *i.e.* when processing under unexpected conditions, has never been studied. This paper presents four different fault analyses and shows how an error during the XTR exponentiation can be exploited by a malicious adversary to recover a part or the totality of the secret parameter. Countermeasures are also presented to counteract fault attacks. They are very simple to implement and induce a negligible performance penalty in terms of both memory and time.

Keywords: Differential fault analysis, public-key system XTR, countermeasures, smart cards.

1 Introduction

XTR has been introduced for the first time in [18] as an extension of the LUC cryptosystem [23]. Whereas LUC uses elements in $\mathrm{GF}(p^2)^*$ with order $p+1$ represented by their trace over $\mathrm{GF}(p)$, XTR represents elements of a subgroup of $\mathrm{GF}(p^6)^*$ of order dividing p^2-p+1 by their trace over $\mathrm{GF}(p^2)$, see [16–19, 24, 25]. In [9], Gong and Harn used a similar idea with a subgroup of order p^2+p+1 of $\mathrm{GF}(p^3)^*$. Compared to the usual representation, Gong and Harn achieved a factor 1.5 size reduction, LUC a factor 2 and XTR a factor 3. The main advantage of the compact representation of XTR is that it speeds up the computations. Finally, to conclude this brief history, Rubin and Silverberg have recently presented an alternative compression method of such cryptosystems in [21], see also [11].

Physical constraints must be taken into account when implementing cryptoalgorithms on tamper-proof devices [2, 3]. In [13, 14], Kocher *et al.* show the importance for an implementation of being resistant against computation leakages by introducing the notion of *side-channel analysis*. In [7, 8, 12, 26], the necessity of

J. López, S. Qing, and E. Okamoto (Eds.): ICICS 2004, LNCS 3269, pp. 440–451, 2004.

taking into account fault induction during computation was underlined. Indeed, if an error occurs during an execution of a crypto-algorithm, the faulty result can be used to obtain information on the secret parameter. Fault attacks are very powerful; for example with an RSA implementation using the CRT, the secret key can be recovered by using only one faulty result of a known input [8,15]. The AES secret key can also be revealed by using only two faulty ciphertexts [20]. These examples illustrate the necessity to ensure that an algorithm does not operate under unexpected conditions.

The rest of this paper is organized as follows. In the next section we briefly describe XTR and its main operation: the XTR exponentiation. Then we present in Section 3 several observations about this exponentiation which are used in Section 4. In the latter, the behaviour of XTR in the presence of a transient fault is analyzed. Four different fault attacks are thus presented. The first one is a random bit-fault attack on a random part of the computation, the second one is a random fault attack on a chosen part of the computation, the third one is an erasing fault analysis on a coordinate at a random moment and the last one is a random bit-fault attack on the exponent. Section 5 deals with practical aspects of fault attacks against XTR and more precisely, with synchronization techniques which can be used to choose the part of the computation to disturb. In Section 6, two efficient countermeasures to counteract fault attacks are presented. It is worth noticing that these countermeasures are extremely simple to implement with a very small amount of extra computation.

2 Generalities About XTR

XTR operations are performed over the field $\mathrm{GF}(p^2)$ where p is chosen as a prime number such that $p^2 - p + 1$ has a sufficiently large prime factor q.

Let p be a prime equal to 2 modulo 3. The polynomial $X^2 + X + 1$ is thus irreducible over $\mathrm{GF}(p^2)$ and the roots α and α^p of this polynomial form an optimal normal basis for $\mathrm{GF}(p^2)$ over $\mathrm{GF}(p)$. Moreover, since $p \equiv 2 \bmod 3$, $\alpha^i = \alpha^{i \bmod 3}$. It follows that:

$$\mathrm{GF}(p^2) \cong \{x_1\alpha + x_2\alpha^2 : \alpha^2 + \alpha + 1 = 0 \text{ and } x_1, x_2 \in \mathrm{GF}(p)\} \tag{1}$$

Each element of $\mathrm{GF}(p^2)$ can thus be represented as a couple (x_1, x_2) where x_1, $x_2 \in \mathrm{GF}(p)$. This representation allows very efficient arithmetic over $\mathrm{GF}(p^2)$ as shown in [16, Lemma 2.2.1] To perform operations over $\mathrm{GF}(p^2)$ instead over $\mathrm{GF}(p^6)$, XTR uses the trace over $\mathrm{GF}(p^2)$ to represent elements of a subgroup $\langle g \rangle$ of $\mathrm{GF}(p^6)^*$ with order dividing $p^2 - p + 1$, *i.e.* elements of this subgroup do not belong to $\mathrm{GF}(p)$, $\mathrm{GF}(p^2)$ and $\mathrm{GF}(p^3)$. XTR provides the $\mathrm{GF}(p^6)$ security with calculations in $\mathrm{GF}(p^2)$.

The trace over $\mathrm{GF}(p^2)$ of an element $g \in \mathrm{GF}(p^6)$ is defined by the sum of the conjugates of g over $\mathrm{GF}(p^2)$, $\mathrm{Tr}(g) = g + g^{p^2} + g^{p^4}$. Corollary 1 leads to a fast algorithm to compute $\mathrm{Tr}(g^n)$ from $\mathrm{Tr}(g)$. For the sake of simplicity, we denote in the rest of this paper $\mathrm{Tr}(g)$ by c and $\mathrm{Tr}(g^k)$ by c_k.

Corollary 1. [16, Corollary 2.3.3] *Let* $c,\ c_{n-1},\ c_n$ *and* c_{n+1} *be given.*

i. Computing $c_{2n} = c_n^2 - 2c_n^p$ *takes two multiplications in* $\mathrm{GF}(p)$*;*
ii. Computing $c_{n+2} = c \cdot c_{n+1} - c^p \cdot c_n + c_{n-1}$ *takes four multiplications in* $\mathrm{GF}(p)$*;*
iii. Computing $c_{2n-1} = c_{n-1} \cdot c_n - c^p \cdot c_n^p + c_{n+1}^p$ *takes four multiplications in* $\mathrm{GF}(p)$*;*
iv. Computing $c_{2n+1} = c_{n+1} \cdot c_n - c \cdot c_n^p + c_{n-1}^p$ *takes four multiplications in* $\mathrm{GF}(p)$.

The XTR exponentiation, *i.e.* computing $\mathrm{Tr}(g^n)$ from $\mathrm{Tr}(g)$ given an integer n, is done by using Algorithm 2.3.5 of [16]. Let $n = \sum_{j=0}^{r} n_j 2^j$ be the secret exponent, we denote by $S_k(c)$ the triplet $(c_{k-1},\ c_k,\ c_{k+1})$ and by $\overline{S}_k(c)$ the triplet $(c_{2k},\ c_{2k+1},\ c_{2k+2})$. Algorithm 2.1 gives the way to compute $S_n(c)$ for any c in $\mathrm{GF}(p^2)$.

Algorithm 2.1 Computation of $S_n(c)$ given n and c, from [16, Algorithm 2.3.5]
INPUT: n and c
OUTPUT: $S_n(c)$

if $n < 0$ **then** apply this algorithm to $-n$ and c, and apply Lem. 2.3.2.*ii* of [16] to the output.
if $n = 0$ **then** $S_0(c) = (c^p, 3, c)$.
if $n = 1$ **then** $S_1(c) = (3, c, c^2 - 2c^p)$.
if $n = 2$ **then** use Cor. 1.*ii* and $S_1(c)$ to compute c_3.
else define $\overline{S}_i(c) = S_{2i+1}(c)$ and let $\overline{m} = n$.
  if $\overline{m}$ is even **then** $\overline{m} \leftarrow \overline{m} - 1$.
  $m \leftarrow \dfrac{\overline{m} - 1}{2}$, $k = 1$,
  $\overline{S}_k(c) \leftarrow S_3(c)$ (use Cor. 1.*i* and $S_2(c)$ to compute c_4).
  $m = \sum_{j=0}^{r} m_j 2^j$ with $m_j \in \{0, 1\}$ and $m_r = 1$.
  for j **from** $r - 1$ to 0 **do**
    if $m_j = 0$ **then** compute $\overline{S}_{2k}(c)$ from $\overline{S}_k(c)$
      (using Cor. 1.*i* for c_{4k} and c_{4k+2} and Cor. 1.*iii* for c_{4k+1}).
    if $m_j = 1$ **then** compute $\overline{S}_{2k+1}(c)$ from $\overline{S}_k(c)$
      (using Cor. 1.*i* for c_{4k+2} and c_{4k+4} and Cor. 1.*iv* for c_{4k+3}).
    $k \leftarrow 2k + m_j$
  if n is even **then** use $S_{\overline{m}}(c)$ to compute $S_{\overline{m}+1}(c)$ (using Cor. 1.*ii*) and $\overline{m} \leftarrow \overline{m} + 1$.
return $S_n(c) = S_{\overline{m}}(c)$

3 Some Useful Remarks

3.1 Computing $\overline{S}_k(c)$ from $\overline{S}_{2k}(c)$ or $\overline{S}_{2k+1}(c)$

From $\overline{S}_{2k}(c)$, c_{2k} and c_{2k+1} can be obtained by using the following corollary. Once these two values are determined, the last third of $\overline{S}_k(c)$ is linearly obtained by using Corollary 1.*iii*: $c_{2k+2}^p c_{4k+1} - c_{2k} \cdot c_{2k+1} - c^p \cdot c_{2k+1}^p$. Recovering $\overline{S}_k(c)$ from $\overline{S}_{2k+1}(c)$ can be done in a similar manner.

Corollary 2. *Computing* $c_k = (x_1, x_2)$ *from* $c_{2k} = (y_1, y_2)$ *can easily be done by computing* $S_{2^{-1} \bmod q}(c_{2k})$.

Proof. By definition, $\forall i \in \mathbb{Z},\ c_i \in \mathrm{GF}(p^2)$. So $\forall j \in \mathbb{Z}$, we can compute:

$$S_j(c_i) = (\mathrm{Tr}((g^i)^{j-1}),\ \mathrm{Tr}((g^i)^j),\ \mathrm{Tr}((g^i)^{j+1}) = (c_{i*(j-1)},\ c_{i*j},\ c_{i*(j+1)})$$

Then, as g is of prime order q,

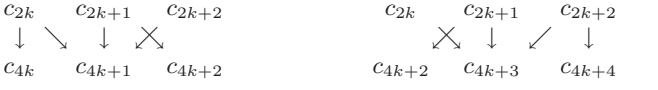
$$S_{2^{-1} \bmod q}(c_{2k}) = (c_{-k},\ c_k,\ c_{3k}) \tag{2}$$

□

3.2 An Observation About the Exponentiation

Considering the formulas used to perform the XTR exponentiation, we remark that the c_i's used during computations inside the main loop are dependent of the corresponding value of the bit of n (*cf.* Fig. 1):

- If $n_j = 0$ then, the first element of $\overline{S}_{2k}(c)$ is only computed from c_{2k}, the second one from all the three elements of $\overline{S}_k(c)$ and the third one only from c_{2k+1}.
- If $n_j = 1$ then, the first element of $\overline{S}_{2k+1}(c)$ is only computed from c_{2k+1}, the second one from all the three elements of $\overline{S}_k(c)$ and the third one only from c_{2k+2}.

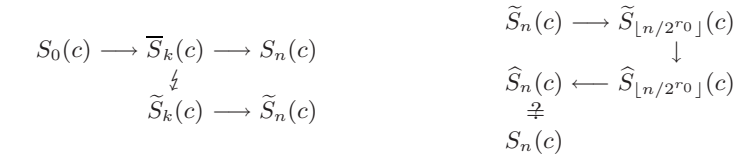


(a) Computation of $\overline{S}_{2k}(c)$. (b) Computation of $\overline{S}_{2k+1}(c)$.

Fig. 1. Algorithms to compute $\overline{S}_{2k}(c)$ and $\overline{S}_{2k+1}(c)$.

4 Fault Analysis

Following the previous remarks, this section deals with fault analysis against XTR. We show how an attacker can retrieve the secret key when the crypto-algorithm XTR proceeds under unexpected conditions. Two types of induced faults can mainly be distinguished: the one induced on a temporary result (Sections 4.1, 4.2 and 4.3) and the (more classical) one induced on the secret scalar (Section 4.4). The current size of the security parameter q (resp. p) is of 160 bits (resp. 170 bits).



(a) Random bit-fault on a random $\overline{S}_k(c)$. (b) Fault analysis skeleton.

Fig. 2. Bit-fault model.

4.1 Random Bit-Faults on a Random $\overline{S}_k(c)$

This first attack stems from the fault attacks on the elliptic curve scalar multiplication described by Biehl *et al.* in [6].

If one bit of a temporary result $\overline{S}_k(c)$ is disturbed during the computation of $S_n(c) = (c_{n-1}, c_n, c_{n+1})$, then a faulty result denoted $\widetilde{S}_n(c) = (\widetilde{c}_{n-1}, \widetilde{c}_n, \widetilde{c}_{n+1})$ is computed instead of the correct one $S_n(c)$.

The attacker knows $S_n(c)$ and $\widetilde{S}_n(c)$ but she does not know which $\overline{S}_k(c)$ has been disturbed neither the bit of $\overline{S}_k(c)$ which has been flipped.

At the beginning, the step r_0 when the fault is induced is guessed. An hypothesis on the first r_0 bits of n is also made. By using these guesses and formulas from Section 3.1, $\widetilde{S}_{\lfloor n/2^{r_0} \rfloor}(c)$ can be obtained from $\widetilde{S}_n(c)$. For each possible value of the induced fault ($6 * 170$ possibilities), $\widehat{S}_{\lfloor n/2^{r_0} \rfloor}(c)$ is computed. Then, $\widehat{S}_n(c)$ is evaluated by using the guess on the first r_0 bits of n. If $S_n(c) = \widehat{S}_n(c)$ then the hypothesis on the first r_0 bits of the secret value n is correct, or else another guess is done on the first r_0 bits of n. If the equality $S_n(c) = \widehat{S}_n(c)$ never occurs for any of the possible values for the first r_0 bits of n, this implies that the guess on the position r_0, where the fault has been induced, has to be changed.

To estimate the cost of this attack, we can suppose from a practical point of view that step r_0 is known with approximately one or two positions (the pertinence of this supposition is justified in Section 6). If we suppose that an error occurs at the r_0^{th} step, this attack then requires at most $2 * 6 * 170 * 2^{r_0 - 1}$ computations of $\widetilde{S}_{2^{-1} \bmod q}(c_i)$ and of $\widehat{S}_n(c)$. For example, if an error occurs at step 20, the previous attack allows one to recover 20 bits by computing less than 2^{30} $\widetilde{S}_{2^{-1} \bmod q}(c_i)$ and 2^{30} $\widehat{S}_n(c)$.

More importantly, if we suppose that a first faulty result is obtained with a fault induced at step r_0 and a second faulty result is obtained with a fault induced at step r_1 with $r_1 > r_0 > 1$ then, r_1 bits of the secret exponent can be found in less than $2040 * (2^{r_0 - 1} + 2^{r_1 - r_0})$ computations of $\widetilde{S}_{2^{-1} \bmod q}(c_i)$ and of $\widehat{S}_n(c)$. This is much less than $2040 * 2^{r_1 - 1}$. Finally, applying this principle to several induced faults r with $r_i > r_j$ for any $i > j$ allows one to find the secret exponent with a very small complexity.

4.2 Random Faults on a Chosen c_i

Let us suppose that the result $S_n(c)$ of the correct exponentiation is known. Let us also suppose the first bits of n are known such as we could recover the value of

a temporary result $S_m(c)$ by iterating the method described in Section 3.1. By observing Algorithm 2.1, we can consider that $S_m(c)$ is computed from a triplet $\overline{S}_k(c) = (c_{2k}, c_{2k+1}, c_{2k+2})$.

If an unknown error is induced on a chosen element of this $\overline{S}_k(c)$, a faulty result $\widetilde{S}_n(c)$ is obtained. From this result, the temporary result $\widetilde{S}_m(c)$ can be recovered by using the same method as the one used to compute $S_m(c)$. By using $S_m(c)$, $\widetilde{S}_m(c)$ and the observation described in Section 3.2, the following cases can be distinguished:

1. The fault has disturbed c_{2k}:
   – The first and the second elements of $\widetilde{S}_m(c)$ are different from the same elements of $S_m(c)$, this implies $n_j = 0$.
   – Only the second element of $\widetilde{S}_m(c)$ is different from the second element of $S_m(c)$, this implies $n_j = 1$.
2. The fault has disturbed c_{2k+1}:
   – The second and the third elements of $\widetilde{S}_m(c)$ are different from the same elements of $S_m(c)$, this implies $n_j = 0$.
   – The first and the second elements of $\widetilde{S}_m(c)$ are different from the same elements of $S_m(c)$, this implies $n_j = 1$.
3. The fault has disturbed c_{2k+2}:
   – Only the second element of $\widetilde{S}_m(c)$ is different from the second element of $S_m(c)$, this implies $n_j = 0$.
   – The second and the third elements of $\widetilde{S}_m(c)$ are different from the same elements of $S_m(c)$, this implies $n_j = 1$.

Thus, by knowing the first j bits of n and by knowing which element of $\overline{S}_k(c)$ has been disturbed, one can recover the value of the j^{th} bit of n.

Finally, by iterating this attack, the whole value of the secret exponent can be recovered by using 160 faulty results obtained from unknown errors.

4.3 Erasing Faults on a Coordinate of c_{2k+1} of a Random $\overline{S}_k(c)$

As aforementioned in Section 3.2, the main loop of the XTR exponentiation depends on the value of the corresponding bit of the exponent n ($n_j = 0$ or 1). In both cases, it can be seen that c_{4k+2} is only computed from c_{2k+1}.

If a perturbation is induced on one of the coordinates of c_{2k+1} and if this perturbation sets the coordinate to zero[1], one of the coordinates of c_{4k+2} is the result of the computation: $0 * (0 - 2 * x) - 2 * 0$ (*cf.* Corollary 1.i and Lemma 2.2.1, i, ii of [16]) and is also equal to zero.

From a practical point of view, such a computation is easily observable by looking at information leakages, as for example the power consumption of the device under which the crypto-algorithm proceeds. Akishita and Takagi, and

[1] Such a perturbation is not easy to induce in practice (22 bytes must be set to zero), but due to improvements of fault attacks, such a fault model has to be considered, see for example [22].

Goubin use such a method to achieve their attacks on elliptic curve cryptosystems [1, 10]. For more information about the detection of a zero value by using power consumption analysis, the reader can refer to [1, § 4.6].

If the observation is made during the computation of the first part of the triplet (c_i, c_{i+1}, c_{i+2}) with $i = 4k + n_j$, it can be deduced that $n_j = 0$ else, if this computation is observed during the computation of the last part of the triplet, it implies that $n_j = 1$. Therefore, one bit can be recovered by this process of observation and by combining a fault attack and a power consumption analysis.

By iterating this principle on every bit of n, the whole value of the secret exponent is recovered. As q is a 160-bit prime and $n < q$, only 160 faulty computations are needed to recover n.

Remark 1. If we succeed in inducing such a fault at the j^{th} step of the exponentiation and if the flipped bit n_j is followed by a set on k complementary bits, the value of these $k + 1$ bits can be deduced by observing the power consumption of the card: if $n_j = 0$ (resp. $n_j = 1$), one of the coordinates of the last part (resp. the first part) of the following k triplets (c_i, c_{i+1}, c_{i+2}) is still zero (*cf.* Example 1). Moreover, we can also deduce the value of the bit n_{j+k+1} which is equal to n_j. The probability of such a sequence is 2^{-k}. Thus, small sequences exist with a non-negligible probability, optimizing the attack.

Example 1.

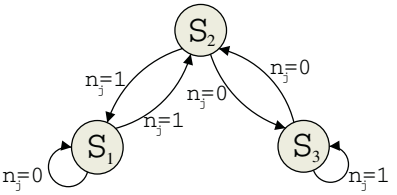
$$
\begin{array}{ccccc}
(x_1, x_2) & & (x_3, 0) & & (x_5, x_6) & \\
\downarrow & \searrow & \downarrow & \swarrow\!\!\searrow & & n_j = 0 \\
(y_1, y_2) & & (y_3, y_4) & & (0, x_3^2) & \\
& \swarrow\!\!\searrow & \downarrow & \swarrow & \downarrow & n_{j+1} = 1 \\
(z_1, z_2) & & (z_3, z_4) & & (x_3^4, 0) & \\
& \swarrow\!\!\searrow & \downarrow & \swarrow & \downarrow & n_{j+2} = 1 \\
(w_1, w_2) & & (w_3, w_4) & & (0, x_3^8) & \\
\downarrow & \searrow & \downarrow & \swarrow\!\!\searrow & & n_{j+3} = 0 \\
(v_1, v_2) & & (v_3, v_4) & & (v_5, v_6) &
\end{array}
$$

This attack can be extended by using a clever remark made in Bévan's thesis [5] in which several chosen-ciphertext power analysis attacks on the XTR exponentiation are described. For one of them, Bévan shows that if the input c of the exponentiation is of the form either $(x, 0)$ or $(0, x)$, there is a finite state machine for the $S_i(c)$'s, $3 \leq i \leq n$ (*cf.* Algorithm 2.1; see Figure 3).

Therefore, if one of the input's coordinates c is set to zero, the secret key can be recovered by analysing the power consumption of only one faulty XTR exponentiation.

4.4 Random Bit-Faults on the Secret Exponent

In previous sections, various faults on various parts of the computation have been considered. Let us suppose now that an attacker is able to flip one bit of the secret exponent during the exponentiation [4]. A faulty result is obtained denoted $c_{\widetilde{n}}$ instead of the correct one c_n. If an attacker succeeds in flipping the i^{th} bit of n, a faulty result $c_{\widetilde{n}}$ is obtained with $\widetilde{n} = n + (-1)^{n_i} 2^i$ where n_i is the



where

$S_1 = ((x_1, x_2), (x_3, 0), (0, x_6))$
$S_2 = ((0, y_2), (y_3, y_4), (y_5, 0))$
$S_3 = ((z_1, 0), (0, z_4), (z_5, z_6))$

(a) if c is of the form $(x, 0)$.

$S_1 = ((x_1, x_2), (0, x_4), (x_5, 0))$
$S_2 = ((y_1, 0), (y_3, y_4), (0, y_6))$
$S_3 = ((0, z_2), (z_3, 0), (z_5, z_6))$

(b) if c is of the form $(0, x)$.

Fig. 3. State machine of the XTR exponentiation with an input of the form $(x, 0)$ or $(0, x)$.

i^{th} bit of n. By testing if $c_{\tilde{n}} = c_{n+2^i}$ or $c_n = c_{\tilde{n}+2^i}$ she obtains the value of the i^{th} bit of n. Moreover c_{j+2^i} can be computed from c_j by using Corollary 1.*ii*. This is achieved with $4 * 2^i$ multiplications in GF(p).

So if the i^{th} bit of n has been flipped, an attacker needs to perform $\sum_{k=0}^{i} 2 * 4 * 2^k = 2^{i+4} - 2^3$ multiplications in GF(p) to recover the value of the i^{th} bit of n. If we suppose that it is feasible to perform 2^{32} multiplications in GF(p), the attacker can thus exploit a faulty result if the fault has been induced on one of the first 30 bits of n. Once discovered the first k bits of n, the attacker can use the method described in Section 3.1 to compute the corresponding temporary result and then continue with the attack on the next 30 bits of n.

5 Practical Aspects: Synchronizing the Attacks

One of the main problem when mounting a fault analysis on a cryptosystem is to respect hypotheses on fault area localization and on fault induction time. By analyzing the power consumption of the device where the algorithm is processed, we can observe the timing where each loop of the XTR exponentiation is performed (see Fig. 4 where a part of the power trace is given). This information is very useful to synchronize fault induction with the beginning of each loop.

Synchronizing an attack to succeed in disturbing the computation of a chosen c_i is slightly more difficult. Computations of the three elements c_{2i}, c_{2i+1} and c_{2i+2} of $\overline{S}_i(c)$ are more often done in ascending order: firstly c_{2i}, then c_{2i+1} and finally c_{2i+2}. As aforementioned, the attacker can use an SPA analysis to detect when the different loops of the exponentiation are performed. The attacker can roughly divide the time required to perform a loop into three (non-equal) periods. As c_{2i} requires 2 multiplications in GF(p), c_{2i+1} requires 4 multiplications and c_{2i+2} 4 multiplications, we can consider that c_i is computed during the first fifth of the loop, c_{2i+2} is computed during the last two fifth of the loop and c_{2i+1} is computed between these two periods of time. Of course the knowledge of the implementation design is a plus when mounting fault analysis and here when defining the subperiods of the loop, *i.e.* order and duration of each c_j.

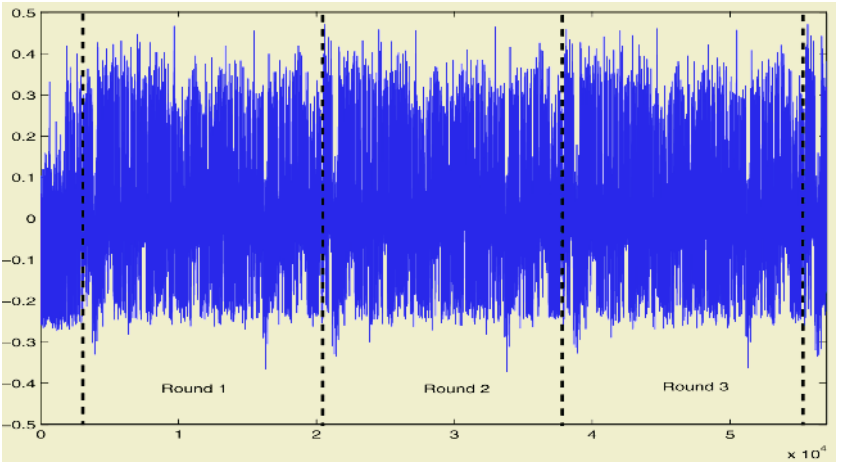


Fig. 4. Power consumption of the XTR exponentiation during the first three loops.

6 Countermeasures

Two kinds of countermeasures can mainly be applied according to the fault that occurs. The first one is to avoid a fault against the secret exponent. The most simple and classical countermeasure is to add a CRC to the exponent at the end of the computation. This CRC is compared with the CRC of the exponent stored in non-volatile memory before outputting the result. This countermeasure is very efficient and simple to implement.

The second countermeasure avoids fault attacks against a temporary result. It uses the fact that we must have three *consecutive* c_i at the end of each loop. If an error is induced on one of these c_i, the coherence between the three elements is lost. To prevent fault attacks on a temporary c_i, a check of consistency can be added at the end of each loop. It proceeds as follows:

- If $n_j = 0$:
  - Compute c_{4k+3} from Corollary 1 *iv*. It uses c_{2k}, c_{2k+1} and c_{2k+2}.
  - Check if $c_{4k+3} = c \cdot c_{4k+2} - c^p \cdot c_{4k+1} + c_{4k}$.
- If $n_j = 1$:
  - Compute c_{4k+1} from Corollary 1 *iii*. It uses c_{2k}, c_{2k+1} and c_{2k+2}.
  - Check if $c_{4k+4} = c \cdot c_{4k+3} - c^p \cdot c_{4k+2} + c_{4k+1}$.

This countermeasure induces a performance penalty of roughly 50% if done on each round, *i.e.* for each bit of n. However, this countermeasure only needs to be applied at the end of the last round since if coherence is broken at anytime of the computation, this incoherence remains until the end and can be detected at this time. The penalty compared with the whole exponentiation is negligible: only 8 multiplications in GF(p), that means 0.6% of penalty. An algorithm including both countermeasures is depicted in Appendix A.

Unfortunately, the previous countermeasure is useless against the second attack described in Section 4.3. Hopefully, by observing that for each possible state there are exactly two computations which result into a zero, this attack can be counteract by randomizing the computation of the three parts of $S_i(c)$, *i.e.* computing the three parts in a random order for each loop of the XTR exponentiation. The attacker could thus not be able to distinguish the different states.

7 Concluding Remarks

In this paper, several fault attacks on the public-key system XTR are described. By using a bit-fault model, the secret key can be revealed by using 160 faulty results, if the faults are induced either on a temporary result or on the secret exponent. Moreover, if one coordinate of the input of the XTR exponentiation can be set to zero, the secret key is obtained in only one shot.

We also describe efficient countermeasures to resist fault attacks, the cost of these countermeasures is very low in terms of both memory space and speed execution. The low cost allows an implementer to efficiently counteract fault analysis with negligible penalty of performance and memory requirements.

Acknowledgements

We are very grateful to Martijn Stam for his careful reading of the preliminary version of this paper and for his very useful comments. We would also like to thank Francesco Sica and Erik Knudsen for answering some mathematical questions and Régis Bévan for helping us when measuring the power consumption of the XTR exponentiation.

References

1. T. Akishita and T. Takagi. Zero-value Point Attacks on Elliptic Curve Cryptosystem. In *Information Security – ISC 2003*, vol. 2851 of *LNCS*, pages 218–233. Springer, 2003.
2. R. Anderson and M. Kuhn. Tamper Resistance - a Cautionary Note. In *Proceedings of the* 2nd *USENIX Workshop on Electronic Commerce*, pages 1–11, 1996.
3. R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In 5th *Security Protocols Workshop*, vol. 1361 of *LNCS*, pages 125–136. Springer, 1997.
4. F. Bao, R. Deng, Y. Han, A. Jeng, A. D. Narasimhalu, and T.-H. Ngair. Breaking Public Key Cryptosystems an Tamper Resistance Devices in the Presence of Transient Fault. In 5th *Security Protocols WorkShop*, vol. 1361 of *LNCS*, pages 115–124. Springer, 1997.
5. R. Bévan. *Estimation statistique et sécurité des cartes à puce – Evaluation d'attaques DPA évoluées*. PhD thesis, Supelec, June 2004.
6. I. Biehl, B. Meyer, and V. Müller. Differential Fault Analysis on Elliptic Curve Cryptosystems. In *Advances in Cryptology – CRYPTO 2000*, vol. 1880 of *LNCS*, pages 131–146. Springer, 2000.
7. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystem. In *Advances in Cryptology – CRYPTO '97*, vol. 1294 of *LNCS*, pages 513–525. Springer, 1997.

8. D. Boneh, R.A. DeMillo, and R.J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology – EUROCRYPT '97*, vol. 1233 of *LNCS*, pages 37–51. Springer, 1997.
9. G. Gong and L. Harn. Public key cryptosystems based on cubic finite field extensions. In *IEEE Transaction on Information Theory*, LNCS. Springer, November 1999.
10. L. Goubin. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystem. In *Public Key Cryptography – PKC 2003*, vol. 2567 of *LNCS*, pages 199–210. Springer, 2003.
11. R. Granger, D. Page, and M. Stam. A Comparison of CEILIDH and XTR. In *Algorithmic Number Theory:* 6^{th} *International Symposium, ANTS-VI*, vol. 3076 of *LNCS*. Springer, 2004.
12. M. Joye, A.K. Lenstra, and J.-J. Quisquater. Chinese Remaindering Based Cryptosystems in the Presence of Faults. *Journal of Cryptology*, 12(4):241–246, 1999.
13. P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO '96*, vol. 1109 of *LNCS*, pages 104–113. Springer, 1996.
14. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO '99*, vol. 1666 of *LNCS*, pages 388–397. Springer, 1999.
15. A.K. Lenstra. Memo on RSA Signature Generation in the Presence of Faults. Manuscript, 1996. Available from the author at `akl@Lucent.com`.
16. A.K. Lenstra and E.R. Verheul. An overview of the XTR public key system. In *Public Key Cryptography and Computational Number Theory Conference*, 2000.
17. A.K. Lenstra and E.R. Verheul. Key improvements to XTR. In *Advances in Cryptology – ASIACRYPT 2000*, vol. 1976 of *LNCS*, pages 220–233. Springer, 2000.
18. A.K. Lenstra and E.R. Verheul. The XTR public key system. In *Advances in Cryptology – CRYPTO 2000*, vol. 1880 of *LNCS*, pages 1–19. Springer, 2000.
19. A.K. Lenstra and E.R. Verheul. Fast irreductibility and subgroup membership testing in XTR. In *Public Key Cryptography – PKC 2001*, vol. 1992 of *LNCS*, pages 73–86. Springer, 2001.
20. G. Piret and J.-J. Quisquater. A Differential Fault Attack Technique Against SPN Structures, with Application to the AES and KHAZAD. In *Cryptographic Hardware and Embedded Systems – CHES 2003*, vol. 2779 of *LNCS*, pages 77–88. Springer, 2003.
21. K. Rubin and A. Silverberg. Torus-based cryptography. In *Advances in Cryptology – CRYPTO 2003*, vol. 2729 of *LNCS*, pages 349–365. Springer, 2003.
22. S. Skorobogatov and R. Anderson. Optical Fault Induction Attack. In *Cryptographic Hardware and Embedded Systems – CHES 2002*, vol. 2523 of *LNCS*, pages 2–12. Springer, 2002.
23. P. Smith and C. Skinner. A public-key cryptosystem and a digital signature system based on the Lucas function analogue to discret logarithms. In *Advances in Cryptology – ASIACRYPT 1994*, vol. 917 of *LNCS*, pages 357–364. Springer, 1994.
24. M. Stam and A.K. Lenstra. Speeding up XTR. In *Advances in Cryptology – ASIACRYPT 2001*, vol. 2248 of *LNCS*, pages 125–143. Springer, 2001.
25. E.R. Verheul. Evidence that XTR Is More Secure then Supersingular Elliptic Curve Cryptosystems. In *Advances in Cryptology – EUROCRYPT 2001*, vol. 2045 of *LNCS*, pages 195–210. Springer, 2001.
26. S.-M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.

A A SPA/DFA-Resistant XTR Exponentiation

Algorithm A.1 Secure computation of $S_n(c)$ given n and c, from Algorithm 2.1
INPUT: n and c
OUTPUT: $S_n(c)$

if $n < 0$ **then** apply this algorithm to $-n$ and c, and apply [16, Lem. 2.3.2.*ii*] to the output.
if $n = 0$ **then** $S_0(c) = (c^p, 3, c)$.
if $n = 1$ **then** $S_1(c) = (3, c, c^2 - 2c^p)$.
if $n = 2$ **then** use Cor. 1.*ii* and $S_1(c)$ to compute c_3.
else define $\overline{S}_i(c) = S_{2i+1}(c)$ and let $\overline{m} = n$.
  if $\overline{m}$ is even **then** $\overline{m} \leftarrow \overline{m} - 1$.
  $m \leftarrow \dfrac{\overline{m} - 1}{2}$, $k = 1$,
  $\overline{S}_k(c) \leftarrow S_3(c)$ (use Cor. 1.*i* and $S_2(c)$ to compute c_4).
  $m = \sum_{j=0}^{r} m_j 2^j$ with $m_j \in \{0, 1\}$ and $m_r = 1$.
  for j **from** $r - 1$ to 0 **do**
    $c_{4k+2m_j} \leftarrow c_{2k+m_j}^2 - 2c_{2k+m_j}^p$
    $c_{4k+1+2m_j} \leftarrow c_{2k+2m_j}.c_{2k+1} - c^{p(1-m_j)+m_j} c_{2k+1}^p + c_{2k+2(1-m_j)}^p$
    $c_{4k+2+2m_j} \leftarrow c_{2k+1+m_j}^2 - 2c_{2k+1+m_j}^p$
    $k \leftarrow 2k + m_j$
  $c_{4k+3-2m_j} \leftarrow c_{2k+2(1-m_j)}.c_{2k+1} - c^{p.m_j+1-m_j}.c_{2k+1}^p + c_{2k+2m_j}^p$
  if $c_{4k+3+m_j} \neq c.c_{4k+2+m_j} - c^p.c_{4k+1+m_j} + c_{4k+m_j}$ **then**
    return ("A fault attack on a temporary result has been detected")
  if n is even **then** use $S_{\overline{m}}(c)$ to compute $S_{\overline{m}+1}(c)$ (using Cor. 1.*ii*) and $\overline{m} \leftarrow \overline{m} + 1$.
if ComputeCRC(n) $\neq$ ComputeCRC(n_{EEPROM}) **then**
  return ($S_{\overline{m}}(c)$)
else
  return ("A fault attack on the exponent has been detected")