

# A Practical Implementation of the Timing Attack

J.-F. Dhem<sup>\*1</sup>, F. Koeune<sup>3</sup>, P.-A. Leroux<sup>3</sup>,  
P. Mestré<sup>\*2</sup>, J.-J. Quisquater<sup>3</sup> and J.-L. Willems<sup>3</sup>

<sup>1</sup> Belgacom Multimedia & Infohighways,  
Bld E. Jacquain 177, B-1030 Brussels, Belgium.

E-mail: `dhem@belbone.be`

<sup>2</sup> Europay International,  
198A Chaussée de Tervuren, B-1410 Waterloo, Belgium.

E-mail: `pme@europay.com`

<sup>3</sup> Université catholique de Louvain, UCL Crypto Group,  
Laboratoire de microélectronique (DICE),  
Place du Levant 3, B-1348 Louvain-la-Neuve, Belgium.

E-mail: `{fkoeune,leroux,jjq,willems}@dice.ucl.ac.be`

URL: <http://www.dice.ucl.ac.be/crypto>

June 15, 1998

**Abstract.** When the running time of a cryptographic algorithm is non-constant, timing measurements can leak information about the secret key. This idea, first publicly introduced by Kocher, is developed here to attack an earlier version of the CASCADE smart card<sup>1</sup>. We propose several improvements on Kocher's ideas, leading to a practical implementation that is able to break a 512-bit key in few hours, provided we are able to collect 300 000 timing measurements (128-bit keys can be recovered in few seconds using a personal computer and less than 10 000 samples). We therefore show that the timing attack represents an important threat against cryptosystems, which must be very seriously taken into account.

**Keywords:** timing attack, cryptanalysis, RSA, smart card.

## 1 Introduction

Implementations of cryptographic algorithms often perform computations in non-constant time, due to performance optimizations. If such operations involve secret parameters, these timing variations can leak some information and, provided enough knowledge of the implementation is at hand, a careful statistical analysis could even lead to the total recovery of these secret parameters (fig. 1).

---

\* Work done when both of these persons were research assistant at the UCL Crypto Group.

<sup>1</sup> Later modified to resist against it.

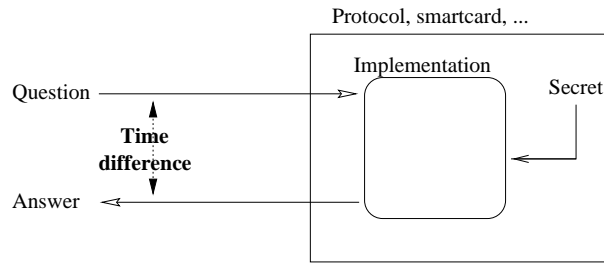


Fig. 1. The timing attack principle.

This idea was first presented by Kocher [Koc96], who laid the foundations of the basic ideas exploited in this paper. However, the results of Kocher were quite theoretical<sup>2</sup> and we found them rather difficult to exploit in practice. This paper presents an effective and efficient attack of a cryptographic algorithm running on a smart card. The first practical timing attack to our knowledge was described at the rump session of CRYPTO'97 by Lenoir. Our paper, however, develops quite different ideas.

Another problem of the attack presented by Kocher is that the attacker needs a very detailed knowledge of the implementation of the system he is attacking, as he has to be able to compute the partial timings due to the known part of the key. As for this paper, the knowledge needed is very limited, which makes the attack quite general and easy to carry out.

Last but not least, some completely new ideas, such as the attack of the square rather than the multiply, are presented.

We begin by presenting the model we are attacking and the characteristics it must present to be vulnerable. We then describe the attack that was carried out, as well as its results and possible improvements. Finally, we describe some countermeasures that would allow to defeat it.

We try to present both a formal and an intuitive view of the timing attack, our goal being to make the principle of the attack easy to understand, but also to provide a detailed enough description to allow the reader to implement it without encountering major problems.

## 2 The General Framework

We here give the characteristics that the system must present to be vulnerable, and briefly formalize the model in which our attack will be drawn.

---

<sup>2</sup> Kocher identified several targets for the timing attack and ran simulations to determine what the success rate would be for an attack of the modular multiplication, but did apparently not carry out the attack itself.

Given a message  $m$  as input, an algorithm  $A$  performs a computation (that we call a signature) using a secret key  $k$ . We note:

$M$ , the set of messages,

$K$ , the set of keys,

$S$ , the set of signed messages,

$A : M \times K \rightarrow S : (m, k) \rightarrow A(m, k)$ , the signature of  $m$  with the secret key  $k$ ,

$B = \{0, 1\}$ ,

$T : M \times K \rightarrow \mathcal{R} : m \rightarrow t = T(m, k)$ , the time taken to compute  $A(m, k)$ .

$O : M \rightarrow B : m \rightarrow O(m)$ , an oracle, based on our knowledge of the implementation, that provides us some information about the details of the computation of  $A(m, k)$ .

*Remark:* It may look surprising that the oracle does not depend on the key  $k$ , although the computation of  $A(m, k)$  does, but this is precisely the idea of this paper: typically, we want to build a decision criterion (formalized by the oracle) that will be meaningful or not, *depending on the actual value of some bit of the key*. By observing the meaningfulness of our criterion, we will deduce the bit value.

The scenario of our attack is the following: Eve disposes of a sample of messages and, for each of them, the time needed to compute the signature of the message with the key  $k$ . Her goal is to recover  $k$ , which can thus be considered as an unknown parameter rather than as a variable. To simplify our notations, we will thus simply note  $T(m)$  instead of  $T(m, k)$ .

To attack the bit  $i$  of the key  $k$ , Eve will use an oracle  $O$  to build two subsets of messages  $M_1, M_2 \subseteq M$ . We will denote the corresponding timings by the functions:

$$F_1 : M_1 \rightarrow \mathcal{R} : m \rightarrow F_1(m) = T(m)$$

$$F_2 : M_2 \rightarrow \mathcal{R} : m \rightarrow F_2(m) = T(m)$$

Suppose these two functions have the following properties:

$$\left\{ \begin{array}{l} \text{If } k_i = 0, \text{ then } F_1 \text{ is a random variable } v_1^0 \\ \qquad \qquad \qquad F_2 \text{ is a random variable } v_2^0 \\ \text{If } k_i = 1, \text{ then } F_1 \text{ is a random variable } v_1^1 \\ \qquad \qquad \qquad F_2 \text{ is a random variable } v_2^1 \end{array} \right.$$

and suppose that, for a parameter of these random variables  $\phi(v)$  (e.g. the mean or the variance) we have:

$$\phi(v_1^0) = \phi(v_2^0) \quad \text{and} \quad \phi(v_1^1) > \phi(v_2^1)$$

then with the following statistical test:

$$H_0 : \phi(F_1) \stackrel{?}{=} \phi(F_2)$$

$$H_1 : \phi(F_1) \stackrel{?}{>} \phi(F_2)$$

we deduce that if  $H_0$  is accepted with error probability  $\alpha$ , then  $i = 1$  with error probability  $\alpha$ .

In other words, this means that Eve is able to construct two samples of messages and two functions whose statistical behaviours will depend on the actual value of the bit  $i$ . By observing the relative behaviours of the two functions, Eve will be able to determine, with a certain error probability, the value of the bit  $i$ . Of course, couples of ciphertexts / decryption timings, with the same properties, could also be used.

### 3 Towards a Practical Attack

#### 3.1 The Implementation

We have attacked an RSA computation (without CRT), performed in an earlier version of the cryptographic library we developed for the CASCADE [Cas] smart card. It is interesting to note that this implementation was already naively protected against a timing attack, but that this protection turned out to be insufficient; we will come back on this later (section 9).

The computation in the smart card was:  $m^k \bmod n$ .

The algorithm is the left to right square and multiply (fig. 3.1).

```
x = m
for i = n - 2 downto 0
  x = x2
  if (ki == 1) then
    x = x · m
endfor
return x
```

Fig. 2. Square and multiply

Both the multiplication and the square are done using the Montgomery algorithm. The time for a Montgomery multiplication is constant, independently of the factors, except that, if the intermediary result of the multiplication is greater than the modulus, then an additional subtraction (called a reduction) has to be performed.

#### 3.2 A First Attempt: Attacking the Multiply

The most obvious way to take advantage of this knowledge is to aim our attack at the *multiply* step of the square and multiply. The idea is the following:

We start by attacking  $k_2$ , the second bit<sup>3</sup> (MSB first) of the secret key. Performing the Montgomery algorithm step-by-step, we see that, if that bit is 1, then the value  $m \cdot m^2$  will have to be computed during the square and multiply.

<sup>3</sup> We can of course suppose that the first bit of the key is always 1.

Now, for some messages  $m$  (those for which the intermediary result of the multiplication will be greater than the modulus), an additional reduction will have to be performed during this multiplication, while, for other messages, that reduction step will not be necessary. So, we are able to divide our set of samples in two subsets: one for which the computation of  $m \cdot m^2$  will induce a reduction and another for which it will not. If the value of  $k_2$  is really 1, then we can expect the computation times for the messages from the first set to be slightly higher than the corresponding times for the second set.

On the other hand, if the actual value of  $k_2$  is 0, then the operation  $m \cdot m^2$  will not be performed. In this case, our “separation criterion” will be meaningless: there is indeed no reason for which a  $m$  inducing a reduction for the operation  $m \cdot m^2$ , would also induce a reduction for  $m^2 \cdot m^2$ , or for any other operation. Therefore, the separation in two subsets should look random, and we should not observe any significant difference in the computation times.

Let us rewrite this a little more formally:

The algorithm  $A(m, k)$  could be split into  $L(m, k)$  and  $R(m, k)$  where  $L(m, k)$  is the computation due to the additional reduction at the multiplication phase for bit  $k_2$  and  $R(m, k)$  the remaining computations. This gives for the computation times:  $T(m) = T^L(m) + T^R(m)$ , where  $T^L(m)$ ,  $T^R(m)$  are the times to compute  $L(m, k)$  and  $R(m, k)$  respectively.

The oracle  $O$  is:

$$O : m \rightarrow \begin{cases} 1 & \text{if } m \cdot m^2 \text{ is done with a reduction,} \\ 0 & \text{if } m \cdot m^2 \text{ is done without a reduction.} \end{cases}$$

As in section 2, define

$$\begin{aligned} M_1 &= \{m \in M : O(m) = 1\}, \\ M_2 &= \{m \in M : O(m) = 0\}, \\ F_1 : M_1 &\rightarrow R : m \rightarrow F_1(m) = T(m), \\ F_2 : M_2 &\rightarrow R : m \rightarrow F_2(m) = T(m). \end{aligned}$$

We have

$$\begin{cases} F_1 = T^R & \text{if } k_2 = 0 \\ F_1 = T^R + T^L & \text{if } k_2 = 1 \end{cases}$$

while,

$$F_2 = T^R$$

independently of the value of  $k_2$ .

Now, analyzing the mean as parameter  $\phi$ , and testing:

$$\begin{aligned} H_0 &: \phi(F_1) \stackrel{?}{=} \phi(F_2) \\ H_1 &: \phi(F_1) \stackrel{?}{\neq} \phi(F_2) \end{aligned}$$

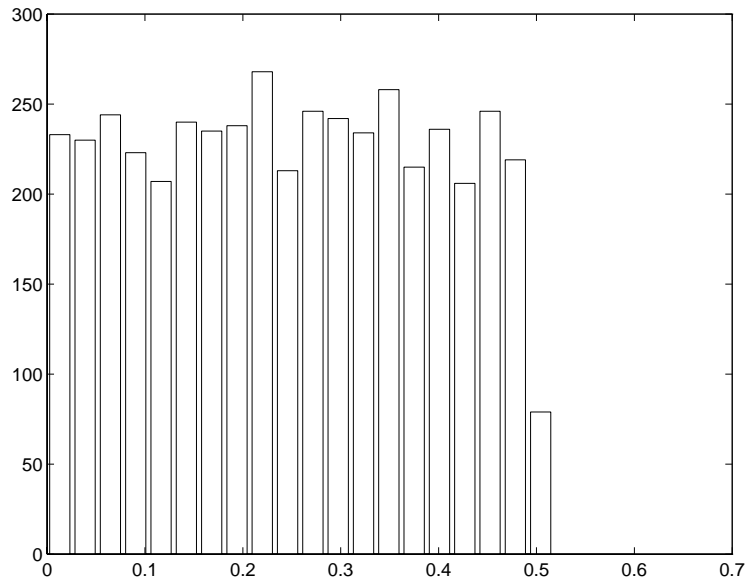
should reveal the value of  $k_2$ .

Once this value is known, we can simulate the computation up to the multiplication due to bit  $k_3$ , attack it in the same way as described above, and so on for the next bits.

### 3.3 Problems

Using the previous attack, we were able to recover 128-bit keys by observing samples of 50 000 timings.

However, this method is not fully satisfying. Mainly two problems arise: Firstly, the operations we observe are multiplications by a constant value  $m$ , and these operations seem to be much more correlated than expected. Rather surprisingly, we observed that, while the probability for an additional reduction to be necessary when the two factors and the modulus are random is about 0.17, this probability will, when the modulus and one factor are fixed, vary between 0 and 0.5, depending on modulus and factor (see figure 3). So, although it seems difficult to explain theoretically, our selection criterion seems to be highly biased.



**Fig. 3.** With fixed modulus, probability for an additional reduction to take place when multiplying by a constant factor. The test has been carried out with 4 500 factors and, for each of them, with 20 000 multiplications.

Secondly, the decision we have to make is of the type: “are these two samples different *or not*”. That is, we have to decide, on the basis of a finite set of measures, whether the differences we observe between the two sets is significant or not. Statistics can be of some help, but not as much as could be expected. As we said before, Montgomery multiplication by a constant seems to be a biased operation, so that the two subsets we build *always* appear different, even if the corresponding bit is 0. The answer to the statistical tests we tried is *always*

positive, at a very high level of confidence. So, the question we have to ask is rather: “are these two samples ‘very’ different, or simply different?”.

Luckily, statistics can though be used to answer that question: we can simply decide that our two subsets are very different when the observed value for the statistic is ‘very high’, and that they are simply different when the value is ‘not so high’. The problem will now be to decide what “very high” and “not so high” mean<sup>4</sup>, and we will have to tune up some swap value for each key we attack. Some heuristics can help us in this tuning operation, but we will not describe them here, as there is a more efficient approach:

Aiming our attack at the *square* operation solves both of these problems.

### 3.4 Attacking the Square

There is a more subtle way to take advantage of our knowledge of the Montgomery algorithm: instead of the multiplication phase, we could turn ourselves to the *square* phase.

The idea is quite similar to that of section 3.2: suppose we know the first  $i - 1$  bits of the key and attack the  $i$ th. We begin by executing the first  $i - 1$  steps of the square and multiply algorithm, stopping just before the possible - but unknown - multiplication by  $m$  due to bit  $k_i$ ; we denote by  $m_{temp}$  the temporary value we obtain.

First, we suppose  $k_i$  is set. If this is the case, the two next operations to be performed are

1. multiply  $m_{temp}$  by  $m$ ,
2. square the result,

and both of these operations will be done using the Montgomery algorithm. We simply execute the multiplication and then, for the square, determine whether an additional reduction will be necessary or not. Doing this for every message, we divide our samples set in two subsets  $M_1$  (additional reduction) and  $M_2$  (no reduction).

Next, we suppose  $k_i = 0$ . In this case, no multiplication will take place, and the next operation will simply be

$$m_{temp}^2.$$

Once again, we divide the samples set in two subsets  $M_3$  and  $M_4$ , depending on whether this square requires a reduction or not.

Clearly, only one of these separations makes sense, depending on the actual value of  $k_i$ . All we have to do now is to compare the separations: if the timing

<sup>4</sup> For example, attacking a 128-bit key using 50 000 samples and the  $\chi^2$  test, a typical observed value for the statistic was about 4300, which is much, much higher than  $\chi_{0.95}^2$ . We can however decide that values above 4320 correspond to “very different”, while values beyond 4320 correspond to “simply different”. This may look tedious on a theoretical point of view, but works well in practice and allowed us to recover the secret key.

difference between  $M_1$  and  $M_2$  is more important than that between  $M_3$  and  $M_4$ , then conclude  $k_i = 1$ , otherwise, conclude  $k_i = 0$ .

Back to formalization, to attack bit  $k_i$  knowing bits  $k_0, \dots, k_{i-1}$ , we split the algorithm  $A(m, k)$  into  $L(m, k)$ , which is the computations due to the additional reduction at the square phase at step  $i + 1$ , and  $R(m, k)$ , the remaining computations.

Compute

$$m_{temp} = (m^b)^2 \quad \text{where } b = k_0 k_1 \dots k_{i-1}.$$

We need two oracles,

$$O_1 : m \rightarrow \begin{cases} 1 & \text{if } (m_{temp} \cdot m)^2 \text{ is done with a reduction,} \\ 0 & \text{if } (m_{temp} \cdot m)^2 \text{ is done without a reduction,} \end{cases}$$

$$O_2 : m \rightarrow \begin{cases} 1 & \text{if } (m_{temp})^2 \text{ is done with a reduction,} \\ 0 & \text{if } (m_{temp})^2 \text{ is done without a reduction.} \end{cases}$$

Define

$$\begin{aligned} M_1 &= \{m \in M : O_1(m) = 1, \} \\ M_2 &= \{m \in M : O_1(m) = 0, \} \\ M_3 &= \{m \in M : O_2(m) = 1, \} \\ M_4 &= \{m \in M : O_2(m) = 0, \} \\ F_k : M_k &\rightarrow R : m \rightarrow F_k(m) = T(m), \quad \text{for } 1 \leq k \leq 4. \end{aligned}$$

If  $k_i = 1$ , we have

$$\begin{cases} F_1 = T^R + T^L \\ F_2 = T^R \\ F_3 = F_4 \end{cases} \quad (= T^R + T^L \cdot O_1, \text{ but this is not important})$$

and thus  $\mu(F_1) > \mu(F_2)$ , while  $\mu(F_3) = \mu(F_4)$ .

On the other hand, if  $k_i = 0$ , we have

$$\begin{cases} F_1 = F_2 \\ F_3 = T^R + T^L \\ F_4 = T^R \end{cases}$$

and thus  $\mu(F_3) > \mu(F_4)$ , while  $\mu(F_1) = \mu(F_2)$ .

Testing which of these conditions is true should reveal the value of  $k_i$ .

*Remark:* The last bit cannot be revealed by this attack and must thus be guessed.

This attack does not suffer from the problems mentioned in previous section:

- Firstly, the operation we are observing (i.e. the square) does not involve a constant factor, and its behaviour appears to be much less biased than for the multiplication.

- Secondly, we do not have anymore to decide whether a separation makes sense or not: we have now to compare two separations and decide which is the most significant. We are thus relieved of the difficult task to tune up an appropriate swap value for a given key.

Using this attack, we were able to recover 128-bit keys with 20 000 timings. Some keys were disclosed with only 12 000 timings.

## 4 Statistics

We have not yet said very much about the statistics we have to use to compare samples, and that is mainly because they were not very useful in practice. We tried several of the tools that statistics offer to compare two samples, such as the Chi-square, Student, Hotteling, and even a test from non-parametric statistics, the Wald-Wolfowitz [Sie56] test. None of them offered really efficient results. Chi-square and Student provided criteria upon which a right decision could be made, but a simple comparison of the means of the two populations allowed the same decision, with a similar, if not better, success rate.

There is, however, a possible use for statistics: as the Chi-square test, for example, does not seem to yield the same errors as the means comparison, it can be used to provide us with some sort of level of confidence in the goodness of our choices. When both tests agree on the value of some bit, it is more probably right than when they disagree, and this can help us to detect erroneous deductions. We implemented this in practice, and it appeared to be of some help, but did not produce any significant breakthrough in efficiency.

The reason for this uselessness of statistics is probably the one we mentioned before, that is, Montgomery multiplications with constant modulus are not independent events, and our decision criteria are thus biased<sup>5</sup>. Perhaps a better understanding of *why* this bias appears would allow to derive an useful statistical test? Up to now, however, they seems to be limited to a role of confirmation.

## 5 Error-Detection

One remarkable property of our attack is that it has an error-detection property. This is easy to understand on an intuitive point of view: remember that the attack basically consists in simulating the computations until some point, then build two decision criteria, with only one of them making sense, depending on the searched value, and finally decide the bit value by observing which criterion actually makes sense. Also note that each step of the attack relies on the previous ones (we need the previous bit values to simulate the computation).

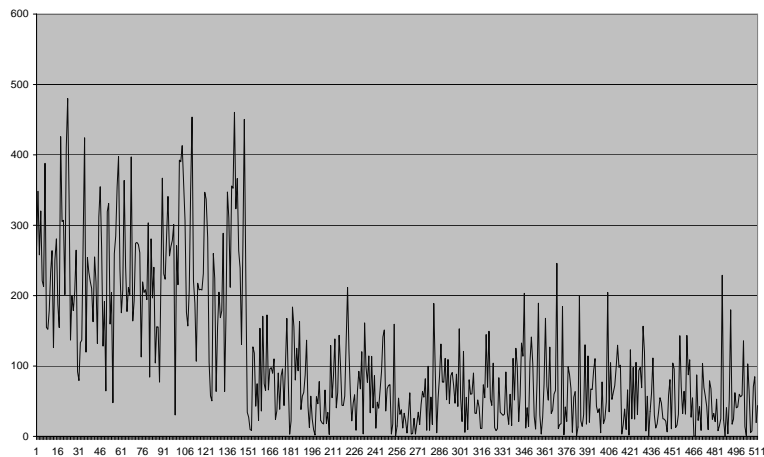
Now, suppose we made an erroneous decision for the value of bit  $k_i$ . In the following step, we will not correctly simulate the computations, so that the value

---

<sup>5</sup> This is also true for square, even it is at a much less extend than for multiplication by a constant factor.

$m_{temp}$  we will obtain will not be the one involved in step  $i + 1$ . Our attempts to decide whether the Montgomery multiplications will involve an additional reduction or not will thus not make sense, and the criteria we will build will *both* be meaningless. This remains true for the following bits.

In practice, this translates to abnormally close values for the two separations: while, as long as the choices were right, the two separations were generally<sup>6</sup> easy to distinguish, one of them being clearly more significant than the other, they appear much more similar (and both bad) after an erroneous choice has been made. This fact is well illustrated in figure 4, showing the attack of a 512-bit key on the basis of 350 000 observations. The decision criterion is simply the difference between the mean times for the two subsets, and the graph shows the absolute value of  $diff_1$  (the difference between  $M_1$  and  $M_2$ ) minus  $diff_2$  (difference between  $M_3$  and  $M_4$ ). Clearly, an error has occurred near bit 149.



**Fig. 4.** Detection of an error for a 512-bit key

Once an error has been detected, it is not difficult to take back, make a different choice for the last chosen bit, and go ahead a few steps to see if things go better; if they do not, then we go back two steps, change the bit value, and so on.

In practice, this error-correction scheme allowed us to reduce significantly the amount of measures needed. Samples of 10 000 timings, for example, were sufficient to recover 128-bit keys, and some of them were revealed with as few as 6 000 timings.

---

<sup>6</sup> There are however some tedious cases, where the two criteria are uneasy to differentiate although no error has been made. That is why it is better to wait until several contiguous low values are observed before to conclude to an error.

## 6 Practical Results

Our attack was first implemented in Visual C++ 4.2 on a 200 MHz PentiumPro PC under Windows NT.

Timings were collected using an emulator of the CASCADE smart card, that was able to monitor the number of cycles between two points. This seems to be quite a realistic scenario: the amount of measures required for a real attack of the electronic device would probably be slightly larger, to filter out additional noise, but we believe it should not grow too much.

With about 10 000 samples (couples messages, time for modular exponentiation), we were able to break 128-bit keys, at a rate of about 4 bits/s. The speed for a 512-bit key was of a little more than 1 bit/minute and approximately 350 000 samples were needed. The implementation was not optimized for speed.

Our results summarize as follows:

Key size	Result			
	without error correction		with error correction	
	sample size	speed	sample size	speed
64	1 500–6 500	> 20 bits/s	1 500–4 500	> 20 bits/s
128	12 000–20 000	2 bits/s	6 000–10 000	4 bits/s
256	70 000–80 000	1 bit/4s	40 000–50 000	1 bit/2s
512	±350 000–400 000	1 bit/65s	200 000 – 300 000	1 bit/37s

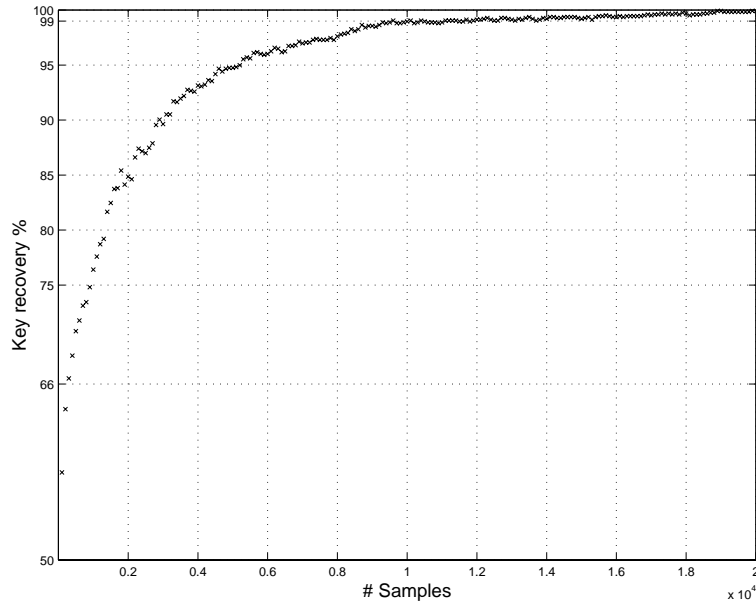
However, these results correspond to a very high success rate: the error-correction algorithm we implemented was very simple and allowed us to correct errors only if they occur for a very small percentage of the bits. Experiments showed that the sample size grows very fast with the desired success rate (see e.g. figure 5).

A more sophisticated algorithm, that would for example explore several choices when a possible error is detected, handle the case of two successive errors, . . . , would probably resist to a higher error rate, thus allowing to reduce drastically the amount of measures needed. As most of the computational effort consists in simulating the exponentiation steps for a large amount of data, the attack would also be very easy to parallelize.

## 7 Remarks about The Attack

**Accuracy of measures** The timing variation we are basing our attack on is very small regarding the total computation. For example, a 512-bit exponentiation on the CASCADE chip takes about 7 400 000 cpu cycles, and the variation we are trying to detect is only 422 cycles long!

The accuracy in measures is therefore of great importance. As the rounding effect induced by less accurate measures can be considered as noise, a greater amount of measures would still make the attack possible, but the sample size would rapidly grow.



**Fig. 5.** Sample size / success rate dependence

**Choice of Messages** It is worth noting that we simply need to be able to determine the value of  $O(m)$  for each message: we do not have to *build* messages for which the oracle will have a given value. This is important because many protocols will not accept to sign arbitrary message, but will require for these to have a specific format (e.g. to exhibit some redundancy). As long as we are able to trace the transformations preceding the modular exponentiation, the attack can be carried out.

**Knowledge of the Implementation** We also insist on the fact that we do not need too many details about the implementation of the cryptographic algorithm itself. All we have to know is that the exponentiation is square-and-multiply with Montgomery multiplication. It is amazing to note that one of the authors began with the timing-attack program before the CASCADE library was available for him. He thus decided to build his own exponentiation routine as a first target. When he finally received the CASCADE library, he discovered that his attack program did not need any modification to work against it.

**Possible Improvements** Even if the number of samples at disposal is not sufficient to discover the complete key, the attack is likely to reveal a part of it : we observed that samples twice as small as the required size for complete recovery could reveal 3/4 of the key before the first error occurs (as we have seen,

nothing significant is produced afterwards). Therefore, any method allowing Eve to guess a part of the key before carrying out the attack or to deduce the last bits once the first ones are known can dramatically improve performances.

Consider for example the frequent case where:

- the public exponent  $e = 3$ ,
- the secret exponent  $k$  is calculated such that  $k \cdot e = 1 \pmod{(p-1)(q-1)}$  (the important point is that we do not use the  $lcm(p-1, q-1)$ ),
- $p$  and  $q$  have the same  $l$ -bit length.

We have:

$$k \cdot 3 - s(p-1)(q-1) = 1, \quad \text{with } s = 1 \text{ or } 2,$$

$$k \cdot 3 = \begin{cases} 1 + 1[n - (p+q) + 1] \\ 1 + 2[n - (p+q) + 1] \end{cases}$$

$$k = \begin{cases} [n - (p+q) + 2]/3 \\ [2n - 2(p+q) + 2]/3 \end{cases}$$

Because the length of  $n$  is twice the length of  $p$  and  $q$ , there is a great probability that the  $l-3$  most significant bits (depending on the length of the propagation of a possible carry due to the subtraction by  $p+q$ ) of the key  $k$  are the  $l-3$  first bits of  $n/3$ . Eve can thus start the attack at the first unknown bit.

## 8 Other Targets and Further Research

The attack could easily be extended against some variants of the square and multiply algorithm. The implementation of RSAREF, for example, processes the bits two by two, performing two square followed by a multiplication by 1,  $m$ ,  $m^2$  or  $m^3$ , depending on whether the bits are 00, 01, 10 or 11. The attack could quite easily be adapted to this case.

Other cryptosystems involving a modular exponentiation are of course subject to the same attack. Consider for example the Diffie-Hellman key exchange protocol: to build a common secret parameter, Alice and Bob exchange the values  $g^x$  and  $g^y$ , where  $g$  is public and  $x$ ,  $y$  are secret values, known only by Alice and Bob, respectively, but often kept constant. The common parameter, that only Alice and Bob can compute, is obtained by computing  $(g^y)^x$  or  $(g^x)^y$ .

Now, suppose Alice wants to discover Bob's secret parameter  $y$ . She chooses several random values  $x_1, \dots, x_N$ , sends the values  $g^{x_i}$  (e.g. pretending to be someone different each time) and collects the corresponding response times (which can be, for example if Bob is a smart card, measured with very good accuracy). Clearly, the conditions of our attack are fulfilled. As a typical value for a Diffie-Hellman key size is 160 bits, a few thousand exchanges would suffice to discover the secret key.

Other protocol could probably be attacked in the same way. It must however be noted that the basis of the exponentiation (i.e., the parameter  $x$  in  $x^k$ ) has to

be known for the attack to be carried out as described here. Therefore, systems such as DSS, ... seem less vulnerable, although a more detailed study should have to be carried out.

One important weakness of our attack is that it cannot be carried out against systems using the Chinese Remainder Theorem for modular exponentiation. Kocher [Koc96] proposes some leads for a timing attack of the CRT, but it is not known whether such an attack would be practicable.

When developing the attack, we faced many difficulties on building a rigorous mathematical model explaining *why* things work. In fact, we encountered more than once the strange situation of building a model which should reveal some information, implementing it, and discovering that the system behaves differently than expected, *although the information is well revealed*. It seems that other researchers interested in the timing attack have faced the same problems with theory. A complete theoretical model would of course be useful, although we believe it is a real challenge.

## 9 Countermeasures

Three countermeasures come to mind when we try to protect ourselves against the above attack.

The first one is to modify the Montgomery algorithm so that an additional subtraction is always carried out, even if its result is simply discarded afterwards. This modification is easy to carry out, does not decrease performance very much and clearly defeats the attack. One must however be very careful when implementing it and make sure to remove *all* time variation. For example, it later turned out that the CASCADE implementation we were attacking was using this countermeasure, but in a too naive way: additional subtraction was always carried out, which hid most time variation, but the difference between discarding and copying its result still induced a difference of some clock cycles (422 in 512-bit version). This countermeasure did not prevent the attack to be carried out, it simply made it a bit more difficult. Experiments showed that, with this naive protection removed, the attack would have been efficient with ten times less samples.

Even with a careful implementation, it cannot be guaranteed that this would make the system immune to *any* type of timing attack, only against those which exploit the reduction of the multiplication algorithm.

Another countermeasure, suggested by [Koc96], would be to use some blinding: before computing the modular exponentiation, choose a random pair<sup>7</sup>  $(v_i, v_f)$  such that  $v_f^{-1} = v_i^e$ ; multiply the message by  $v_i \pmod n$  and multiply back the output by  $v_f \pmod n$  to obtain the searched result. As Eve can no more simulate the internal computations, she can hardly exploit her knowledge of the timing measurements.

It is worth noting that the attack is quite general, in the sense that it was not directed against the peculiar case of a Montgomery multiplication, but against

---

<sup>7</sup> [Koc96] proposes a way to generate such pairs at a reasonable cost.

the fact that this algorithm is constant-time, except for a potential final reduction. This means that the use of other modular multiplication schemes, such as the standard versions of the Barrett or Quisquater algorithms, would not protect the system against our timing attack.

However, Dhem [Dhe98] recently proposed an improvement of these multiplications schemes, allowing several modular multiplications to be chained with only *one* extra reduction being performed after the last multiplication. This scheme seems to be especially interesting here, as it would suppress our attack's main target.

An often proposed countermeasure is simply to add random delays to the algorithm, in order to hide time variation. We insist on the fact that this countermeasure is inefficient, as it is equivalent to adding white noise to a source. Such noise can easily be filtered out for a linear increase in sample size.

## 10 Conclusion

This paper shows that the timing attack represents a practical, important threat against cryptosystems implementations, namely in the case of a smart card, where the attacker can quite easily collect large amount of message decryptions and measure time with high precision.

It is also shown that a very good understanding on the way the timing attack works is necessary in order to be able to implement efficient countermeasures. The implementation we broke, for example, possessed some protection means, which turned out to be insufficient.

It is important to note that the attack is quite general, in the sense that it does not require a very detailed knowledge of the implementation: all we have to know, besides some general hardware characteristics such as the word size, is that the modular exponentiation is done using the square and multiply and Montgomery algorithm. Computation details, timings necessary for specific operations, . . . , are not necessary. This is an important improvement on Kocher's attack.

As shown in previous section, the attack is also general in the sense that it could have been directed against other classical modular multiplication schemes, if they are used in there standard form and not with Dhem's improvement.

In view of these results, the design of the CASCADE smart card has been modified to make it immune against the timing attack. It is however our belief that few smart cards take care of this, and that similar attacks could be successfully conducted against many of them.

## 11 Acknowledgements

The authors wish to thank Gael Hachez for useful comments and technical help, and HP Labs, Bristol, UK for the grant of the PCs which were used to carry out the attack.

## References

- [Cas] Cascade (Chip Architecture for Smart CARds and portable intelligent DEvices). Project funded by the European Community, see <http://www.dice.ucl.ac.be/crypto/cascade>.
- [Dhe98] J.F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. PhD thesis, Université catholique de Louvain - UCL Crypto Group - Laboratoire de microélectronique (DICE), May 1998.
- [Koc96] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, Santa Barbara, California*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
- [Ler98] P.-A. Leroux. Timing cryptanalysis : Breaking security protocols by measuring transaction times. Master's thesis, Université catholique de Louvain - UCL Crypto Group, June 1998.
- [RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. In *Proc. Communications of the ACM*, volume 21, pages 120–126. ACM, February 1978.
- [Sie56] S. Siegel. *Nonparametric Statistics*. McGraw-Hill, 1956.
- [Wil98] J.-L. Willems. Timing attack of secured devices (in French). Master's thesis, Université catholique de Louvain - UCL Crypto Group, June 1998.