TIMING ATTACK: WHAT CAN BE ACHIEVED BY A POWERFUL ADVERSARY?

Gaël Hachez, François Koeune, Jean-Jacques Quisquater

UCL Crypto Group,
Place du Levant 3, B-1348 Louvain-la-Neuve, Belgium.
{hachez,fkoeune,jjq}@dice.ucl.ac.be
URL: http://www.dice.ucl.ac.be/crypto

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. This idea, due to Kocher [Koc96], was developed in [DKL+98], were a timing attack against an actual smart card implementation of the RSA[1] was conducted.

The paper's conclusion was that, however impressive, the obtained results could be improved even further in several aspects, especially regarding the error-correction policy.

The paper first presents the basic principle of the timing attack, then briefly discusses several error-correction policies and describes the results we obtain implementing them on a parallel architecture of 4 processors PA8000 @ 180Mhz with 4 Gbytes RAM.

TIMING ATTACK

We begin by describing the main features of the timing attack. A more complete description can be found in [DKL+98].

We will attack an RSA implementation, performed in an earlier version of the cryptographic library we developed for the CASCADE [Cas] smart card.

[1] In fact, a pre-version of the cryptographic library the UCL Crypto Group developed for CASCADE[Cas]: in view of the devastating results of the timing attack, the final version was modified to make it immune against it.

The scenario of the attack is the following: Eve disposes of a sample of messages M 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.

To perform the timing attack, Eve needs a bit information about the implementation. We suppose she knows that the computation of $m^k \bmod n$ is performed using the well-known left to right square and multiply (fig. 1). She also knows that both the multiplication and the square are done using the Montgomery algorithm.

```
x = m
FOR i = n − 2 DOWNTO 0
    x = x^2
    IF (k_j == 1) THEN
        x = x · m
ENDFOR
RETURN x
```

Figure 1: Square and multiply

We will not describe the Montgomery algorithm in detail here. The only point of interest is that 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 at the end of the multiplication. This means, and that is the basis of our attack, that for some factors the multiplication time will be longer than for others.

A first target: 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.

We start by attacking k_2, the second bit[2] (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.

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

[2]Of course we can suppose that the first bit of the key is always 1.

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.

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.

The previous approach allowed us to recover 128-bit keys by observing samples of 50 000 timings. However, this approach was unsatisfying on several respects (see [DKL+98]) and another one, described below, turned out to be much more efficient.

Second target: 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 : suppose we know the first $i-1$ bits of the key and attack the ith. 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 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 $k_i = 0$.

This approach has several advantages on the previous one, but its most interesting feature for the current purpose is its error-detection property. This will be developed in next section.

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

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 building two decision criteria, with only one of them making sense, depending on the searched value, and finally deciding 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 bits 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 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[3] easy

[3]There are however some tedious cases, were 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.

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 2, 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.

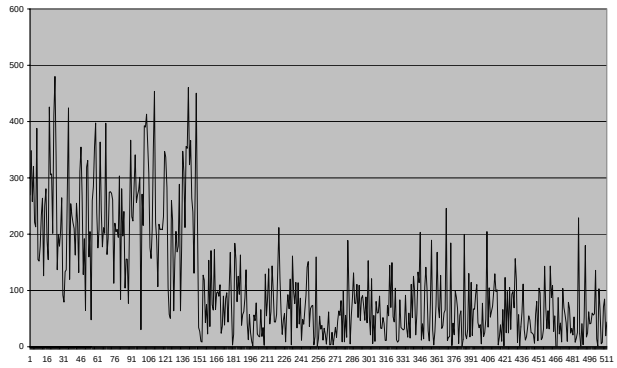
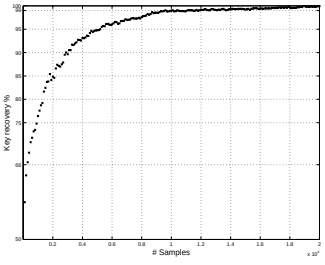


Figure 2: Detection of an error for a 512-bit key

Two things are worth noting about this figure: first, it can be seen that, even when the guesses are correct, some values of the criterion turn out to be very small. This suggests that, to detect an error, it would be better to consider the criterion on few consecutive bits instead of single ones. Secondly, it must be noted that this graph corresponds to a rather large sample; as the sample size decreases, errors will of course become much more difficult to detect.

Thanks to this error-detection property, it should be possible to build attack variants that would be robust against errors, as long as their proportion remains sufficiently low. This can be extremely important, as the sample size appears to decrease very fast when the error proportion grows (see figure below). A good correction policy should thus allow to reduce drastically the sample size. We will come back on this later.

Success rate	64 bits	128 bits	256 bits
75%	300	800	2500
90%	1000	2700	10000
95%	1750	5000	20000
99%	3750	9500	42000



PARALLEL IMPLEMENTATION

Once the samples have been collected, most of the computation time is spent in simulating the algorithm's behavior for each message and computing statistics. These computations can easily be parallelized.

We built a parallel implementation, in which the total sample is cut into parts and shared among several processors. Each processor handles its own subset (checking Montgomery multiplications, computing partial means, ...) and they only interact to share their statistics and make a common decision about the bit value. The level of parallelism is thus very high, and the speed is multiplied by the number of processors at disposal.

Our implementation on an HP/S-Class of 4 processors PA8000 @ 180Mhz, with 4 Gbytes SDRAM, allowed to reduce the time to break a 512-bit key to about 40 minutes. 128-bit keys were recovered in a few seconds.

Unfortunately, the real bottleneck for the attacker is not so much his computation power than the smart card's one. As a matter of fact, the only way to build the time sample is by forcing the smart card to sign every message. Such a processing of some hundred thousands different messages by the same smart card seems, at several respects, unrealistic. Therefore, of much more interest would be to be able to reduce the sample size we need for the attack. This is the purpose of next section.

ERROR CORRECTION POLICY

As we said before, it should be possible, by implementing an adequate error-correction policy, to reduce drastically the size of the needed sample. We therefore tried several ways to make the attack robust against an as big error proportion as possible.

Errors are always detected based on some criterion value. After many trials, we decided to use a criterion based on the decision criterion for the next bit value. We trace our decision criterion back from the last bit. If we find a bit with a small criterion value preceded by a window of bits with high criterion values, we concluded that the last bit was not correct. If we apply this criterion to figure 2, we find that bit 149 was the first bad bit. Quite surprising, more sophisticated criteria based on statistical models, did not reveal more efficient.

Basically, the error correction policy is the following:

1. start by performing the attack until the end, without trying any correction;

2. identify the last place P_1 were the criterion was high: we can reasonably suppose that the guess was correct until this point;
3. try a first correction, by changing the bit value at position $P_1 + 1$; continue the attack until its end and identify the last place P_2 were the criterion was high;
4. if P_2 is further than P_1, then the correction we tried was probably right; we can now recursively repeat the process: execute step 3, using P_2 instead of P_1;
5. otherwise, our correction attempt was incorrect: restore previous bit value, and try to change the bit value at position $P_1 + 2$, then $P_1 + 3$, ...
6. if the correct key is not found this way, we conclude the first error occurred *before* P_1; we thus find the last place, before P_1, were the criterion was high, and restart the same process.

Of course, every time a complete key is guessed, we check whether it is the right one (this is easily done in a public key cryptosystem) and stop as soon as the key is found.

However, one problem remains: we have seen that some bits are more difficult than others, in the sense that the criterion is not very clear there, and that the guess decision is taken in the wrong direction. The above algorithm will try to detect these errors and correct them. But what will happen if two "difficult bits" are consecutive? If this is the case, then the correction attempt will be directly followed by an error ; the criterion will thus remain low for subsequent bits and the algorithm will not detect it has done the good correction.

One solution is to try to correct not only single bits, but small bit windows. In the above algorithm, instead of changing a single bit value, we will explore all possibilities for three or four consecutive bits. We build a list with all possibilities and order this list from the most to the least probable using our decision criterion. For each possibility, we then try to complete the attack and proceed recursively as before. Note that, for this to be fully efficient, we need to use at least two different window sizes for the evaluation criterion window and for the correcting window. (otherwise we simply report the problem a bit further).

CONCLUSION

The previous paper [DKL$^+$98] showed that timing attacks are serious threats against cryptosystems implementation on devices where timing measurements

can be precisely done. This is especially the case for smart cards. The longest step (in time) is to get enough timing samples. The number of needed samples are quite large for usual key sizes. Further reduction of this number will increase the practicability of the attack.

In this paper, we focused on improvements able to reduce the number of samples needed to successfully recover a secret key. We showed that the timing attack enjoy good error-correcting properties. With our error-correcting method, we are able to reduce the number of samples by a factor two.

Key size	Results (number of samples)	
	without error correction	with error correction
64	1 500–6 500	1 500–4 000
128	12 000–20 000	5 000–8 000
256	70 000–80 000	30 000–40 000
512	350 000–400 000	150 00–200 000

We are still working on devising better error-correcting schemes because we believe that more information can be extracted from these timings.

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.

[DKL+98] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A practical implementation of the timing attack. In *Proc. CARDIS 1998, Smart Card Research and Advanced Applications*, 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.