Public Key Cryptosystem Timing Analysis: Evaluating Obscuring Techniques*

B. Canvel and C.T.J. Dodson,
Department of Mathematics,
University of Manchester Institute of Science and Technology,
Manchester M60 1QD, UK

August 27, 2000

Abstract

The security of encryption devices using common public key cryptosystems depends on the difficulty in the task of factoring large numbers used as moduli in the exponentiation. There is a drastic reduction in security if an attacker can obtain information about the relative computational effort to perform the necessary steps in exponentiation algorithms. Such information may be accessible from electromagnetic sensors placed in the vicinity of the device, or by submitting sequences of chosen texts to a public resource server. Here we report some simple computations which provide an upper bound on the quality of such methods of attack, by placing timing markers at procedure changes in an implementation using C^{++}. This type of approach may be helpful in evaluating the effectiveness of auxiliary procedures used to obscure the internal operations of a device.

KEYWORDS: RSA, EXPONENTIATION, SQUARE AND MULTIPLY, TIMING, SECURITY, OBSCURING, EVALUATION

1 Introduction

In public key encryption, for example using RSA, we compute $R = y^e (\mod m)$ or $R = y^d (\mod m)$ depending on whether we are encoding or decoding a message y; the (very large) modulus m is made public. The attack depends on discovering the time taken for the computation of R for a set of chosen y values; then, with some knowledge of the system design, it is feasible to deduce the exponents e and d. Here, we assume that the implementation used to perform the exponentiation is the square and multiply algorithm. In practice, suitable timing information may be obtained from data on power consumption by the processing device, using electromagnetic sensors. Such power traces may be noisy but can be cleaned by suitable statistical procedures.

Head's algorithm is a method for calculating the modular product of two integers. Normally, we would need to handle numbers of size m^2 when multiplying integers x and y $(0 \leq x, y < m)$ modulo m but Head's algorithm does the multiplication without introducing numbers bigger than $4m$, which makes the multiplication process faster. Using Head's algorithm and the square-and-multiply method, we are able to compute the modular exponentiation $x^n (\mod m)$ where $n = d_k d_{k-1} \ldots d_1 d_0$ is the exponent in binary form using the following algorithm,

Lemma 1.1 (Modular exponentiation) `Let r=1`

```
While n>0
   Let d=n-2[n/2]
   If d=1 then
      r=xr (mod m) (using Head's algorithm)
   End If
   x=x^2 (mod m)  (using Head's algorithm)
   n=(n-d)/2
End While
```

We then have $r \equiv x^n (\mod m)$. *([3] p. 90)*

*Rump Session Presention, CRYPTO 2000, Santa Barbara, 19-24 August 2000

2 Kocher's Attack

In the original paper on timing attacks by Kocher [4] (cf also [5, 6]), the timing attack is carried out on the multiply step of the square and multiply algorithm. As a hypothesis, we assume that the target system has a modular multiplication algorithm that is very fast in general but can sometimes take longer than an entire modular exponentiation. In the algorithm, for some values of x and exponent r, the calculation of $xr(\mod m)$ will be slow, and, because we know the implementation algorithm, it is easy to find those values for which it is slow.
The exponent is odd, so we start our search on the second bit of this exponent, setting x_1 to 1. Now working on the second bit of the exponent, and using our samples, we can deduce that the second bit of n is 0 whenever the exponentiation time is fast when $xr(\mod m)$ is expected to be slow and 1 whenever both operations are slow. This can be explained by the fact that if the exponent bit is set to 1 then the modular multiplication will be performed and the overall time will be increased. The operation is carried out for subsequent bits from the exponent until the full exponent has been recovered. In [2] is derived a more practical version of this attack, not requiring such in-depth knowledge of the implementation of the algorithm.
We assume that the method used to carry out the multiplication runs in constant time, independently of the factors except when the result of the multiplication is negative and then a further addition has to be performed, called the reduction step. As above, we set n_1 to 1. If the second bit of the exponent is 1, then the modular multiplication step in the square and multiply algorithm will be performed together with the squaring step. This yields two cases:

- reduction has to be performed
- no reduction.

If the second bit of the exponent is set to 0 though, this modular multiplication step will not be performed. We then have three samples which can be analysed using statistical analysis.
Taken from [2] is an example of a signature algorithm using RSA and a private key k :

```
M, set of messages K, set of keys S, set of signed messages
A : MxK -> S : (m,k) -> A(m,k), signature of m with secret key k
B={0,1} T : M x K -> R : m -> t = T(m,k), time taken to compute
A(m,k) O : M -> B : m -> O(m), an oracle, based on what we know
about the implementation, that provides some information about the
computation of A(m,k).
```

Based on what was said previously in this section, we can divide this algorithm into two parts, namely $L(m,k)$, the computation due to the additional reduction step for the second bit of the exponent, and $R(m,k)$, the remaining computations. Then the computation times are as follows : $T(m) = T^L(m) + T^R(m)$ where $T^L(m)$ and $T^R(m)$ are the times to compute $L(m,k)$ and $R(m,k)$. The oracle O is

$$O : m \mapsto \begin{cases} 1 & \text{if } mm^2 \text{ is done with a reduction} \\ 0 & \text{if } mm^2 \text{ is done without a reduction} \end{cases}$$

Let us define,

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

We have,

$$\begin{cases} F_1 = T^R & \text{if } k_2 = 0 \\ F_1 = T^R + T^L & \text{if } k_2 = 1 \\ F_2 = T^R & \forall k_2 \end{cases}$$

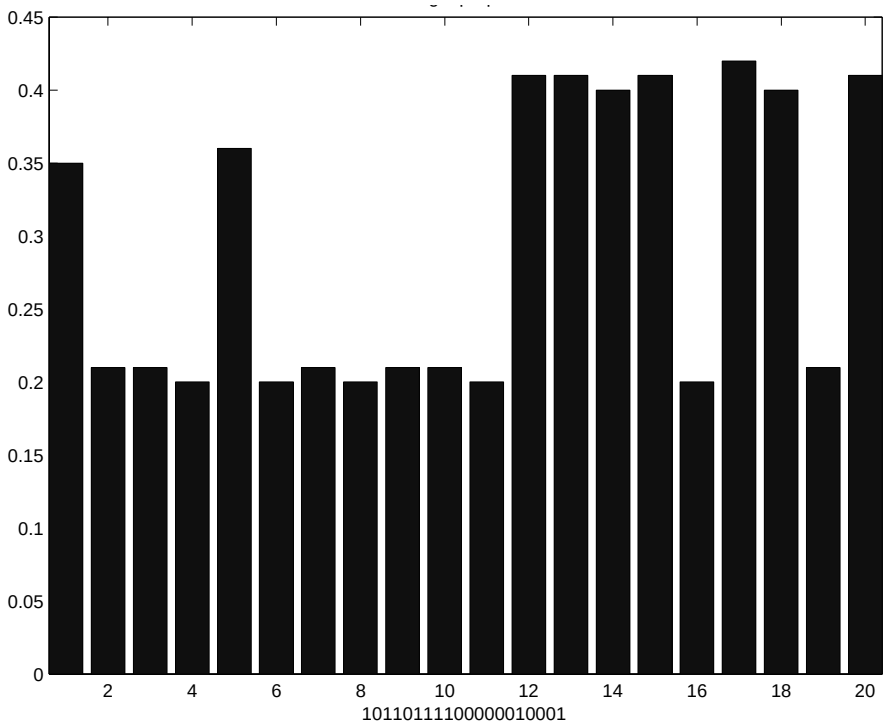

Figure 1: *Timings for an encrypting key of size 20 bits with p and q of size 195 bits. The key bits can be read directly from the diagram, right to left.*

Given these sets, we can analyze the calculation of ϕ for F_1 and F_2 and test

$$H_0 : \phi(F_1)? = \phi(F_2)$$
$$H_1 : \phi(F_1)? \neq \phi(F_2)$$

which should give us the value of k_2. In theory, if H_0 is accepted with error probability α then $k_2 = 1$ with error probability α and we can use the χ^2-test for significance estimates. We can continue similarly for the next bits until the whole key has been recovered. Another way of implementing the attack is by analyzing the square step instead of the multiplication one. This can also be seen in [2].

3 Timing Tags in C^{++} Code

To obtain timings in C++, we used a library called timer.h, written by Roque D. Oliviera; we ran it under Unix as it uses some of the system variables. The idea is to create an object of type timer and to initialise it at the start of the required timed portion of the program and get a value in milliseconds at the end of this portion of code. Here, we require a time for each exponent bit in the square and multiply algorithm. At the end of each exponentiation, the timings obtained are stored into an array which is returned from the exponentiation function.
For generating provable primes we used Maurer's algorithm in an implementation and a C++ library called LIP written by Lenstra [7]. From this library, we use the function PROVABLE_PRIME. To obtain the sample timing results for a simple message we used following procedure.

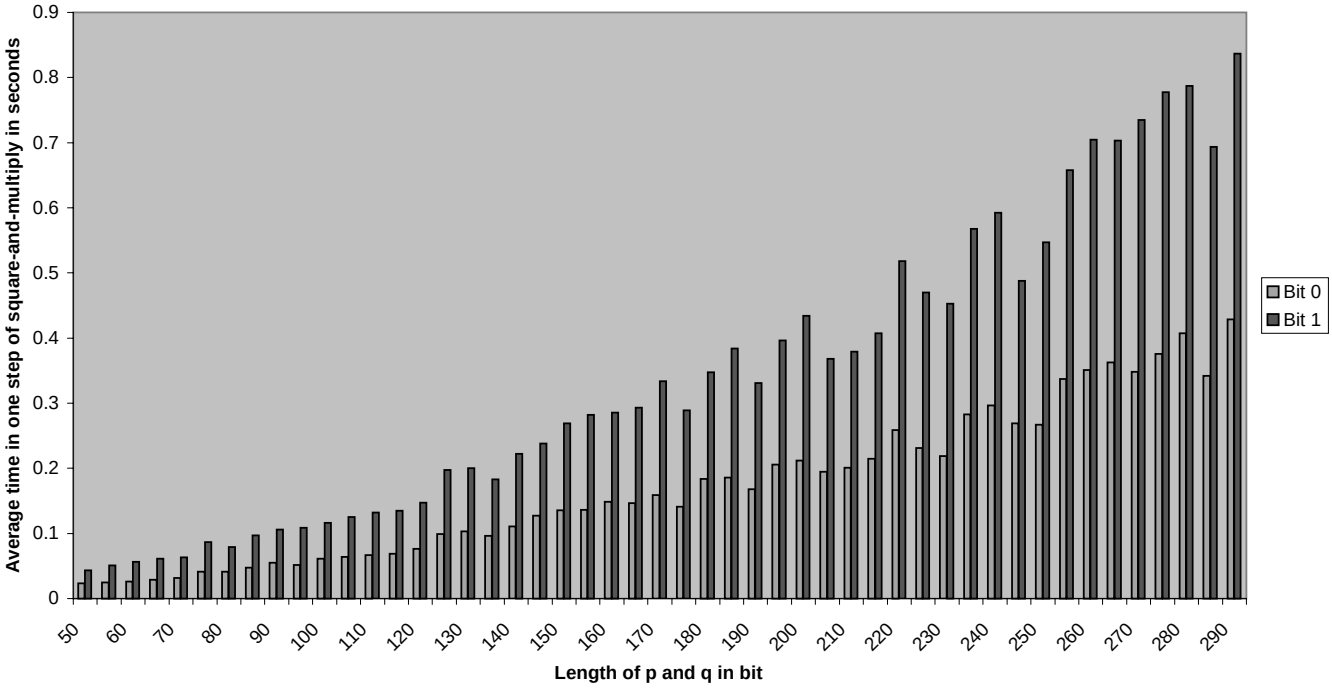

Figure 2: *Growth of average time in one step of square and multiply using 20 bit encrypting key, for bit 0 and bit 1 as the size of p, q increases from 50 to 290 bits.*

```
FOR p,q of size p_min to p_max in steps of p_incr
   FOR e of size e_min to e_max in steps of e_incr
      Generate p and q of the required size and
      so that q is not equal to p
      Compute e
      Compute phi
      Set e=phi
      WHILE (e,phi) not equal to 1
         Generate a prime e of the required size
      END WHILE
      Compute d so that e*d is congruent to 1 modulo phi
      Save e*d mod phi into check.dat
      Save (e,phi) into check.dat
      Set message=12345678
      Encrypt message as encr
      Decrypt encr as decr
      Save data used into data.dat
      Save the timings into timing.dat
      IF decr is not equal to encr
         Save ''Error'' into check.dat
      ELSE
         Save encr and decr in decimal form
         into check.dat
      END IF
   END FOR
END FOR
```

The full code is available in the thesis [1]. We have obtained results for p and q of length varying from 50 to 290 bits in steps of 5 bits and e of length varying from 20 to 100 bits also in steps of 5 bits. The

code was run on a computer with two 150Mhz SPARC processors running Solaris 2.7 and 320 Mb RAM. As p and q increase, the gap between the time intervals for a zero and a one increases significantly. For example, looking at Figure 1, corresponding to p and q of length 195 bits, we see very clearly that for a time below 0.2 seconds, a 0 is being processed and for a time above 0.35 seconds, a 1 is being processed. The difference in timing increased slightly more than in proportion to the size of p and q, as may be seen in Figure 2.

4 Conclusions

It is possible to read off the value of the exponent by looking at the time intervals obtained from the modular exponentiation algorithm. The length of the key only has an influence on the overall time of execution of the modular exponentiation algorithm but the time intervals for computing one step of the square-and-multiply algorithm are the same regardless of the length of the exponent. Hence, making the encrypting (or decrypting) key bigger only increases the overall time taken to find its value but does not affect the complexity of the task.
The incorporation of an obscuring procedure in a device like that described above performing encryption or decryption would complicate, by spurious internal processes, power or timing data obtained by an attacker. The corresponding graph to Figure 1 would be less distinct if it was subject to such obscuring signals. Then the way to test the obscuring procedure is to compare its consequence with that resulting from a mixture of pure timing data and noise of appropriate type. A certain minimum level of noise would be needed to reduce the information content below that which presents a security risk. Since the timing data represents the best possible data obtainable by an attacker, the requisite amount of noise to obscure it acceptably can be measured, so the obscuring procedure may be evaluated.

References

[1] B. Canvel. **Timing Tags for Exponentiations in RSA** MSc Dissertation, UMIST 1999.

[2] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater and J.-L. Willems. A practical implementation of the timing attack UCL Crypto Group Technical Report Series, June 15, 1998 http://www.dice.ucl.ac.be/crypto/

[3] P. Giblin. Primes and Programming : An Introduction to Number Theory with Computing Cambridge University Press, 1993.

[4] P. Kocher Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems Crypto' 96, volume 1109 of Lecture Notes in Computer Science, pp. 104-113, Springer-Verlag, 1996 http://www.cryptography.com/

[5] P. Kocher, J. Jaffe and B. Jun Introduction to Differential Power Analysis and Related Attacks http://www.cryptography.com/

[6] P. Kocher, J. Jaffe, Benjamin Jun Differential Power Analysis in Advances in Cryptology : Proceedings of CRYPTO '99, Springer-Verlag, Berlin 1999.

[7] A.K. Lenstra. LIP, Anonymous ftp /usr/spool/ftp/pub/lenstra on flash.bellcore.com

[8] A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. **Handbook of Applied Cryptology** CRC Press, Boca Raton, 1996.