Timing Attacks on RSA

Mark van Cuijk

March 20, 2009

Abstract

In general, timing attacks are used to analyze differences in execution time that result from differences in input parameters of a cryptographic algorithm. These timing differences are often caused by optimizing algorithm implementations, but they may leak information about the input parameters. Using a timing attack, an adversary hopes to find secret information, like bits from a secret RSA exponent. This paper summarizes several algorithms used in RSA implementations and how timing attacks can be used to reconstruct the entire secret RSA exponent.

1 Introduction

In implementations of cryptographic algorithms, the author must always make a trade-off between performance and security. First presented in 1995 by Kocher, the runtime performance of a cryptosystem implementation may depend on the input parameters. He describes how the runtime of a certain implementations of computing $r \equiv b^e \pmod{n}$ is related to the exponent e. By measuring the runtime of several computations, he is able to deduce the value e. In a setting where this value represents the secret exponent of a Diffie-Hellman key exchange or the secret value in an RSA implementation, the information leaked by the timing attack may compromise the security of the entire cryptosystem.

The method described by Kocher is not applicable to RSA implementations that optimize the private key operation using the Chinese Remainder Theorem (see section 2.2). In 2000, Schindler introduced a timing attack to factor the RSA modulus N into it's prime factor p and q when the target implementation uses the Chinese Remainder Theorem, while the multiplications and squarings modulo the prime factors p and q are carried out with Montgomery's algorithm (see section 2.3).

In 2003, Brumley and Boneh published a method for reconstructing the secret RSA exponent from a webserver using an active timing attack. The method uses some known facts about the implementation of the RSA private key operation in OpenSSL.

After this introduction to the subject, this paper describes several algorithms that are used to implement modular exponentiation used for the RSA private key operation in chapter 2. In chapter 3 some timing characteristics of these algorithms are explained, combined with methods to recover information from this timing information. Some countermeasures to prevent an attacker to recover this information are briefly discussed in chapter 4. The paper then finishes with a conclusion.

2 Exponentiation in RSA

For a given ciphertext C, a receiver can compute the corresponding plaintext M by computing $C^d \pmod{N}$, where d is the secret RSA exponent and N is the modulus that is public. Although the modulus N is public, it's factorization into prime factors p and q is secret.

This chapter introduces several algorithms that can be used to implement the exponentiation of the RSA private key operation.

2.1 Binary algorithm

Given an a base b and an exponent e, the right-to-left binary algorithm is among the simplest algorithms to compute r, such that $r \equiv b^e \pmod{N}$:

```
1: r ← 1
2: y ← b
3: for i = 0 to NumberOfBits(e) − 1 do
4:     if Bit_i(e) = 1 then
5:         r ← (ry) mod N
6:     end if
7:     y ← y^2 mod N
8: end for
9: return r
```

This algorithm computes the result in such a way that the exponent is in the lead. Notice how

$$a^{10} = (a^2)^5 = (a^2)^4 a^2 = ((a^2)^2)^2 a^2 \qquad (1)$$

demonstrates how the right-to-left binary algorithm computes $r = a^{10}$.

2.2 Chinese Remainder Theorem

A very easy and effective optimization to perform when implementing the exponentiation for RSA is the use of the Chinese Remainder Theorem (CRT) [6]. This optimization can only be performed when implementing the private key operation, since it requires knowing the factorization of the modulus N into it's prime factors p and q. The CRT allows the the computation of $r \equiv b^e \pmod{pq}$ to be broken up into computing $r_p \equiv b^{e_p} \pmod{p}$ and $r_q \equiv b^{e_q} \pmod{q}$, such that $e \equiv e_p \pmod{p-1}$ and $e \equiv e_q \pmod{q-1}$. The values r_p and r_q can then be combined into r. Since the runtime complexity of exponentiation algorithms are related to the size of the exponent, computing the values r_p and r_q can be done much more efficiently than computing $r \equiv b^e \pmod{pq}$ directly.

2.3 Montgomery reduction

When computing $r \equiv b^e \pmod{N}$, the author of an implementation originally had two options:

1. first compute $r' = b^e$ and then reduce r' to r, such that $r \equiv r' \pmod{N}$; or
2. reduce each intermediate result r_i modulo N, such that it is congruent to r_i.

When choosing the first option, the intermediate results will get very large and take a lot of computer memory during execution of the algorithm. Besides the large memory consumption, performing computations on very large numbers takes a lot of time and therefore these implementations tend to be too slow to be practical. However, when choosing the second option, the intermediate result r_i is reduced modulo N on each step. Modular reduction has the same complexity as a division, which is a rather expensive operation compared to addition, bitwise operations and register shifts.

In [3], Montgomery describes an algorithm that can be used to replace the modular reductions, used in the second option, by a more efficient method. The algorithm introduces some one-time overhead by first converting b into a Montgomery representation b' and converting the result r' back from Montgomery representation to r, but reduces the time needed for every step of the loop in the algorithm.

For working with numbers modulo N in Montgomery representation, a value R must be picked, such that $R > N$ and $gcd(N, R) = 1$. For correct results, any value of R that meets these two requirements can be used. However, to use the Montgomery representation to optimize the exponentiation, the value of R can best be set to a power of two. In practice, implementations use the power of two that is one bit larger than N or whose size is rounded up to the next word boundary on the target archicture.

The following values can be computed during initialization, as they are going to be used multiple times:

- $R^{-1} \pmod{N}$ is used during modular multiplication and converting output values;
- $N' \equiv N^{-1} \pmod{R}$ is used to make the reduction step more efficient; and
- (optionally) $R \pmod{N}$ is used to convert the input values.

A number a, $0 \leq a < N$, can be converted into Montgomery representation by computing $a' \equiv aR \pmod{N}$, which requires a modular reduction to be performed. Converting the number a' back can be done by computing $a \equiv aR^{-1} \pmod{N}$. It is trivial to see that this results in the original value of a, as long as the requirement $gcd(N, R) = 1$ is met.

For numbers in Montgomery representation, operations like addition, subtraction, negation and (in)equality testing are unchanged. To demonstrate this for additions: $c' \equiv a' + b' \equiv aR + bR \equiv (a+b)R \equiv cR \pmod{N}$. A similar demonstration is possible for the other operations.

To compute $c \equiv ab \pmod{N}$ having $a' \equiv aR \pmod{N}$ and $b' \equiv bR \pmod{N}$ in Montgomery representation, the value c' must be computed, such that $c' \equiv cR \equiv abR \equiv aRbRR^{-1} \equiv a'b'R^{-1} \pmod{N}$. The algorithm REDC [3] computes such a value, ensuring that also $c' < N$ holds after it terminates:

```
1: m ← (c' mod R)N' mod R
2: t' ← (c' + mN)/R
3: if t' ≥ N then
4:     return t' − N
5: else
6:     return t'
7: end if
```

The correctness of the REDC algorithm is demonstrated in [3].

2.4 Karatsuba multiplication

The numbers that are being used in cryptographic systems are generally much larger than the word size of the processor that performs the computations. To be able to perform the computations, multi-precision algebraic operations must be used.

For additions this is pretty straight-forward, starting with the least significant word, the implementation can perform an addition on each word individually, while keeping track of a carry that might be added to the next word.

For multiplications this isn't possible. A naive multi-precision multiplication algorithm takes $O(n^2)$, but in 1962 Karatsuba discovered an algorithm to compute this value in $O(n^{\log 3})$ [5]. The algorithm computes $P = N_0 N_1$ by writing $N_0 = a_0 + a_1 w$ and $N_1 = b_0 + b_1 w$, such that $a_0 < w$ and $b_0 < w$, for some w. The product can now be written as

$$N_0 N_1 = a_0 b_0 + (a_0 b_1 + a_1 b_0) w + a_1 b_1 w^2. \quad (2)$$

Notice how computing equation 2 directly requires four multiplications. However, the term $a_0 b_1 + a_1 b_0$ equals $(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1$. Since $a_0 b_0$ and $a_1 b_1$ already need to be computed, replacing the coefficient for w in equation 2 removes two multi-precision multiplications and introduces only one new multiplication to compute, together with some cheaper additions.

What happens when using Karatsuba multiplication is that a single multi-precision multiplication operation involving large integers is replaced by three multi-precision multiplication operations involving integers of approximately half the size.

3 Timing characteristics

3.1 Binary algorithms

The running time of the algorithm introduced in section 2.1 depends on the number of bits in the exponent e that have the value 1, as line 5 in the algorithm is executed conditionally. Intuitively, one can suggest that timing information may leak the Hamming weight of the exponent, but Kocher shows in [2] that timing measurements can be analyzed to completely recover the exponent e.

In the paper, Kocher uses the right-to-left binary algorithm and determines the exponent e bit by bit. For each bit e_i, he assumes that the attack already has knowledge of all the bits $e_0 ... e_{i-1}$. To determine the entire key, start with bit e_0 — thus having no known bits — and repeat the procedure up to the last bit. The attacker performs a timing measurement for k computations of $b_i^e \pmod N$, such that the value N and each of the values b_i, $0 \leq i < k$, are known for the attacker.

For each measurement, the attacker determines

- t, the total time used for the exponentiation;
- c, the time the implementation used for processing the known bits $e_0 ... e_{i-1}$; and
- d, the time it takes to compute line 5 in the algorithm.

The total time t can be measured by supplying the input value b to the algorithm implementation and measuring the time it takes before the output value r is returned. The values c and d must be determined by the attacker in another way, which would probably require knowledge of the inner workings of the implementation or performing some other experiments to retrieve timing information. The value c depends on the number of known bits $e_0 ... e_{i-1}$ that have the value 1. The value d is a fixed time and is exactly the timing difference between processing a 0 bit and a 1 bit.

When these values are determined, the attacker can compute the time it takes to process the unknown exponent bits $e_i ... e_{\text{NumberOfBits}(e)-1}$, which equals $t - c$ minus the time to process bit i, which equals d when e_i equals 1, or 0 otherwise.

By modeling these timings as random variables having a normal distribution, the attacker can — after determining the mean value $\mu(X)$ and the standard deviation $\sigma(X)$ based on information on the inner workings of the implementation or statistical analysis of timing measurements — compute the probabilities

$$P(b, t, c, d | e_i = 1) \approx \varphi\left(\frac{(t - c - d) - \mu(t - c - d)}{\sigma(t - c - d)}\right) \quad (3)$$

and

$$P(b, t, c, d | e_i = 0) \approx \varphi\left(\frac{(t - c) - \mu(t - c)}{\sigma(t - c)}\right), \quad (4)$$

where $\varphi(x)$ is the density function of the standard normal distribution:

$$\varphi(x) = \frac{e^{-\frac{1}{2}x^2}}{\sqrt{2\pi}}. \quad (5)$$

Using Bayes' theorem, equations 3 and 4 can be combined, such that the attacker can determine the probability that e_i is 1, given a known base value b and timing values t, c and d:

$$P(e_i = 1 | b, t, c, d) = \frac{P(b, t, c, d | e_i = 1)}{\sum_{v \in \{0,1\}} P(b, t, c, d | e_i = v)}. \quad (6)$$

After performing k measurements, the attacker can combine the results to determine the probability that bit e_i has value 1 using equation 7. By repeatedly obtaining exponent bits e_0 up to $e_{\text{NumberOfBits}(e)-1}$, the attacker is able to reconstruct the entire secret RSA exponent.

$$P(e_i = 1) \approx \frac{\prod_{i=0}^{k-1} P(e_i = 1|y_i, t_i, c_i, d_i)}{\sum_{v\in\{0,1\}} \prod_{i=0}^{k-1} P(e_i = v|y_i, t_i, c_i, d_i)} \quad (7)$$

3.2 Montgomery reduction

The running time of the REDC algorithm introduced in section 2.3 depends on the input value c', such that when it causes the expression $t' \geq N$ to be true, an extra reduction (line 4) is computed. Equation 7 in [4] states that the probability $\mathrm{pr_i}(u)$ that the extra reduction has to be performed for a modular multiplication of $u \in Z_n$ with a random variable B, uniformly distributed on Z_p, is

$$\mathrm{pr_i}(u) = \frac{u \bmod p}{2R}. \quad (8)$$

Therefore the expected number of extra reductions to be performed when directly computing the exponentiation $r \equiv b^e \pmod N$ is linear to the value of b, modulo N. When the Chinese Remainder Theorem (see section 2.2) is used to split the computation into $r_p \equiv b^e \pmod p$ and $r_q \equiv b^e \pmod q$ — with p and q being the two prime factors of N — hundreds of Montgomery multiplications will be carried out with factors $u \pmod p$ and $u \pmod q$. While the expected number of extra reductions increase when the value of b increases, there will be a sudden drop when the value of b jumps over any multiple of p or q.

In [4], Schindler explains that exactly this timing behaviour leaks an important piece of information: the exact value of a prime factor p of N. The attack that he describes falls in three phases. The first phase finds an "interval set" $\{u_1 + 1, ..., u_2\}$, with $0 < u_1 < u_2 < N$, that contains an integer multiple of either p or q. When such an interval is found, the second phase narrows down the interval so that it still contains the integer multiple of the prime factor, but gets small enough for the third phase to calculate $gcd(u, N)$ for all values of u in the interval. For most values of u, the greatest common divider with N will be 1, but one particular value of u will reveal a prime factor of N.

As soon as one prime factor is found, the other can be calculated by dividing N by the just found factor, so both the values p and q are revealed. All that is left at this point, is computing the RSA secret exponent d — which is the the modular multiplicative inverse of the public exponent e — using the extended version of Euler's algorithm.

3.3 Multiplication in OpenSSL

As described by Brumley and Boneh in [1], OpenSSL implements two multi-precision multiplication algorithms, the normal version of $O(n^2)$ and Karatsuba multiplication (see section 2.4) with a complexity of $O(n^{\log 3})$. When two numbers m and n with the same number of words are multiplied, the Karatsuba algorithm is used. The normal multiplication algorithm is used otherwise.

Since Karatsuba multiplication is typically faster than normal multiplication, time measurements can reveal which multiplication algorithm was used and therefore whether the two numbers are of the same word length. Interestingly, when computing $r \equiv b^e \pmod p$, this means that the Karatsuba multiplication is used much more often when the base b is just below p. When it is just above p, then $b \pmod p$ is small and the slower algorithm is used much more.

Brumley and Boneh use this fact in [1] to reveal one of the prime factors of the RSA modulus of a secure webserver connected through a network. They rely on the fact that OpenSSL uses the CRT (see section 2.2) to implement the RSA private key operation. The attack recovers one of the prime factors, say p, one bit at a time.

When the i most significant bits of p are known, $p_0...p_{i-1}$, define g as the number that has bits $g_0...g_{i-1}$ equals the known bits of p and all lower significant bits set to zero. Now, define g' as the number that is equal to g, except that $g'_i = 1$. Notice how $g < g' < p$ when bit p_i is 1 and $g < p < g'$ when bit p_i is 0. The result is that when bit p_i is 1, OpenSSL will mostly use the same multiplication routines in computing $g^e \pmod p$ and $g'^e \pmod p$, but when bit p_i is 0, there will be a significant timing difference in computing these two values.

4 Countermeasures

Timing attacks are an attack on the implementation of a certain algorithm, not on the algorithm itself. To prevent timing characteristics to leak information about the secret components — like the secret RSA exponent or the factorization of the modulus N into prime factors p and q — modifications must be made to the implementation. In the case of RSA, there are two common methods to prevent secret information to be recovered using timing attacks: making sure that the implementation always takes the same amount of time or implement RSA blinding to decouple the runtime of a single private key operation from the input parameters.

4.1 Equal timing

An intuitive way to prevent differences of the running time of a private key operation to leak information about the secret values involved in the computation is to make sure that every private key operation takes the same amount of time.

As an example, the binary right-to-left algorithm to compute $r \equiv b^e \pmod N$, introduced in section 2.1 can be changed to the following algorithm:

```
1:  r ← 1
2:  y ← b
3:  for i = 0 to NumberOfBits(e) − 1 do
4:     r' ← (ry) mod N
5:     if Bit_i(e) = 1 then
6:        r ← r'
7:     else
8:        r ← r
9:     end if
10:    y ← y^2 mod N
11: end for
12: return r
```

Notice how the value $(ry) \bmod N$ is computed for every bit (line 4), regardless of the value, so that it always takes the same amount of time to compute the value r.

However, since the time to compute a modular reduction (like those performed on lines 4 and 10) depends on the value of N, this example is still faulty and it immediately demonstrates an important disadvantage of equal timing: it is very difficult to implement correctly. But also if programmed carefully, modern compilers tend to optimize the generate code; a modern compiler would optimize this version in such a way, that it will actually reduce to the original version from section 2.1.

Another disadvantage is that running this version of the algorithm not only takes the same amount of time — regardless of the input values — it always takes the maximum amount of time.

4.2 RSA blinding

Another method to prevent timing differences to leak secret information is to decouple the runtime of an algorithm from the input values. Actually, in chapter 6 of [1] this method is presented as the preferred method. Before computing the exponentiation in the RSA private key operation, a transformation can be perform on the ciphertext, such that the actually computed value depends on a random variable and therefore the runtime of the algorithm has no relation to the ciphertext anymore.

Let r be a random value, such that $0 < r < N$. Notice how $(r^e C)^d r^{-1} \equiv r^{ed-1} C^d \pmod N$. Since $N = pq$, $r^{ed-1} \equiv 1 \pmod N$ by Eulers' theorem [7]. Comparing this equation to the original RSA secret key operation, the same value is computed, but a random value that decouples the runtime of the algorithm from the input values is introduced.

5 Conclusion

The RSA private key operation consist of computing a single modular exponentiation operation. Several algorithms for implementing this operation are available, yet the runtime of these algorithms often depends on the input values. When an attacker is able to supply the input value and measure the time it takes for the target implementation to perform the secret key operation, the attacker can reveal the secret RSA exponent or the factorization of N into prime factors p and q.

Several methods are available to prevent secret information to leak through timing differences. The preferred method is using RSA blinding, which is actually successfully implemented in cryptographic libraries like OpenSSL.

References

[1] David Brumley and Dan Boneh. Remote timing attacks are practical. In *In Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.

[2] Paul C. Kocher. Cryptanalysis of diffie-hellman, rsa, dss, and other cryptosystems using timing attacks. In *Advances in cryptology, CRYPTO 95: 15th Annual International Cryptology Conference*, pages 171–183. Springer-Verlag, 1995.

[3] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[4] Werner Schindler. A timing attack against rsa with the chinese remainder theorem. In *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 109–124. Springer-Verlag, 2000.

[5] Eric Weisstein. Karatsuba multiplication. MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/KaratsubaMultiplication.html`.

[6] Wikipedia. Chinese remainder theorem. `http://en.wikipedia.org/wiki/Chinese_Remainder_Theorem`.

[7] Wikipedia. Eulers' theorem. `http://en.wikipedia.org/wiki/Euler's_theorem`.