# Timing Attacks on Software Implementation of RSA

Project Report

Harshman Singh
School of Electrical Engineering and Computer Science
Oregon State University

Major Professor: Dr. Çetin Kaya Koç

# Acknowledgements

I am deeply grateful to my advisor Dr. Çetin Kaya Koç for his support, guidance and patience throughout my project work. I would like to thank Dr. Werner Schindler who answered my questions and provided me with lot of inputs. Special thanks to David Brumley for clarifying my doubts.

## Abstract

Timing attacks enable an attacker to extract secret information from a cryptosystem by observing timing differences with respect to different inputs given to an encryption or decryption algorithm. Werner Schindler has proposed a timing attack on smart card devices. We implemented this attack based on the same approach for RSA implementation provided by OpenSSL library. The attacking client can obtain private key information demonstrating the vulnerability of even software implementations of RSA to these attacks.

# Table of Contents

# List of Figures

# 1 Introduction

A timing attack is somewhat analogous to a burglar guessing the combination of a safe by observing how long it takes for someone to turn the dial from number to number. Timing attacks exploit the fact that, when the running time of a cryptographic algorithm is non-constant, then it may leak information about the secret parameters the algorithm is handling. Timing attack can expose private information, such as RSA keys, by measuring the amount of time required to perform private key operations e.g. encryption or decryption etc.

Timing attacks are related to a class of attacks called side-channel attacks. Others include power analysis and attacks based on electromagnetic radiation. Unlike these extended side channel attacks, the timing attack does not require special equipment and physical access to the machine. Here we only focus on the timing attacks that target the implementation of RSA decryption in OpenSSL.

Until now, timing attacks were only applied in the context of hardware security tokens such as smartcards. It is generally believed that timing attacks cannot be used in complex environment like networks or to attack general purpose servers, such as web servers, since decryption times are masked by many concurrent processes running on the system. It is also believed that common implementations of RSA (using Chinese Remainder Theorem and Montgomery multiplication) are not vulnerable to timing attacks. These assumptions are challenged by developing a remote timing attack against OpenSSL [1], an SSL library commonly used in web servers and other SSL applications.

Timing attacks are generally performed to attack devices such as smart cards which are weak computing devices. Werner Schindler has shown that timing attack against weak

computing devices such as smart cards could be implemented efficiently [4]. Smart cards traditionally contain embedded microchips which include some tamper resistantance and attack countermeasures. But due to their weak processing capacity, they can reveal important information about secret keys when subjected to timing attacks. We have picked the same approach to perform timing attacks against general software systems. The attack presented here is based on the implementation of RSA decryption in OpenSSL version 0.9.7. The basic attack works as follows: the attacking client measures the time an OpenSSL server takes to respond to decryption queries. The client is then able to extract the private key stored on the server.

The idea of timing attack was first introduced by Kocher [5] in 1996. Schindler [7] introduced timing attack against RSA decryption based in Chinese Remainder Theorem. Boneh and Brumley [3] has implemented the same attack on software systems. Current attack is an extension of Schindler's approach towards OpenSSL's implementation of RSA decryption.

# 2 RSA Cryptosystem

Traditional cryptography is based on the sender and receiver of a message using the same secret key. The sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. This method is known as secret-key or symmetric cryptography. The main problem is getting the sender and receiver to agree on the secret key without anyone else finding out.

The concept of *public-key cryptography* was introduced in 1976 by Whitfield Diffie and Martin Hellman in order to solve the key management problem. In their concept, each person gets a pair of keys, one called the *public key* and the other called the *private key*. Each person's public key is published while the private key is kept secret. Thus the need for the sender and receiver to share secret information is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. No longer is it

necessary to trust some communications channel to be secure against eavesdropping or betrayal. An algorithm using a pair of different, though related cryptographic keys to encrypt and decrypt is called and *asymmetric key algorithm*.

## 2.1 The RSA Algorithm

RSA is a one such asymmetric algorithm that is most widely used in public key cryptography. The RSA Algorithm was named after Ronald Rivest, Adi Shamir and Leonard Adelman, who first published the algorithm in April, 1977. Since that time, the algorithm has been employed in Internet electronic communications encryption program namely, PGP, Netscape Navigator and Microsoft Explorer web browsers in their implementations of the Secure Sockets Layer (SSL), and by MasterCard and VISA in the Secure Electronic Transactions (SET) protocol for credit card transactions.

The security of the RSA system is based on the intractability of the integer factorization problem. It is very quick to generate large prime numbers using probabilistic algorithms and Rabin-Miller test but very hard to factor large numbers. The next section describes RSA algorithm in more detail.

***Key Generation Algorithm***

1. Generate two distinct large random primes, $p$ and $q$, of approximately equal size such that their product $n = pq$ is of the required bit length, e.g. 1024 bits.
2. Compute $n = pq$ and Euler's totient function $\varphi(n) = (p-1)(q-1)$.
3. Choose an integer $e$, $1 < e < \varphi(n)$, such that $\gcd(e, \varphi(n)) = 1$.
4. Compute the secret exponent $d$, $1 < d < \varphi(n)$, such that $ed \equiv 1 \bmod \varphi(n)$.
5. The *public key* is $(n, e)$ and the *private key* is $(n, d)$. The values of $p$, $q$, and $\varphi(n)$ should be kept secret.

In practice, common choices for $e$ are 3, 17 and 65537 ($2^{16}+1$). These are *Fermat primes* and are chosen because they make the modular exponentiation operation faster. Value for

$d$ is calculated using the *Extended Euclidean Algorithm* $d = e^{-1} \bmod \varphi(n)$ which is also known as *modular inversion*.

Here $n$ is known as the *modulus*, $e$ is known as the *public exponent* and $d$ is known as the *secret exponent*.

### *Encryption*

Sender A does the following:

- Obtains the recipient B's public key ($n$, $e$).
- Represents the plaintext message as a positive integer $m < n$.
- Computes the ciphertext $c = m^e \bmod n$.
- Sends the ciphertext $c$ to B.

### *Decryption*

Recipient B does the following:

- Uses his private key ($n$, $d$) to compute $m = c^d \bmod n$.
- Extracts the plaintext from the integer representative $m$.

**RSA Algorithm**

- $n = pq$ where p and q are distinct primes.
- $\varphi(n) = (p-1)(q-1)$
- $e < n$ such that $\gcd(e, \varphi(n)) = 1$
- $d = e^{-1} \bmod \varphi(n)$
- $C = M^e \pmod n$
- $M = C^d \pmod n$

Fig 2.1 Summary of RSA Algorithm

# 3 Fast Implementation of RSA

## 3.1 Modular Exponentiation

Once an RSA cryptosystem is set up, i.e. the modulus $n$, the private exponent $d$ and public exponent are determined and the public components have been published, the senders as well as the recipients perform a single operation for encryption and decryption. The RSA algorithm in this respect is one of the simplest cryptosystems. In RSA encryption or decryption, the core part of the algorithm which takes up much time is the modular exponentiation. Especially in decryption we need to calculate, $M = C^d \pmod n$ and since $d$ is generally a big number (usually more than 512 bits). For the decryption to run acceptably, speeding up of modular exponentiation is very important. This section describes various methods that are generally employed to speed up the exponentiation process. Later sections describe how these very mechanisms are exploited to retrieve RSA key private information using timing attacks on decryption process.

## 3.2 Chinese Remainder Theorem

To speed up the modular exponentiation during decryption RSA uses Chinese Remainder Theorem. In essence, the CRT says it is possible to reconstruct integers in a certain range from their residues modulo a set of pairwise relatively prime moduli. Formally,

**Theorem 1 (CRT)** Let $m_1, m_2, \ldots, m_k$ be pairwise relatively prime integers, that is $\gcd(m_i, m_j) = 1$ whenever $i \neq j$. Then, for any $a_1, a_2, \ldots, a_k \in \mathbb{Z}$, the system of equations

$$\begin{aligned} x &= a_1 \bmod m_1 \\ x &= a_2 \bmod m_2 \\ &\ldots \\ x &= a_k \bmod m_k \end{aligned}$$

has a solution, and the solution is unique modulo $m = m_1 m_2 \ldots m_k$. That is if $x$ is a solution, then any $x' = x \bmod m$ is also a solution.

For example, let $m$, $n$ be two integers and gcd($m$,$n$) = 1. Now given $0 \leq a < m$ and $0 \leq b < n$, there exists a unique integer $X$, $0 \leq X < mn$ such that,

$$X \equiv a \pmod{m}$$
$$X \equiv b \pmod{n}$$

and $X$ is given by,

$$X \equiv ac_1 m + bc_2 n \pmod{mn}$$

where,

$$c_1 = n \bmod m$$
$$c_2 = m \bmod n$$

or $X$ can also be calculated using,

$$X = a + m[(b-a)c_2 \bmod n]$$

RSA decryption calculates $M = C^d \bmod n$, where $n = pq$ is the RSA modulus. With Chinese remaindering this decryption process could be made faster by computing it in in two steps instead. Write the decryption as $M = C^d \pmod{pq}$. This can then be broken into two congruences,

$$M_1 \equiv C^d \pmod{p}$$
$$M_2 \equiv C^d \pmod{q}$$

Now by applying Fermat's theorem, the above equations could be written as,

$$M_1 \equiv C^{d_1} \pmod{p}$$
$$M_1 \equiv C^{d_2} \pmod{p}$$

Evaluate those two where $d_1$ and $d_2$ are precomputed from $d$,

$$d_1 \equiv d \pmod{p\text{-}1}$$
$$d_2 \equiv d \pmod{q\text{-}1}$$

Then $m_1$ and $m_2$ are combined using CRT to yield $M$,

$$M = M_1 + p[(M_2 - M_1)P^{-1} \bmod q$$

In the above method $d_1$ and $d_2$ are almost half the size of $d$ and $p$ and $q$ are almost half the size of $n$. This is faster by the factor of four than the original calculation to perform the exponentiation.

### 3.2.1 Exploiting CRT in Timing Attack

RSA decryption with CRT generally gives a speed up of four times. Although RSA with CRT is not vulnerable to Kocher's timing attack [5]. But an RSA implementation with CRT uses factors of public modulus *n*. These factors could be exposed by performing a timing attack on the decryption process. If factors of *n* become known then the decryption key could be calculated by computing $d = e^{-1} \bmod (p-1)(q-1)$, thus effectively breaking the cipher.

## 3.3 Modular Exponentiation Speed-up Techniques

RSA decryption with CRT needs to calculate $C^{d_1} \pmod{p}$ and $C^{d_2} \pmod{p}$ . These modular exponentiations are computationally expensive operations. Various techniques exist for speeding up this most crucial operation in RSA decryption. The simplest algorithm for computing modular exponentiation is square and multiply, also called binary method. Next sections explain this algorithm and various techniques employed in fast implementation of RSA.

### 3.3.1 Square and Multiply Exponentiation

A naïve way of computing $C = M^e \pmod{n}$ is to start with $C := M \pmod{n}$ and keep performing the modular multiplication operations,

$$C := C \cdot M \pmod{n}$$

until $C = M^e \pmod{n}$ is obtained. The naïve method requires $e - 1$ modular multiplications to compute $C := M^e \pmod{n}$, which would be prohibitive for large *e*.

The square and multiply method is designed to do modular exponentiation more quickly. It runs in arounf $O(k^3)$ operations if *k* is the number of bits in the exponent. It is based on two facts that exponentiation could be done faster as a series of squaring operations and occasional multiplications, when the exponent is odd. And while using binary arithmetic

all the multiplications are not necessary, zeros can be skipped. The square-and-multiply or binary method is described below:

The square-and-multiply method scans the bits of the exponent either from left to right. A squaring is performed at each step, and depending on the scanned bit value, subsequent multiplication is performed. Let $k$ be the number of bits of $e$, i.e. $k = 1 + \lfloor \log_2 e \rfloor$, and the binary expansion of $e$ be given by,

$$e = (e_{k-1} e_{k-2} \ldots e_1 e_0) = \sum_{i=0}^{k-1} e_i 2^i$$

for $e_i \in \{0,1\}$. The binary method for computing $C = M^e \pmod n$ is given below**:**

**The square-and-multiply method**
*Input: M, e, n.*
*Output:* $C = M^e \bmod n$.
**1.** **if** $e_{k-1} = 1$ **then** $C$ **:=** $M$ else $C$ **:=**1
**2.** **for** $i = k$-2 **downto** 0
**2a.** $C$ **:=** $C \cdot C \pmod n$
**2b. if** $e_i = 1$ **then** $C$ **:=** $C \cdot M \pmod n$
**3.** **return** $C$

Fig. 3.1 The square-and-multiply algorithm

## 3.3.2 Sliding Window Technique

Simple square-and-multiply processes only one bit of $e$ at each iteration in $C = M^e \pmod n$. To further optimize the performance of square-and-multiply sliding window technique is used. When using sliding windows a block of bits, called *window*, of $e$ are processed at each iteration. The speed of square-and-multiply is improved at the cost of some memory which is required to store certain precomputed values. Sliding windows requires pre-

computing a multiplication table, which takes time proportional to $2^{w-1}+1$ for a window of size $w$.

The sliding window technique divides the given number into zero and non-zero words of variable length. To calculate $C = M^e \pmod n$, M is divided into zero and non-zero windows of variable length. Let $w$ be the *window size*.

**Algorithm** Sliding-window exponentioation

INPUT: $g$, $e = (e_t e_{t-1} \ldots e_1 e_0)_2$ with $e_t = 1$, and $w \geq 1$.
OUTPUT: $g^e$

**1.** *Precomputation*.
  a. $g_1 := g$, $g_2 := g^2$
  b. **for** $i$ **from** 1 **to** $(2^{w-1} - 1)$ **do**
     $g_{2i+1} := g_{2i-1} \cdot g_2$
**2.** $A := 1$, $i := t$
**3.** **while** $i \geq 0$ **do**
  a. **if** $e_i = 0$ **then do**
     $A := A^2$
     $i := i - 1$
  b. **otherwise** ($e_i \neq 0$), find the longest bitstring $e_i e_{i-1} \ldots e_v$ such that $i\text{-}v+1 \leq w$ and $e_v = 1$, and **do**
     $A := A^{2^{i-v+1}} \cdot g_{(e_i e_{i-1} \ldots e_v)_2}$
     $i := v - 1$
**4.** return($A$)

Fig. 3.2 Exponentiation using sliding window technique

There is an optimal window size that balances the time spent during precomputation versus actual exponentiation. For a 1024-bit modulus, OpenSSL uses a window size of five so that about five bits of the exponent $e$ are processed in every iteration.

## Exploiting Sliding Window in the New Timing Attack

For the timing attack the point to mark is that during the sliding windows algorithm there are many multiplications by $g$, where $g$ is the input ciphertext. By querying on many

inputs $g$ the attacker can expose information about bits of the factor $q$. But timing attack on sliding windows is trickier since there are far fewer multiplications by $g$ in sliding windows. Special techniques has to be employed to handle sliding windows exponentiation used in RSA implementations utilizing it.

## 3.4 Modular Multiplication and Modular Reduction

The sliding windows exponentiation algorithm performs a modular multiplication at every step. Given two integers $x$, $y$ computing $xy$ mod $q$ is done by first multiplying the integers $x \cdot y$ and then reducing the result modulo $q$.

A naïve method to perform a reduction modulo $q$ is done via multi-precision division and returning the remainder. This is a very expensive operation. Montgomery [7] has proposed a method for efficiently implementing a reduction modulo $q$ using a series of operation in hardware and software.

### 3.4.1 Montgomery Multiplication

In 1985, P. Montgomery introduced an efficient algorithm for computing $u = a \cdot b \pmod{n}$ where $a$, $b$, and $n$ are $k$-bit binary numbers. The algorithm is particularly suitable for implementation on general-purpose computers namely, signal processors or microprocessors) which are capable of performing fast arithmetic modulo a power of 2. The Montgomery reduction algorithm computes the resulting $k$-bit number $u$ without performing a division by the modulus $n$. Via an ingenious representation of the residue class modulo $n$, this algorithm replaces division by $n$ with division by a power of 2. The latter operation is easily accomplished on a computer since the numbers are represented in binary form. Assuming the modulus $n$ is a $k$-bit number, i.e.$2^{k-1} \leq n < 2^k$, let $r$ be $2^k$. The Montgomery reduction algorithm requires that $r$ and $n$ be relatively prime, i.e.,

$$\gcd(r, n) = \gcd(2^k, n) = 1$$

This requirement is satisfied if $n$ is odd. In the following, we summarize the basic idea behind the Montgomery reduction algorithm.

Given an integer $a < n$, define its *n-residue* or *Montgomery representation* with respect to $r$ as,

$$\overline{a} = a \cdot r \pmod{n}$$

Clearly, the sum or difference of the Montgomery representations of two numbers is the Montgomery representation of their sum or difference respectively:

$$\overline{a} + \overline{b} = \overline{(a+b)} \pmod{n} \text{ and } \overline{a} - \overline{b} = \overline{(a-b)} \pmod{n}$$

It is straightforward to show that the set

$$\{\, i \cdot r \pmod{n} \mid 0 \le i \le n\text{-}1 \,\}$$

is a complete residue system, i.e. it contains all numbers between 0 and $n$-1. Thus, there is a one-to-one correspondence between the numbers in the range 0 and $n$-1 and the numbers in the above set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the $n$-residue of the product of the two integers whose n-residues are given. Given two $n$-residues $\overline{a}$ and $\overline{b}$, the *Montgomery product* is defined as the scaled product,

$$\overline{u} = \overline{a} \cdot \overline{b} \cdot r^{-1} \pmod{n}$$

where $r^{-1}$ is the (multiplicative) inverse of $r$ modulo $n$ i.e. it is the number with the property,

$$r^{-1} \cdot r = 1 \pmod{n}.$$

As the notation implies, the resulting number *u'* is indeed the $n$-residue of the product,

$$u = a \cdot b \pmod{n}$$

since,

$$\begin{aligned} \overline{u} &= \overline{a} \cdot \overline{b} \cdot r^{-1} \pmod{n} \\ &= (a \cdot r) \cdot (b \cdot r) \cdot r^{-1} \pmod{n} \\ &= (a \cdot b) \cdot r \pmod{n}. \end{aligned}$$

In order to describe the Montgomery reduction algorithm, we need an additional quantity, η, which is the integer with the property,

$$r \cdot r^{-1} - n \cdot n' = 1$$

The integers $r^{-1}$ and $n'$ can both be computed by the extended Euclidean algorithm. The Montgomery product algorithm which computes,

$$\overline{u} = \overline{a} \cdot \overline{b} \cdot r^{-1} \pmod{n}$$

given $\overline{a}$ and $\overline{b}$ is given below**:**

**Algorithm** MonPro($\overline{a}$, $\overline{b}$)

1. $t := \overline{a} \cdot \overline{b}$
2. $m := t \cdot n' \pmod{r}$
3. $\overline{u} := (t + m \cdot n)/r$
4. **if** $\overline{u} \geq n$ **then return** $\overline{u} - n$
   **else return** $\overline{u}$

Fig. 3.3 Montgomery Multiplication Algorithm

The most important feature of the Montgomery product algorithm is that the operations involved are multiplications modulo *r* and divisions by *r*, both of which are intrinsically fast operations since *r* is a power 2. The MonPro algorithm can be used to compute the "normal" product *u* of *a* and *b* modulo *n* provided that *n* is odd:

**Function** ModMul(*a*, *b*, *n*) { *n* is odd}

1. *Compute n' using the extended Euclidean algorithm.*
2. $\overline{a}$ := $a \cdot r \pmod{n}$
3. $\overline{b}$ := $b \cdot r \pmod{n}$
4. $\overline{u}$ := ModPro($\overline{a}$, $\overline{b}$)
5. $u$ := ModPro($\overline{u}$, 1)
6. **return** *u*

## 3.4.2 Montgomery Exponentiation

The Montgomery product algorithm is more suitable when several modular multiplications are needed with respect to the same modulus. Such is the case when one needs to compute a modular exponentiation, i.e., the computation of $M^e \pmod{n}$. Algorithms for modular exponentiation decompose the operation into a sequence of squarings and multiplications using a common modulus $n$. This is where the Montgomery product operation MonPro finds its best use. In the following, we exemplify modular exponentiation using the standard "square-and-multiply" method, i.e. the left-to-right binary exponentiation method, with $e_i$ being the bit of index $i$ in the $k$-bit exponent $e$. Presented ahead in the section is the algorithm that uses Montgomery exponentiation to calculate $M^e \pmod{n}$:

**Algorithm** MonExp($M$, $e$, $n$)
1. *Compute n' using the extended Euclidean algorithm.*
2. $\overline{M}$ **:=** $M \cdot r \pmod{n}$
3. $\overline{x}$ **:=** $1 \cdot r \pmod{n}$
4. for i = k – 1 down to 0 do
5. $\overline{x}$ **:=** MonPro($\overline{x}$ , $\overline{x}$ )
6. **if** ei = 1 **then** $\overline{x}$ **:=** MonPro($\overline{M}$ , $\overline{x}$ )
7. $x$ **:=** MonPro($\overline{x}$ , 1)
8. **return** $x$

Fig. 3.4 Montgomery Exponentiation Algorithm

Thus, we start with the ordinary residue $M$ and obtain its $n$-residue $\overline{M}$ and the $n$-residue $\overline{1}$ of 1 using division-like operations, as was given above. After this pre-processing is done, the inner loop of the binary exponentiation method uses the Montgomery product operation, which performs only multiplications modulo $2^k$ and divisions by $2^k$. When the loop terminates, the $n$-residue $\overline{x}$ of the quantity $x = M^e \pmod{n}$ is obtained. The ordinary residue number $x$ is obtained from the $n$-residue by executing the MonPro function with arguments $\overline{x}$ and 1. This is readily shown to be correct since

$$\bar{x} = x \cdot r \pmod{n}$$

implies that,

$$x = \bar{x} \cdot r^{-1} \pmod{n} = \bar{x} \cdot 1 \cdot r^{-1} \pmod{n} := \text{MonPro}(\bar{x}, 1).$$

The resulting algorithm is quite fast, as is demonstrated by its various implementations.

### 3.4.3 Montgomery Exponentiation and Sliding Window Technique

To achieve even faster exponentiation Montgomery multiplication can be combined with sliding window technique. To calculate $M^e$ mod $n$ we proceed the following way:

**Algorithm** Montgomery exponentiation with sliding-windows

INPUT: $M$, $e$, $n$
OUTPUT: $M^e \pmod{n}$

1. $\bar{M}$ := $M \cdot r \bmod n$
2. $\bar{x}$ := $1 \cdot r \bmod n$
3. partition $e$ into zero and no-zero windows, $w_i$ of length $l_i$ where $i = 1,2,3,\ldots,k$
4. precompute and store $\bar{M}^j \pmod{n}$ where $j = 3,5,\ldots 2^m-1$
5. **for** $i = k$ **downto** 1 **do**
6. $\bar{x} = \text{MonPro}(\bar{x}^{2^{l_i}})$
7. **if** $w_i \neq 0$ then $\bar{x} = \text{MonPro}(\bar{x}, \bar{M}^{w_i})$
8. $x = \text{MonPro}(1, \bar{x})$
9. return $x$

Fig 3.5 Montgomery Exponentiation Algorithm (with sliding windows)

## 3.5 Multiplication in RSA operations

Multi-precision integer multiplication routines lie at the heart of most of the essential RSA operations and the speed up techniques. OpenSSL implements two multiplication routines: Karatsuba and normal brute force multiplication. The simplest of them is the classical method for multiplying large numbers as described below:

Assume that $u$ and $v$ are two $m$ bit and $n$ bit numbers, respectively, and $t$ is their product, which would be a $mn$ bit number. The calculation of $t$ could be done as follows:

**Algorithm** Brute force multiplication of large numbers

INPUT: positive integers $u = (u_{m-1}\ u_{m-2}\ \ldots\ u_1\ u_0)$ and $v = (v_{n-1}\ v_{n-2}\ \ldots\ v_1\ v_0)$
OUTPUT: the integer product $t = u \cdot v$

1. **for** $i = 0$ **to** $m + n + 1$ **do**
2. $t_i \leftarrow 0$
3. $c \leftarrow 0$
4. **for** $i = 0$ **to** $m$ **do**
5. **for** $j = 0$ **to** $n$ **do**
6. $(c, s)_B \leftarrow t_{i+j} + u_i \cdot v_j + c$
7. $t_{i+j} = s$
8. $t_{i+n+1} = c$
9. **return** $(t)$

Fig 3.6 Classical Multi-precision Multiplication Algorithm

The algorithm proceeds in a row-wise manner. That is, it takes one digit of one operand and multiplies it with the digits of the other operand, in turn appropriately shifting and accumulating the product. The two digit intermediary result $(c,s)_B$ holds the value of the digit $t_{i+j}$ and the carry digit that will be propagated to the digit $t_{i+j+1}$ in the next iteration of the inner loop. The subscript $B$ indicates that the digits are represented in radix-$B$. As can be easily seen the number of digit multiplications performed in the overall execution of the algorithm is $m \cdot n$. Hence, the time complexity of the classical brute force multiplication algorithm grows with $O(n^2)$, where $n$ denotes the size of the operands.

But this algorithm is too slow to be solely dependent upon. OpenSSL implementation of RSA employs Karatsuba's algorithm to speed up multiplication of large numbers.

### 3.5.1 Karatsuba Multiplication

Karatsuba and Ofman proposed a faster algorithm for multiplication in 1962. The splitting technique introduced by Karatsuba reduces the number of multiplication in exchange of extra additions. The algorithm works by splitting the two operands $u$ and $v$ into halves,

$$u = u_1B + u_0 \text{ and } v = v_1B + v_0$$

where $B = 2^{n/2}$ and $n$ denotes the length of the operands. First the following three multiplications are computed:

$$d_0 = u_0v_0$$
$$d_1 = (u_1 + u_0)(v_1 + v_0)$$
$$d_2 = u_1v_1$$

Note that these multiplications are performed with operands of half length. In comparison, the grammar school method would have required the computation of four multiplications. The product $u \cdot v$ is formed by appropriately assembling the three products together as follows,

$$u \cdot v = d_0 + (d_1 - d_0 - d_2)B + d_2B^2$$

In total three multiplications, two additions and two subtractions are needed compared to the four multiplications and one addition required by the classical multiplication method. Hence, the Karatsuba technique is preferable over the classical method whenever the cost of one multiplication exceeds the cost of three additions counting additions and subtraction as equal. The true power of the Karatsuba method is realized when recursively applied to all partial product computations in a multi-precision multiplication operation. For Karatsuba algorithm, the length of the operands must be powers of two. This technique could be used recursively to further speed up the process when dealing with very large numbers.

**Algorithm** Karastuba( $a$, $b$, $n$ )

1. Let $p = \lfloor \frac{n}{2} \rfloor$ and q = $\lceil \frac{n}{2} \rceil$ (such that $p + q = n$ ).
2. Write $u = u_0 + X^p a_1$ and $b = b_0 + X^p b_1$.
3. Compute $\alpha = a_0 - a_1$ and $\beta = b_0 - b_1$.
4. **return** Karastuba($a_0$, $b_0$, $p$) $\cdot$ (1+$X^p$) + Karastuba($\alpha$, $\beta$, $q$) $\cdot$ $X^p$
5. + Karastuba($a_1$, $b_1$, $q$) $\cdot$ ($X^p + X^{2p}$).

Fig 3.7 Karatsuba Algorithm

Essentially, Karastuba method drills down the multiplication down to the level that the actual multiplication takes place in the sizes of words depending on the platform being used. Karatsuba algorithm takes $O(n^{\log_2 3})$ time which is $O(n^{1.58})$.

## Exploiting multiplication optimizations in timing attacks

Multi-precision libraries represent large integers as a sequence of words. OpenSSL implements both classical and Karastuba routines for multi-precision integer multiplication.

OpenSSL uses Karastuba multiplication when multiplying two numbers with an equal number of words, which runs in $O(n^{\log_2 3})$ or $O(n^{1.58})$ time. OpenSSL uses normal multiplication when multiplying two numbers with an unequal number of words of size $n$ and $m$, which runs in $O(nm)$ time. Hence, for numbers that are approximately the same size, so that n is close to m, normal multiplication takes quadratic time. That leads to the inference that OpenSSL's integer multiplication routine leaks important timing information. Since Karastuba is typically faster, multiplication of two unequal size words takes longer thatn multiplication of two equal size words. Time measurements can reveal how often the operands given to the multiplication routine have the same length. This fact is used in the timing attack on OpenSSL.

# 4 OpenSSL's Implementation of RSA

OpenSSL is a widely deployed, open source implementation of the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. The SSL and TLS protocols are used to provide a secure connection between a client and a server for higher level protocols such as HTTP.

The OpenSSL *crypto* library implements a wide range of cryptographic algorithms used in various Internet standards. The services provided by this library are used by the OpenSSL implementations of SSL, TLS and S/MIME, and they have also been used to implement SSH, OpenPGP, and other cryptographic standards. This is the library that provides functionality required to use RSA public key cryptosystem. Following are the main functions used to implement RSA public key encryption and decryption:

```
#include <openssl/rsa.h>

int RSA_public_encrypt(int flen, unsigned char *from, unsigned char *to,
                  RSA *rsa, int padding);
int RSA_private_decrypt(int flen, unsigned char *from,
                  unsigned char *to, RSA *rsa, int padding);
int RSA_private_encrypt(int flen, unsigned char *from,
                  unsigned char *to, RSA *rsa,int padding);
int RSA_public_decrypt(int flen, unsigned char *from, unsigned char *to,
                  RSA *rsa,int padding);
```

OpenSSL maintains the following RSA data structure to store public keys as well as private RSA keys. The structure also stores several precomputed values, `dmp1`, `dmq1` and `iqmp` etc. which when available help RSA operations execute much faster.

```
struct
        {
        BIGNUM *n;              // public modulus
        BIGNUM *e;              // public exponent
        BIGNUM *d;              // private exponent
        BIGNUM *p;              // secret prime factor
        BIGNUM *q;              // secret prime factor
        BIGNUM *dmp1;           // d mod (p-1)
        BIGNUM *dmq1;           // d mod (q-1)
        BIGNUM *iqmp;           // q^-1 mod p
        // ...
```

```
        };
RSA
```

Implementation of RSA decryption in OpenSSL is the most relevant to the presented timing attack. Core of RSA decryption lies in modular exponentiation $M = C^d \bmod n$ where $n = pq$ is the RSA modulus, $d$ is the private exponent and $C$ the ciphertext. OpenSSL employs various speedup techniques which have been described in sections above. The exponentiation marked out here is performed using CRT. RSA decryption with CRT speeds up by a factor of four. Since CRT uses factors of public modulus $n$ hence they are vulnerable to be discovered by a timing attack. Once they are discovered it is easy to discover the private key by simple modular arithmetic.

Also for multi-precision multiplications required in CRT based exponentiations, a number of techniques are used. OpenSSL uses square-and-multiply with sliding windows exponentiation. Sliding windows perform a pre-computation to build a multiplication table. This precomputation takes $2^{w-1}+1$ for a window of size $w$. In case of a 1024-bit modulus OpenSSL employs window size of five.

From timing attack perspective, the most important is that in silding windows there are many multiplications by the ciphertext input to RSA decryption algorithm. By querying on many such inputs the information about factor $q$ could be exposed. But timing attack on sliding windows is much harder due to lesser number of multiplications being performed there. Special techniques have to be employed to overcome this.

OpenSSL uses Montgomery technique to perform multiplications and their modular reductions as necessary in sliding window exponentiations. Montgomery reductions transform a reduction modulo $q$ to a reduction modulo some power of two. This is faster since many arithmetic operations could be implemented directly in hardware in that case. So for a little penalty of transforming input to Montgomery form a large gain is achieved in modular reduction performance.

## 4.1 Extra Reductions

The most relevant fact about Montgomery technique is that at the end of multiplication there is a check to see if the output is greater than modulo reduction number.

**if** $\bar{u} \geq q$ **then return** $\bar{u}$ - $q$
**else return** $\bar{u}$

Here while calculating $C^d \pmod{q}$, $q$ is subtracted from the output to ensure that the output is in range $[0, q)$. This extra step is called an *extra reduction* and causes noticeable timining difference in RSA decryption for different ciphertext inputs. Werner Schindler [9] observed that the probability of an extra reduction happening during some eponetiation $g^d \bmod q$ is proportional to how close $g$ is to $q$. The probability for an extra reduction is given by:

$$\Pr[extra - reduction] = \frac{g \bmod q}{2R}$$

Where $R := h^2$ for some $h$ such that Montgomery multiplication was used to transform reduction modulo $q$ to reduction modulo $R$.

OpenSSL calls following function in the order whenever a decryption query is made.

```
int RSA_private_decrypt(int flen, unsigned char *from,
    unsigned char *to, RSA *rsa, int padding);
```

calls the following function,

```
int BN_mod_exp(BIGNUM *r, BIGNUM *a, const BIGNUM *p,
         const BIGNUM *m, BN_CTX *ctx);
```

which in turn calls the Montgomery multiplication function is the modulus is odd,

```
int BN_mod_mul_montgomery(BIGNUM *r, BIGNUM *a, BIGNUM *b,
         BN_MONT_CTX *mont, BN_CTX *ctx);
```

this function then uses the following code to convert back and forth the Montgomery form:

```
int BN_from_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
         BN_CTX *ctx);
```

The above mentioned extra reduction as appears in file *bn_mont.c*:

```
if (BN_ucmp(ret, &(mont->N)) >= 0){
      if (!BN_usub(ret,ret,&(mont->N))) goto err;
}
```

So if ciphertext input *g* approaches either factor *p* or *q* from below, the number of extra reductions greatly increases. But at exact multiple of *p* or *q* the number of extra reduction drops vastly. Figure 4.1 clearly marks the discontinuity in number of extra reductions appearing at multiples of *p* or *q*.

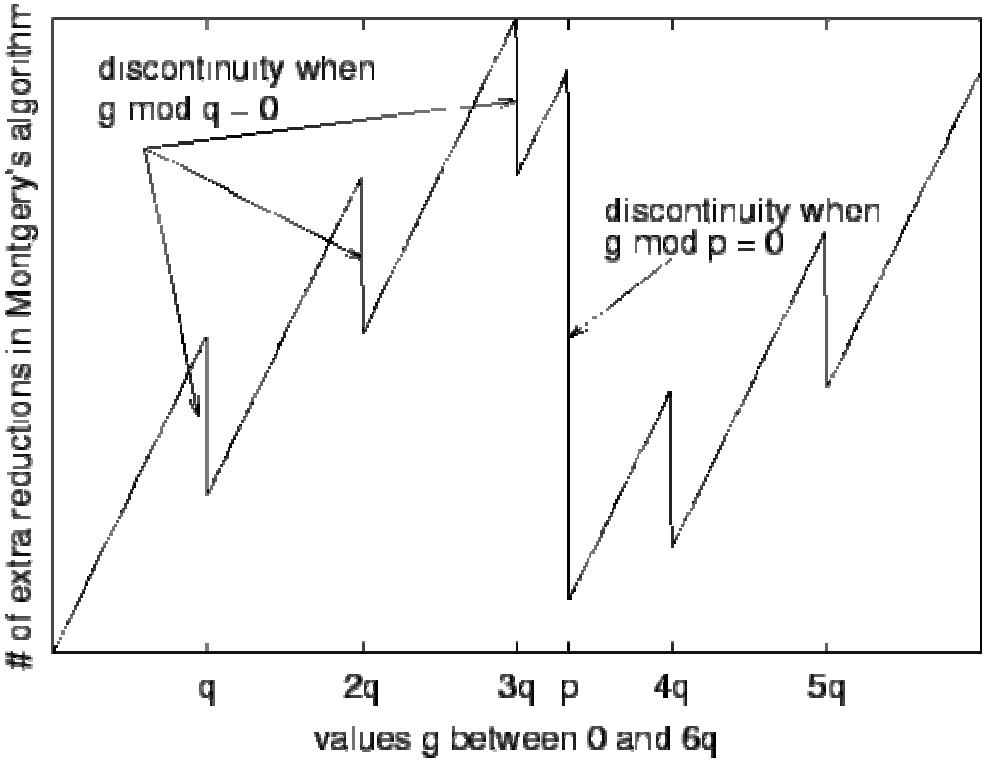


Fig 4.1 Extra Reductions in Montgomery Reduction

By detecting resulting timing differences in decryption it is possible to tell how close input *g* is to *p* or *q*.

As has been described in the sections above, OpenSSL utilizes both classical and Karatsuba methods for doing multi-precision integer multiplications. A choice about which method to use is based on the input values. Karastuba method is used whenever multiplication of two unequal size words is required, otherwise classical method is used. OpenSSL leaks important timing information here. Karastuba is typically faster so timing measurements could reveal how often the operands given to the multiplication routine have the same length. This fact is used in the timing attack.

# 5 The New Timing Attack on OpenSSL

Two factors cause the time variance in RSA decryption, namely, extra reductions in Montgomery multiplication and choice of Karastuba versus classical multiplication routine.

Consider decryption of a chosen ciphertext $g$ by OpenSSL. Begin with a smaller $g$ and let it approach towards a multiple of factor $q$ from below. The number of extra reductions keeps on increasing until it reaches some multiple of $q$. After that the number of extra reductions decreases dramatically. So decryption of $q < q$ is slower than decryption of $g > q$.

Karastuba versus classical multiplication causes opposite effect. When $g$ only slightly less than some multiple of $q$, Karatsuba is used. If $g$ is over the value of a multiple of $q$ then $g$ mod $q$ is a small number consequently OpenSSL is more likely to choose slower classical multiplication. In this case the decryption of g < q should be faster than decryption of g > q, which is the exact opposite of the effect of presence of extra reductions in Montgomery's algorithm.

## 5.1 Exploiting Sliding Window Pre-computations

Since OpenSSL uses CRT with sliding windows with window size 5 there is a significant amount of precomputation involved. If input is chosen carefully noticeable timing

differences could be observed in exponentiation calculations. Werner Schindler [2] made the following observation about implementation of sliding windows exponentiation in OpenSSL:

Let $y$ be the input for the exponentiation algorithm and $R$ the Montgomery constant, in present case $R = 2^{512}$. Further, let MonPro($a$, $b$) mean the Montgomery multiplication of $a$ and $b$, i.e.,

$$\text{MonPro}(a, b) := a \cdot b \cdot R^{-1} \pmod{q}.$$

If the exponentiation uses CRT with sliding window (window size = 5) then following values are precomputed and stored [10]:

$$\begin{aligned}
y_1 &:= \text{MonPro}(y, R^2 \pmod{q}) &&= y \cdot R \pmod{q},\\
y_2 &:= \text{MonPro}(y_1, y_1) &&= y^2 \cdot R \pmod{q},\\
y_3 &:= \text{MonPro}(y_1, y_2) &&= y^3 \cdot R \pmod{q},\\
y_5 &:= \text{MonPro}(y_3, y_2) &&= y^5 \cdot R \pmod{q},\\
&\cdots\\
y_{29} &:= \text{MonPro}(y_{27}, y_2) &&= y^{29} \cdot R \pmod{q},\\
y_{31} &:= \text{MonPro}(y_{29}, y_2) &&= y^{31} \cdot R \pmod{q}.
\end{aligned}$$

Now if the input to exponentiation is some number $y$ such that,

$$y = [h \cdot R^{-1/2} \pmod{n}] \pmod{n}$$

where $h \sim q^{1/2}$, then from the calculations above we have $y_2 = u$. To compute the values $y_3, y_5, \ldots, y_{31}$ the pre-computations process multiplies 15 times with $y_2$.

Whereas in Boneh and Brumley's timing attack [Ref BnB] the input is,

$$y = [g \cdot R^{-1} \pmod{n}] \pmod{n}$$

where $g \sim q$. In this case the multiplication with $y_1$ is exploited. In particular, $y_1 = g$. And one would expect that about 5 - 6 multiplications with $y_1$. The exact number of multiplications depends on $q$.

This difference in number of calculations directly relating to input, helps in observing the time variations more clearly.

## 5.2 Implementation Details of the Attack

The presented time attack is the chosen input attack. The basic idea is to make an initial guess and refine it by learning bits one at a time, from the most significant to the least. Let $n = pq$ with $q < p$. The attack proceeds by building approximations to $q$ that get progressively closer. Thus our attack can be viewed as a binary search for $q$. After having recovered half-most significant bits of $q$, the rest of them could be retrieved using Coppersmith's algorithm. We initially begin with the guess $g$ of $q$ lying between $2^{512}$ (or $2^{\log_2 N/2}$) and $2^{511}$ (or $2^{\log_2 (N/2)-1}$). We try all the combinations of the top few bits. Having timed the decryptions, we pick the first peak as the guess for $q$ (We at least know that the first bit is 1). The attack proceeds basically in the following steps:

Suppose we have already recovered the top $i$-1 bits of $q$. Let g be an integer that has the same top $i$-1 bits as $q$ and the remaining bits of $g$ are 0. Here $g < q$.

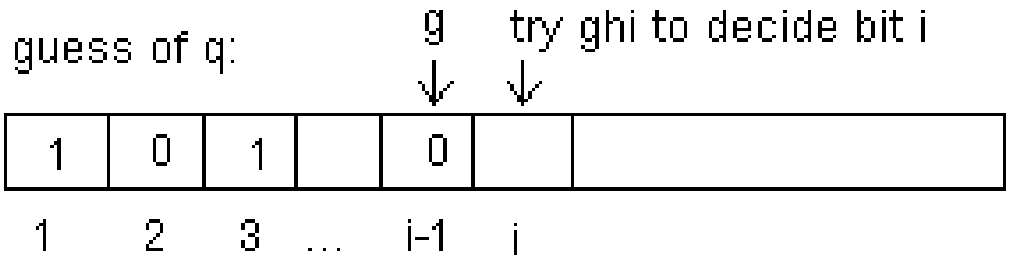


Fig 5.1 Bits of guess $g$ for factor $q$

We go with the following strategy to recover the next $i$th bit of $q$ and rest of higher bits:

1. Initialize $R$ **:=** $2^a$, whereas $a$ is the number of bits of the prime factors $p$ or $q$ which is typically 512.
2. After having guessed bit $i$-1 of the prime $q$ counting from the left and starting the count with $i = 1$, there exists two bounds for current guess as, $g_{i\text{-}1,\,\text{low}}$ and $g_{i\text{-}1,\text{high}}$ with,

$$g_{i\text{-}1,\text{high}} = g_{i\text{-}1,\text{low}} + 2^{a-i}$$

If all the guessing have been correct so far then $g_{i\text{-}1,\text{low}} < p < g_{i\text{-}1,\text{high}}$ should hold. Now let $g$ **:=** $g_{i\text{-}1,\text{low}}$ and $g_{hi}$ **:=** $g + 2^{a-i-1}$ i.e. $g_{hi}$ is of the same value as $g$, with only the $i$th bit set to 1.

**3.** Now if bit $i$ of $q$ is 1, then $g < g_{hi} < q$. Otherwise, $g < q < g_{hi}$.

**4.** Compute $h_{hi}$ **:=** $Round(g_{hi}^{1/2})$ and $h$ **:=** $Round(g^{1/2})$. Here *Round* function rounds the argument to the closest integer.

**5.** Compute $u_h = hR^{-1/2} \bmod n$ and $u_{hi} = h_{hi}R^{-1/2} \bmod n$. This step is needed because RSA decryption with Montgomery reduction and sliding windows will put $u_h$ and $u_{hi}$ in Montgomery form before exponentiation.

**6.** We measure the time to decrypt both $u_h$ and $u_{hi}$ . Let $t_1$ = DecryptTime( $u_h$ ) and $t_2$ = DecryptTime( $u_{hi}$ ).

**7.** We calculate the difference $\Delta = | t_1 - t_2 |$. If $g < q < g_{hi}$ then, the difference $\Delta$ will be "large", and bit $i$ of $q$ will be 0. If $g < g_{hi} < q$, the difference $\Delta$ will be "small", and bit $i$ of $q$ will be 1. We use previous $\Delta$ values to know what to consider "large" and "small". Thus we can use the value $| t_1 - t_2 |$ as an indicator for the $i$th bit of $q$.

When the $i$th bit is 0, the "large" difference can either be negative or positive. In this case, if $t_1 - t_2$ is positive then DecryptTime( $g$ ) > DecryptTime( $g_{hi}$ ), and the Montgomery reductions dominated the time difference. If $t_1 - t_2$ is negative, then DecryptTime( $g$ ) < DecryptTime( $g_{hi}$ ), and the multi-precision multiplication dominated the time difference.

To overcome the effect of using sliding window we query at a neighborhood of values $h$-$n$, $h$-$n$-1,...$h$-1, $h$, $h$+1, $h$+2, ..., $h$+$n$, and use the result as the decrypt time for $h$ (and similarly for $h_{hi}$). In that case we determine the time difference by taking the difference of the following two equations:

$$T_L := \frac{1}{2b+1}\sum_{i=-b}^{b} DecryptTime((h+i)\cdot R^{-1/2} \bmod n)$$

$$T_H := \frac{1}{2b+1} \sum_{i=-b}^{b} DecryptTime((h_{hi} + i) \cdot R^{-1/2} \bmod n)$$

and then compute $\Delta := T_L - T_H$. This could be repeated for $b$ = 100, 200, 300, 400 etc. for finer results. As neighbourhood values grows, $| T_L - T_H |$ typically becomes a stronger indicator for a bit of $q$ (at the cost of additional decryption queries).

The Boneh & Brumley [4] attack exploits the multiplications with the base (multiplied with the Montgomery constant $R$) instead of exploiting the multiplications with the second power of the base (multiplied with $R$) in the initialization phase of the table.

### 5.2.1 Experimental Setup

The attack were performed against OpenSSL 0.9.7. The experimental setup consisted of a TCP server and client both running on a Pentium 4 processor with 2.0 GHz clock speed and RedHat Linux 7.2 on it. We performed the attacks on server running on Pentium Xeon processor with 2.4 GHz and running Linux. We also tested our attack on campus network running client on a very fast *bee* machine which is a Linux Beowulf cluster, all servers and clients compiled using gcc 2.96. All keys used in the experiments were generated randomly using OpenSSL's key generation routine.

### 5.2.2 Client –server design of the timing attack for the experiment

The client and server both were implemented in C. The pseudocode for server is given in the figure below:

```
1. Create a TCP socket. Binds socket to a port.
2. Generate a 1024 bit key by calling RSA_generate_key() of
   Crypto library.
3. Accept a connction from the client.
4. Send the public modulus (n) to the client.
5. while(true)
6.       Recieve the guess.
7.       Decrypt it using RSA_private_decrypt().
8.       Send end of decryption message back to client
```

Fig. 5.2 Pseudocode of the Server Attacked

The pseudo code for attacking client is as follows. The `Round()` function here rounds the argument to the closest integer:

```
1. Create Tcp Socket. Connect to Tcp Server.
2. Receive n from server.
3. Generate a 512 bit guess in g.
4. Set the first 2 bits of g equal to 1 which is the first two
   bits of q. set rest of the bits to 0.

5. For i = 3 to 256     /* i is the bit position */
      a. Set T0 = T1 = 0.
      b. Set g0 = g1 and set bit i to 1 in g1.
      c. Calculate h0 = Round( sqrt( g0 ) ).
      d. Calculate h1 = Round( sqrt( g1 ) ).
      e. for neighbourhood s = 0 to 1000

            i.  Calculate u0 = R05^{-1} * (h0+s) (mod n)
           ii.  Send the above guess 7 times and record the
                difference in clock ticks from the time the
                message is sent to the time end of decryption is
                received each time.
          iii.  Set t0 = median of these 7 values.
           iv.  T0 = T0 + t0


            v.  Calculate u1 = R05^{-1} * (h1+s) (mod n)
           vi.  Send the above guess 7 times and record the
                difference in clock ticks from the time the
                message is sent to the time end of decryption is
                received each time.
          vii.  Set t1 = median of these 7 values.
         viii.  T1 = T1 + t1

      f. Calculate (T0-T1).
      g. If difference is large, bit is 0 else bit is 1, copy the
         correct bit to g.
6. Return( g ).
```

Fig. 5.3 Pseudocode for attacking client

# 6 Experimental Results

We performed the timing attack under different parameters. Experiments were conducted in interprocess as well as networked environment. In the former case the server and attacking client both were running on the same machine whilst in the latter case server was run on a remote machine and attacking client was run on a fast local machine to expose RSA private key information.

The success rate of recovering private key from the server varies greatly with the correct parameter values and incorporation error correction logic in the client.

The graph in figure 6.1 depicts the value of decryption time recorded by the attacking client (in CPU clock cycles) for bits ranging from 0 to 256. The lower of the two curves is for time taken to decrypt while the bit being guessed was 1 and the other higher curve shows the decryption time while the bit being guessed was 0 in the factor $q$. One can observe that for lower bit positions, especially for bits ranging from 0 to 32 the curves overlap and the difference between the timings of different bit values is inconsistent. But there is noticeable difference for higher bit values. We explored further in our experiments about what is going on with lower bit ranges and higher bit ranges as well as techniques to enhance the distinction between bit timings.

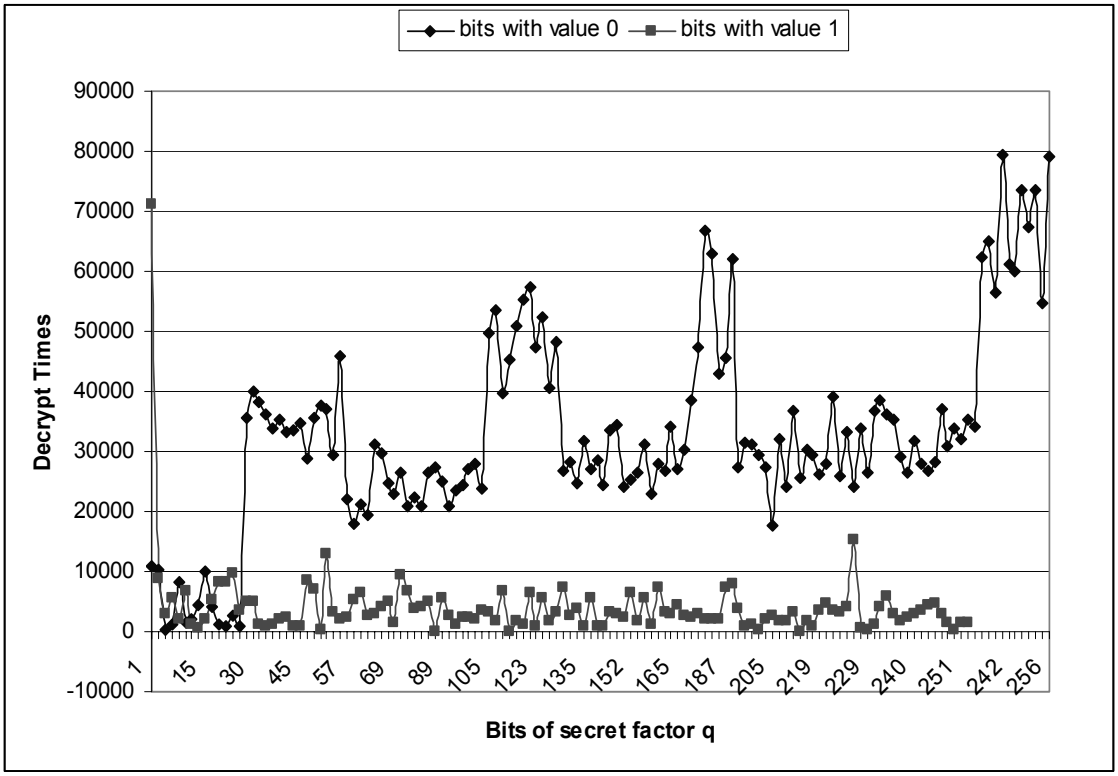


Fig 6.1 Timing attack on bits ranging from 0 to 256 with neighbourhood = 200.

We performed further experiments to enhance the distinction so that it is even easier to mark out bits of the guess *g*. We recorded repeated decryption timings to look at more clearly how the decryption times differ if the bit is 0 or if it is 1. Graphs in figure 6.2 shows that decryption timings for bit value 0 are very much constant whereas they fluctuate if the bit value is 1. This gives a clue to repeatedly sample the decryption timings and record the trend of values to get a clearer indication of the bit value currently under attack.

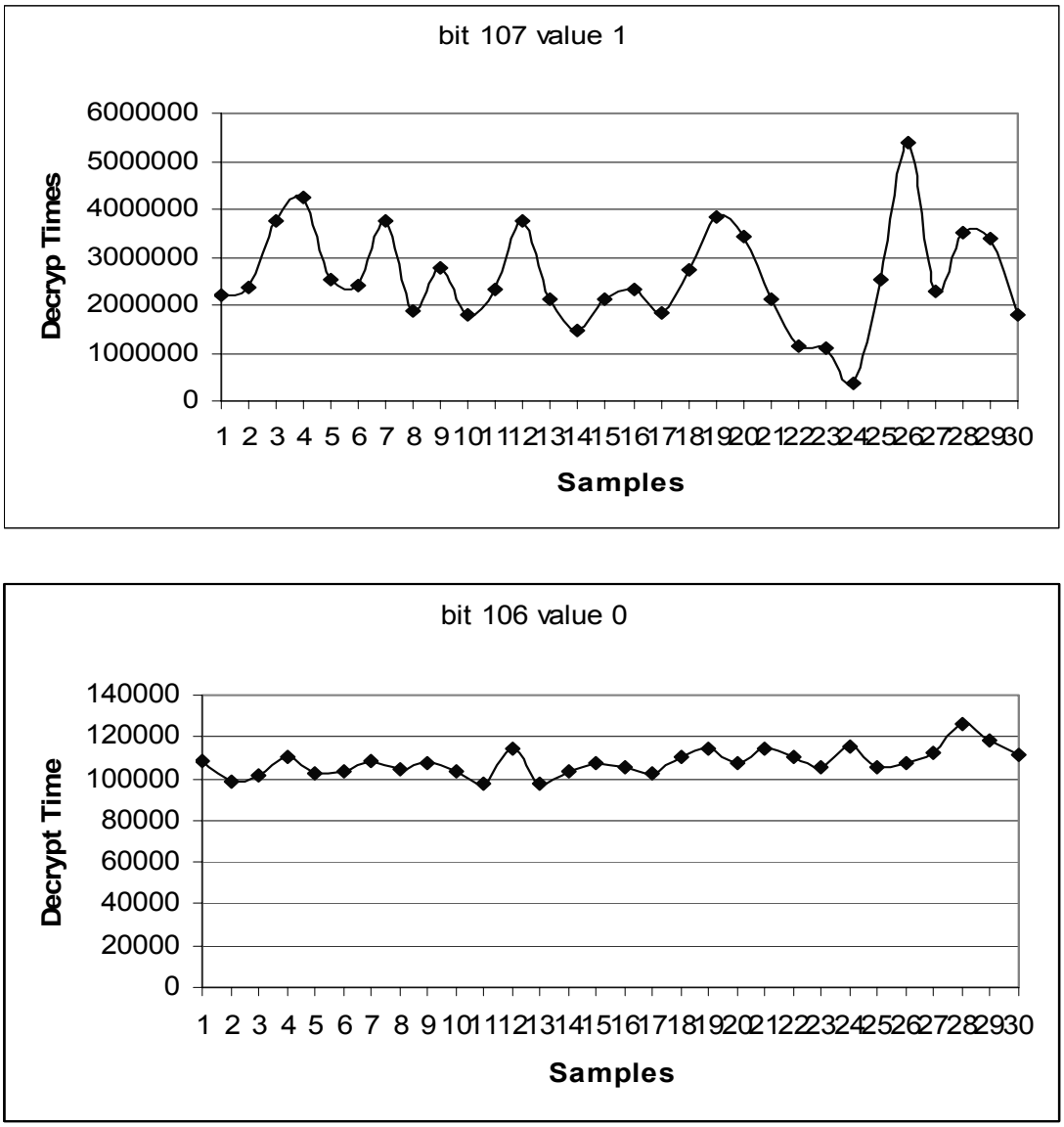


Fig. 6.2 Repeated samples of decryption timings for bits 106 and 107.

The timing data shown in figure 6.3 clearly confirms this observation that the bits could be distinguished based on the trend of their decryption timings. Bits that are 1 in value yield almost constant and low decryption timings while for bits that are 0 in value decryption times are high and they fluctuate.

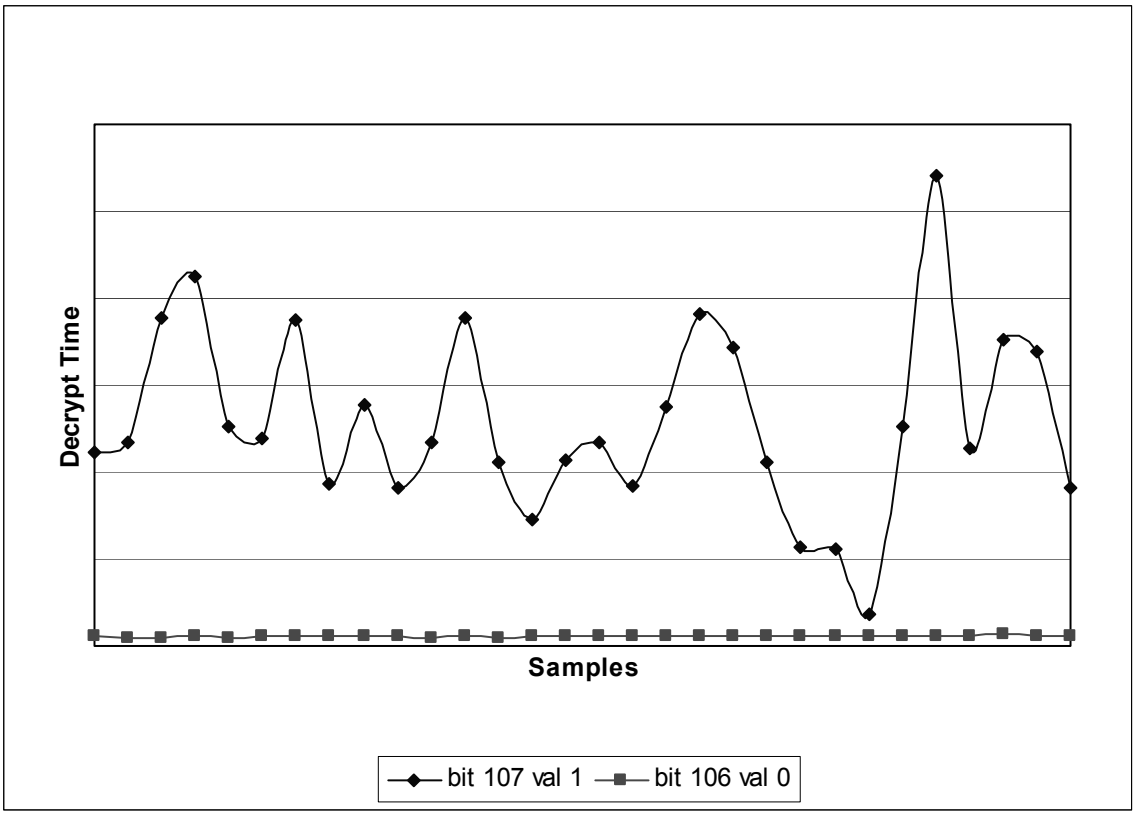


Fig 6.3 Trend in decryption timings for different bit values

Figures 6.4 depicts that for very low bit positions (here bit positions 16 and 17) it is difficult to predict the bit values even with higher neighbourhood values. Only more extensive sampling of decryption timings can eliminate the errors in guessing bit values for the factor $q$.

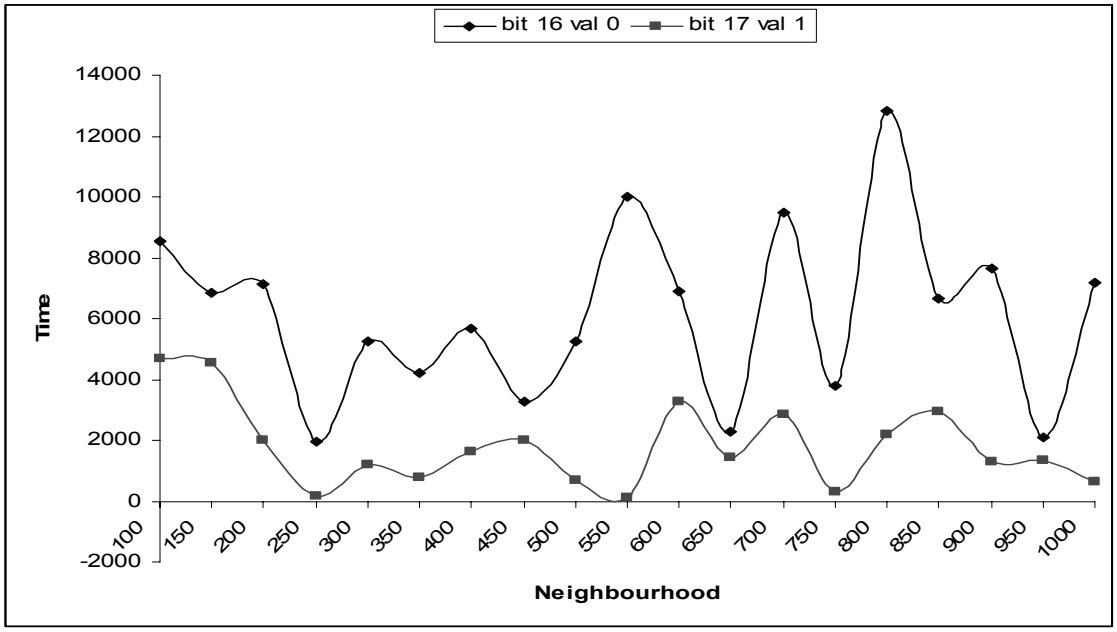


Fig 6.4 Lower bit positions of *q* are difficult to guess

Figure 6.5 Shows the gap between decryption timings of comparatively higher bits.

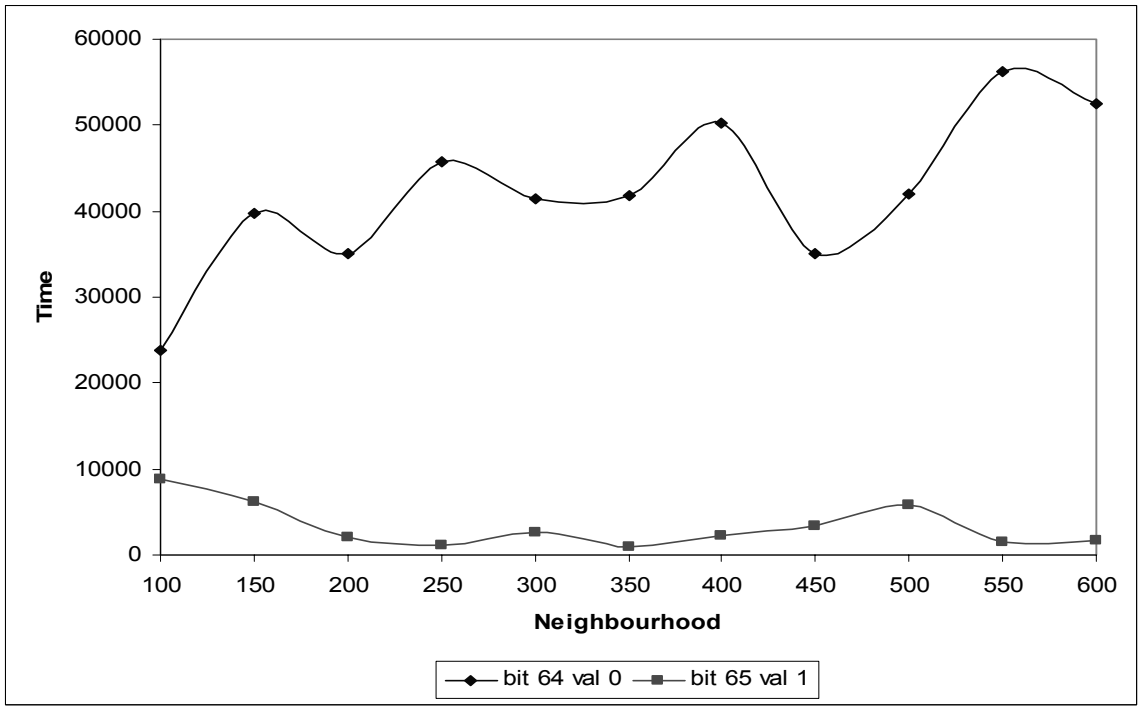


Fig 6.5 Higher bit position can be distinguished clearly

Figure 6.6 shows the effect of increasing neighbourhood values. We varied the neighbourhood size from 200 to 1000. Because we accumulate the timing difference values in out attack so the distinction between bits with different values becomes more pronounced. But this usually varies from key to key.

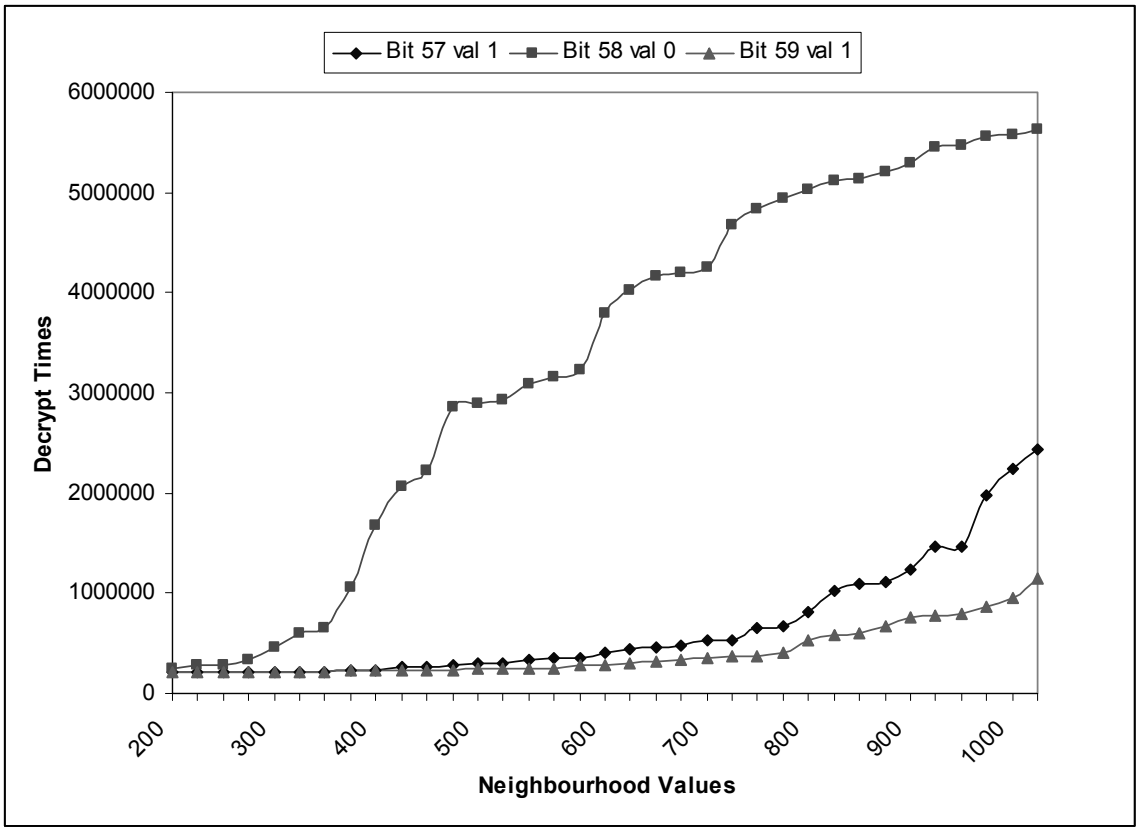


Fig 6.6 Effect of increasing neighbourhood values for bits 57, 58 and 59.

Figure 6.7 shows the repeated decryption timings for a small bit range with high neighbourhood value (1000). One can see that repeating the decryption queries makes it easier to distinguish among the bit values. Fig 6.8 shows the actual bit values of the factor $q$ for easy comparison. The actual bit values in $q$ were 152 = 1, 153 = 0, 154 = 1, 155 = 156 = 0, 157 = 158 = 159 = 1 and 160 = 0.

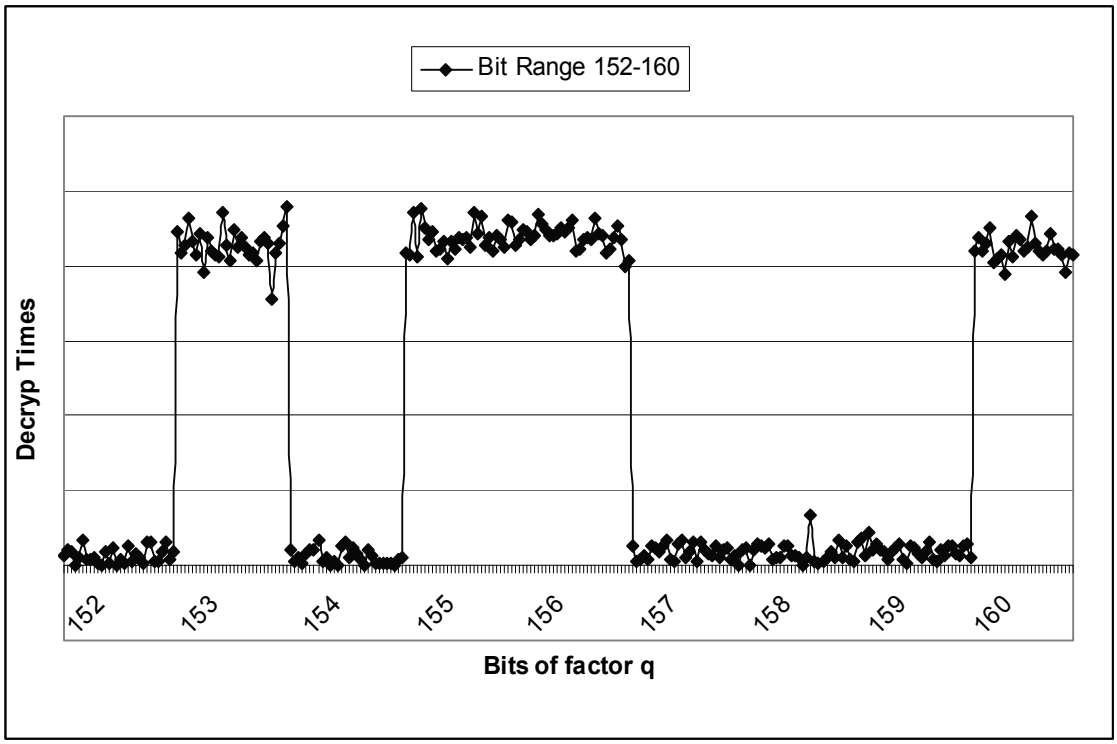


Fig 6.7 Decryption timings with repeated samples for bit range 152 – 160.

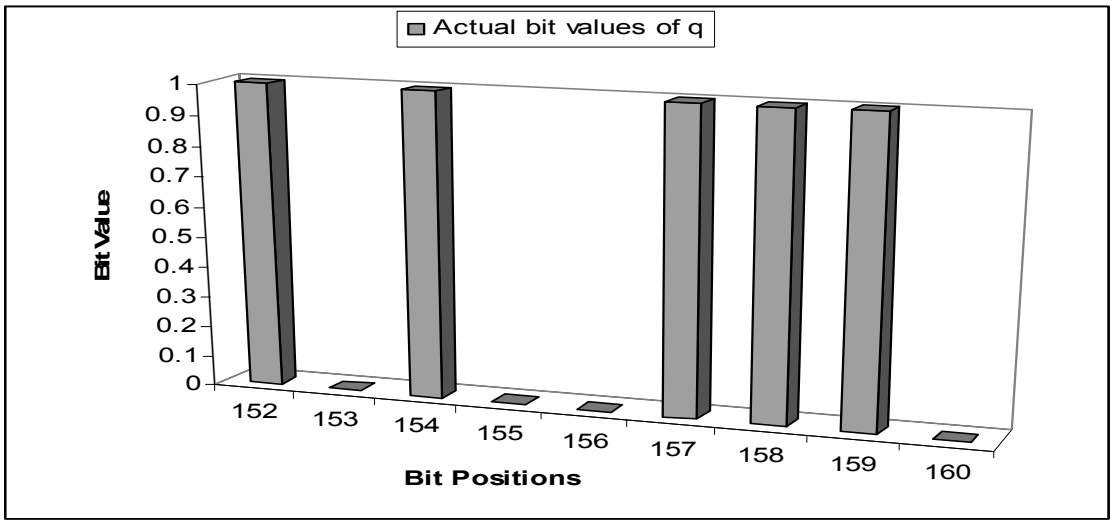


Fig 6.8 Actual bit values of factor *q* shown graphically

It is clear from the results that effectiveness of presented timing attack could be significantly improved by adopting repeated sampling or special error correcting strategies. But use of such techniques results in increased time to attack.

# 7 Countermeasures

Although the timing attack is a serious threat, there are simple counter measures that can be used, including the following:

**Constant exponentiation time:** Ensure that all the exponentiations take the same amount of time before returning a result. This is a simple fix but it does degrade performance.

**Random delay:** To achieve better performance and yet secure against adversary timing decryption random delay could be used. Here a random delay to the exponentiation algorithm is added to confuse the timing attack. But Kocher [5] pointed out that if defenders do not add enough noise, attackers could still succeed by collecting additional measurement to compensate for the random delays.

**Blinding:** This could be one of the most effective counter measures used to thwart potential timing attacks. Here the ciphertext is multiplied by a random number before performing exponentiation. This process prevents the attacker from knowing what ciphertext bits are being processed inside the computer and therefore prevents the bit-by-bit analysis essential to the timing attack. In RSA implementation with blinding feature enabled the private-key operation $M = C^d \bmod n$ is implemented as follows:

1. Generate a secret random number $r$ between 0 and $n$-1.
2. Compute $C' = C(r^e) \bmod n$, where $e$ is the public exponent.
3. Compute $M' = (C')^d \bmod n$ with the ordinary RSA implementation.
4. Compute $M = M'r^{-1} \bmod n$. In this equation, $r^{-1}$ is the multiplicative inverse of $r \bmod n$. This is the correct result because $r^{ed} \bmod n = r \bmod n$.

There is a 2 to 10% penalty in performance if blinding is enabled in RSA exponentiations.

# 8 Conclusion & Future Work

This project implemented timing attack on software implementation of RSA cryptosystem that uses CRT, Montgomery's multiplication and reduction and sliding window techniques. The experiments were based on the similar attack suggested by Werner Schindler on hardware devices like smart cards [2]. We demonstrated here that counter to current belief, timing attacks are effective even when carried out for software implementations and in complex networked environments.

To avoid the actual possibility of such attacks, taking countermeasure is very much necessary, since even networked servers are vulnerable. Use of blinding or other techniques discussed in previous sections is recommended strongly in contemporary RSA implementations.

These attacks could be further improved to reduce the attack time and the number of queries required to guess a single bit of RSA secret parameter. One possible strategy could be to combine various approaches, e.g. Boneh & Brumley [4] with the current to gain maximum advantage. Use of various statistical tools could also be used to make the attack even more effective. Also similar techniques could be used to more powerful attacks where several bits could be guessed simultaneously.

# References

[1] OpenSSL Project. Openssl. `http://www.openssl.org`.

[2] Werner Schindler. A timing attack against RSA with the Chinese remainder theorem. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, pages 109–124, 2000.

[3] Intel. Using the RDTSC instruction for performance monitoring. Technical report, 1997, `http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM`.

[4] D. Boneh and D. Brumley. Remote Timing Attacks are Practical. *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[5] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of Crypto 96*, *LNCS 1109*, *Springer, 1996*, pp. 104—113

[6] Cetin Kaya Koc, Tolga Acar, and Burton S. Kaliski Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, pages 26--33, June 1996.

[7] P. Montgomery, "Modular Multiplication Without Trial Division," *Mathematics of Computation*, Vol. 44, No. 170, 1985, pp. 519--521.

[8] Menezes, van Oorschot, Vanstone: Handbook of Applied Cryptography, Algorithm 14.85.

[9] C .K. Koc, "High-speed RSA implementations," *RSA Laboratories Technical Notes and Reports* TR 201, RSA Laboratories, Nov. 1994