Frankfurt University of Applied Sciences

Master Thesis

Revisiting Remote Timing Attacks on OpenSSL

Author:
Md Intekhab Shaukat

1st Academic Supervisor:
Prof. Dr. Christian Baun

2nd Academic Supervisor:
Prof. Dr. Matthias F. Wagner

Industrial Supervisor:
Mr. Herve Seudie

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in

High Integrity Systems
FB 2: Informatik & Ingenieurwissenschaften

April 2015

Declaration of Authorship

I, Md Intekhab Shaukat, declare that this thesis titled, 'Revisiting Remote Timing Attacks on OpenSSL' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a masters degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- This work was done as an industry thesis with Robert Bosch GmbH, and no company proprietary information is published in this report.

Signed: ______________________________

Date: ______________________________

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Abstract

FB 2: Informatik & Ingenieurwissenschaften
High Integrity Systems

Master of Science

Revisiting Remote Timing Attacks on OpenSSL

by Md Intekhab Shaukat

Timing attacks have the power to retrieve a secret quite simply what a classic attack finds difficult. Basic idea behind timing attacks is that the implementation of a cryptosystem has timing difference based on input data or the secret. The attacker leverage this behaviour using selective inputs to recover the secret. These attacks had been demonstrated mainly on hardware cryptosystems such as smart cards and it was a general belief that software systems could not be attacked due to its nondeterministic execution time behaviour. In 2003, Brumley and Boneh[12] broke this notion and showed that it is possible to retrieve the private key of a web server using timing attack. They showed that implementation of OpenSSL, an SSL library, of RSA is vulnerable to timing attack. Schindler et al [19] once again implemented similar attack in 2005 and provided an improvement in the efficiency by a factor of more than 10. RSA is an asymmetric cryptography algorithm which is also used as key exchange algorithm by web servers. We revisit these attacks and implement these in this work. We recover the private key of a server which uses OpenSSL for providing data encryption. We compare these attacks as well in the context of efficiency.

We re-establish that timing attacks are not limited to theories rather they are possible and could leak big deal of information. Developers can no more ignore these vulnerabilities. We must defend against these attacks using suitable defences.

Acknowledgements

I would like to express my gratitude towards my industrial supervisor *Mr. Herve Seudie* at Robert Bosch GmbH for providing me this opportunity and for his thorough guidance and support. His encouragement and valuable inputs have been key to the successful work. I would also like to thank my academic supervisor *Prof. Dr. Christian Baun* for undertaking this work. His calm and sound supervision has helped a lot. I extend my sincere thanks to my second academic supervisor *Prof. Dr. Matthias F. Wagner*. I express my thanks to *Sebastian Schizel* for his talk on Timing Attacks which caused the major breakthrough.

I take this opportunity to thank all my friends especially *Muzammil*, *Zeeshan*, and *Osama* for their prayers and altruistic support. Their intermittent encouragement has kept me focused and motivated. I appreciate my friends - *Anand*, *Modassar*, *Nadim*, and *Viplav* for being there for me. I thank *Sameer* for the idea that set the stage to start this venture.

In the end, I would like to place on record my deepest gratitude towards my family who undoubtedly have been a great support through out. I owe a lot to my elder brother *Md Aftab Shaukat* whose support has enabled me to create a niche for myself.

Last but not the least, I thank one and all, who directly or indirectly, have been part of this journey and supported me even in the least bit.

Sincerely,
Md Intekhab Shaukat

Contents

Abbreviations

AES	**A**dvanced **E**ncryption **S**tandard
CRT	**C**hinese **R**emainder **T**heorem
DES	**D**ata **E**ncryption **S**tandard
DH	**D**iffie **H**ellman
GMP	**G**NU **M**ultiple **P**recision
IDS	**I**ntrusion **D**etection **S**ystem
IoT	**I**nternet **o**f **T**hings
LSB	**L**east **S**ignificant **B**it
LTS	**L**ong **T**erm **S**upport
MSB	**M**ost **S**ignificant **B**it
NIC	**N**etwork **I**nterface **C**ard
PKI	**P**ublic **K**ey **I**nfrastructure
RC4	**R**ivest **C**ipher 4
RSA	**R**ivest **S**hamir **A**dleman
SCA	**S**ide **C**hannel **A**nalysis
SOA	**S**ervice **O**riented **A**rchitecture
SSL	**S**ecure **S**ocket **L**ayer
SWE	**S**liding **W**indow **E**xponentiation
TLS	**T**ransport **L**ayer **S**ecurity
TSC	**T**ime **S**tamp **C**ounter
WWW	**W**orld **W**ide **W**eb

Chapter 1

Introduction

Invent of **Internet** is the result of the quest to have a life which is connected, fast, and easy. Since its invent the World Wide Web (WWW) has grown massive. People exchange information over the Internet excessively. Irrespective of individual, group, or organization *information* is a vital part of their lives. They want to have information secure. People share information over the web, which might cross several continents through plenty of routers to reach its destination. In such scenario, *information security* becomes necessary part of the computing world. This necessity has given birth to *Cryptography*, which is a very secure way to share information. Today, we use *https* in web browser and we feel that we are talking to the right person and our information is secure. This feeling of security is due to Cryptography. But, wait. Even now, we sometimes hear the news of information breach in such systems, like *Heartbleed*. Nevertheless, Cryptographic methods are not pure of any loopholes and researchers have strived hard to find such loopholes over the years. These efforts have resulted in the subject *Cryptanalysis*. There are properties of cryptographic methods, which might give away sensitive information if not implemented properly. Side Channel Analysis (SCA), a branch of Cryptanalysis, deals with such topics.

In this chapter, we will discuss about Cryptography and its types. We will also introduce RSA, a cryptographic method. Side Channel Analysis will be introduced briefly and Timing Analysis, a type of Side Channel Analysis, will be discussed in detail. We will go on further to talk about RSA and do timing analysis of it. Here, we will talk about the basic idea behind timing attack on RSA.

1.1 Cryptography

Cryptography is a technique through which one can send information in a jumbled manner such that the receiver would not be able to understand the piece of information until he or she knows a secret. This technique is not something new. The history of cryptography dates back to over thousand years to Romans, Greeks, and, Egyptians. Julius Caesar is known to replace the letters of original message with other letters in the alphabet. He used this method to communicate secretly to his generals. Some used to shuffle the words of the original message. The Greeks are known to hide the original message in order to keep it secret. According to Wikipedia [1], *Herodotus* hides the message in the form of a tattoo on a slave's head. *Invisible Ink* could be one example of similar method in modern times. In ancient times, such methods have been used mainly for national and political security reasons. Those methods are known to be broken after careful analysis.

As discussed earlier, there is a great need of security and privacy in modern times in computer science. Researchers and Scientists have studied the ancient procedures and extended the methods to design more robust and secure methods. Cryptography deals with such study of methods and techniques to communicate securely between two parties. According to the book *Handbook of Applied Cryptography* [2], Cryptography is defined as *"the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication".* Cryptography must fulfill the information security goals - *Confidentiality, Data Integrity, Authentication, and Non-repudiation.*

Let us now briefly discuss a few terminologies used in cryptography, which will enable the readers of this report to understand the terms when they are used. We refer the original message as *plaintext.* The method used to modify the original message is called the *cipher* and used secret is known as the *key.* The resulted message after modification is called the *ciphertext. Encryption* is the process of converting the plaintext into the ciphertext and *Decryption* is the process of converting the ciphertext into the original message or plaintext. The key used in encryption is referred as *Public or Encryption Exponent, e* and the key used in decryption is referred as *Private or Decryption Exponent, d. Cryptanalysis* involves the study of the cryptographic methods in order to break it.

Based on the number of keys used in the process, cryptography can be divided into two categories - *Symmetric Cryptography* and *Asymmetric Cryptography.* Let us discuss these categories briefly.

1.1.1 Symmetric Cryptography

Symmetric Cryptography is a type of Cryptography in which the keys used in Encryption and Decryption are same or it is easy to derive one key from the other and vice-versa [2]. The term *symmetric* comes from the fact that both keys are same. This is also known as *private-key cryptography.* Data Encryption Standard (DES), Advanced Encryption Standard (AES), and RC4 are examples of symmetric cryptography.

The symmetric ciphers can also be classified as *block cipher* and *stream cipher.* In block cipher, plain text is divided into fixed size blocks and then each block is processed at a time. On the other hand, stream cipher processes the plaintext bit-by-bit or character-by-character.

Each method has its advantages and disadvantages. Symmetric cryptography is fast and, usually, the key length is shorter. These are best suited for message encryption and decryption. In symmetric cryptography, one of the major issues is the key management. It is really a challenge to secretly share the symmetric key between the two parties.

1.1.2 Asymmetric Cryptography

Unlike symmetric cryptography, Asymmetric Cryptography has two different keys for encryption and decryption. The two keys are mathematically related, but it is infeasible to derive one key from the other. This scheme is widely known as *Public-key Cryptography.* The two keys are called Public Key and Private Key. In general sense, public key is used in encryption while private key is used in decryption. This has been considered one of the most revolutionary concept in computer science. Diffie-Hellman (DH), proposed by Whitfield Diffie and Martin Hellman, and RSA are widely used asymmetric cryptography methods.

These methods overcome the major problem of key management in symmetric schemes. On the other hand, the key lengths are comparatively larger. Also, these are slower than symmetric schemes. These features make them suitable for Authentication and Key Exchange. RSA is widely used in *Digital Signatures*, a method of authentication. Considering the properties of both symmetric and asymmetric schemes, combination of both, sometimes referred as *hybrid cryptography*, is often used in client-server environment.

Since our study is based on RSA, we will discuss it in detail in later sections.

1.2 RSA

In 1977 Ron Rivest, Adi Shamir, and Leonard Adleman brought a revolution in cryptography, when they proposed a very robust asymmetric cryptography scheme. This scheme is named *RSA* on behalf of their names. According to the book [2], RSA is the most widely used cryptosystem. It provides both secrecy and authentication. This scheme involves three steps - Key Generation, Encryption, and Decryption. First a key pair (Public Key and Private Key) is generated. These keys are then used in encryption and decryption. In general, encryption uses public key and decryption uses private key. Hence, they are often referred as *encryption or public exponent, e* and *decryption or private exponent, d* respectively.

Alice (the receiver of the message) shares his public key with Bob (the sender). Bob encrypts the message using Alice's public key and sends the ciphertext to Alice. Alice decrypts the ciphertext using his private key to get the message. Figure 1.1 depicts this scheme.

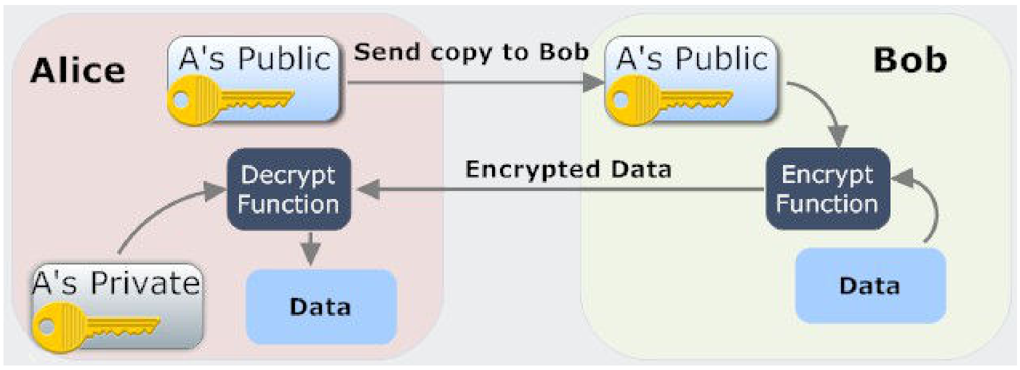


FIGURE 1.1: RSA Encryption-Decryption scheme

Key Generation, Encryption, and Decryption algorithms are explained below.

Once public key and private key are generated, one can use below algorithm to encrypt and decrypt a message. Let us assume that A wants to send a message to B.

Our attack is based on implementation of RSA decryption in OpenSSL. So, let us look at the specifics of RSA decryption implementation in OpenSSL.

Algorithm 1 RSA Key Generation Scheme[2, p. 286]

```
1: Generate two large prime numbers $p$ and $q$, roughly of the same size.
2: Compute $N = pq$ and $\phi = (p\text{-}1)(q\text{-}1)$
3: Choose a random number $e$, such that, $1 < e < \phi$ and $\gcd(e,\phi) = 1$
4: Compute $d$, such that, $1 < d < \phi$ and $ed \equiv 1 \pmod{\phi}$.
5: $(N,e)$ is public key and $d$ is private key.
```

Algorithm 2 RSA Encryption & Decryption[2, p. 286]

```
Encryption by A
1: A must obtain the public key of B.
2: Compute an integer equivalent $m$ of the message.
3: Compute $C = m^e \pmod{N}$
4: $C$ is the encrypted message. Send $C$ to B.
Decryption by B
1: B will receive C.
2: Compute original message as $m = C^d \pmod{N}$
```

1.3 OpenSSL

OpenSSL [3] is a cryptographic library which provides implementation of SSL/TLS protocols incorporating many cryptographic algorithms including RSA. This is an open source library under Apache and BSD License, and is most used cryptographic library. Most web servers use this to provide *https* connection. This is considered to be an extension of a library **SSLeay**, a work of Eric Andrew Young and Tim Hudson.

To speed up the decryption, OpenSSL uses a combination of few algorithms to solve the RSA decryption equation, $m = C^d \bmod N$. It uses Square and Multiply, Sliding Window Exponentiation, Montgomery Reduction, Chinese Remainder Theorem, and Karatsuba Multiplication. In this section, we will get to know these algorithms.

1.3.1 Square and Multiply

The equation of the form $s = x^d \bmod N$ is called modular exponentiation. A faster method to solve such equation is called *Square and Multiply Algorithm*. The method is described in algorithm 3. First, we convert the exponent d in its binary form such that $d_0 = 1$. Then, we start with value x and iterate through the bits of d to get the result s.

Algorithm 3 Square and Multiply Algorithm : $s = x^d \bmod N$

```
1: $d = (d_0, d_1, .., d_n)_2$, where $d_0 = 1$
2: $s = x$
3: for $i = 1$ to $n$ do
4:     $s = s^2 \bmod N$
5:     if $(d_i == 1)$ then
6:         $s = s \cdot x \bmod N$
7:     end if
8: end for
9: return $s$
```

Although, this method is faster, it has scope of improvement. We process the bits of d one by one, which can be improved by considering a group of bits at a time.

1.3.2 Sliding Window Exponentiation

Sliding Window Exponentiation (SWE) is a variant of *Square and Multiply Algorithm* and is used to solve modular exponentiation. It improves the *Square and Multiply Algorithm* by processing a group of bits of d at a time. This algorithm works in two phases - (1) Table Initialization Phase, and (2) Exponentiation Phase.

1. **Table Initialization Phase**: In this phase, we pre-compute the odd powers of x including x^2 such that $x^1, x^2, x^3, x^5, ...$ and saves it into a table. So, we prepare a table with odd powers of x.

2. **Exponentiation Phase**: This is the actual exponentiation phase where we process w bits at a time. w is called window size. During this phase, we carry out multiplication by one of the table entries that we prepared in first phase. The table entry is decided by the value of w.

OpenSSL uses window size of 5 for 1024-bit modulus.

1.3.3 Chinese Remainder Theorem

A Chinese mathematician *Sun Tzu* developed Chinese Remainder Theorem (CRT). In general, it is used to find out a number which when divided by some divisors leaves given remainders. For example, find out the least number which when divided by 5 gives remainder 3 and when divided by 7 gives remainder 2. A comprehensive application could be found in the book [4].
openSSL uses CRT to solve decryption equation $m = C^d \pmod N$ where $N = pq$. This equation is solved in two steps. First step calculates $m_1 = C^{d_p} \pmod p$ and $m_2 = C^{d_q} \pmod q$. Second step combines m_1 & m_2 to get m. d_p and d_q are precomputed from d. This is explained in below algorithm.

Algorithm 4 Chinese Remainder Theorem[5, p. 466]

```
1: Compute $d_p \equiv d \bmod (p-1)$ and $d_q \equiv d \bmod (q-1)$.
2: Compute $m_1 \equiv C^{d_p} \pmod p$ and $m_2 \equiv C^{d_q} \pmod q$
3: Compute $u \equiv q(q^{-1} \pmod p)$ and $v \equiv p(p^{-1} \pmod q)$
4: Compute $m \equiv u \cdot m_1 + v \cdot m_2 \pmod N$
5: Return $m$ as decrypted message.
```

CRT results in speed up by a factor of four in comparison to normal modular exponentiation.

1.3.4 Montgomery Multiplication

The multiplication of the form "$x \cdot y \bmod q$" is called *modular multiplication*. During Sliding Window Exponentiation (SWE), modular multiplication is performed at every step. Peter L. Montgomery published a paper [6] in 1985 to perform such calculations efficiently. It avoids integer division. This method is quite fast on hardware and software since it involves multiplication and division by powers of 2 denoted by R, which is achieved by cheap shift operations. We will refer this as **MM**. So, it performs $MM(x,y) = x \cdot y \cdot R^{-1} \bmod q, \quad where \quad R > q$. First x and y are converted into Montgomery form. Montgomery form of x is $xR \pmod q$. Then, multiplication $xR \cdot yR = cR^2$ is performed. After this, modular reduction of the product is done as $cR^2 \cdot R^{-1} = cR \bmod q$. The result is in Montgomery form which can be used in subsequent Montgomery operations. At the end q is subtracted from cR if it is greater than q to make sure output is in the range $[0, q)$. This step is called **Extra Reduction**. We will see later how this would help in our attack. Eventually, the result is put back into normal form by multiplying the result by $R^{-1} \bmod q$ at the end of exponentiation. This algorithm has been put in steps below.

Algorithm 5 Montgomery Multiplication, MM(x,y) $= x \cdot y \cdot R^{-1} \bmod q$

```
1: Compute $S = x \cdot y$
2: Compute $z = S \cdot q^{-1} \bmod R$
3: Compute $S = (S - z \cdot q)/R$
4: if $S > q$ then
5:     $S = S - q$
6: end if
7: return $S$
```

1.3.5 Multiplication Routines

The calculations in RSA decryption also involves multiplications of very large numbers represented in over 1000-bits. These numbers are called *arbitrary-precision or multi-precision numbers* in mathematical terms. There are libraries that can be used to deal with such numbers e.g. OpenSSL and GMP [7]. These libraries represent these numbers as sequence of words in computer. Since these numbers are very large, we need fast algorithm to do the multiplication. Karatsuba

Multiplication is the answer. This is faster than normal multiplication. Time complexity of this algorithm is $O(n^{1.58})$. However, there is one pre-requisite of this algorithm that the operands of multiplication should have same number of words. Thus, OpenSSL uses two multiplication routines - Karatsuba (for operands of equal words) and Normal (for operands of different words). Normal multiplication has time complexity of $O(mn)$ for operands of word sizes m and n. Clearly, it will take $O(n^2)$ for equal size operands.

1.4 Side Channel Analysis

Side Channel Analysis can be thought of knowing about a secret using certain features. You and I have been using such technique for long without having noticed. You might be aware of how your friend/relative behave in certain situations. So, when you see such behavior you interpret the situation without being told. For example, your friend might *lisp* when he is in tension. According to the article [8], Pizza shop owners around White House claimed to know what is going on inside the White House because pizza orders used to go up before every attack. These are examples of side channel analysis.

Cloud computing is becoming ubiquitous today. There is a shift from desktop applications to software-as-a-service, where applications are being provided over web. To provide privacy and security encrypted transmission is used. They are considered to be safe because adversary cannot understand the encrypted data. However, certain properties of the encrypted traffic can leak out sensitive information e.g. packet size, packet sequence, and timings etc. These properties are called Side Channels. Chen et al [9] have shown that such information can give away the sensitive information of users.

So, *Side Channel Analysis* is an attack, which is based on certain characteristics of the implementation of the cryptosystem [10] and those characteristics are called *Side Channels*. Power Consumption, Timing detail, Packet sequence, Packet size, Electromagnetic Waves, Sound, etc. are examples of Side Channels. Researchers have shown several attacks using these details.

Our study is based on one of the side channels - Timing.

1.4.1 Timing Side Channel Analysis

In Timing Attacks or Timing Side Channel Analysis, we use timing details of the implementation of the cryptosystem to break it. As we know that algorithms have different execution time for different inputs. If we can profile the execution time of different inputs, we can use the execution time to guess the input. Basically, this is the idea behind a timing attack. These vulnerabilities usually get overlooked by developers due to lack of knowledge. Also, Our tests are not bothered to catch such issues. The attacker is assumed to have knowledge of the implementation. So, if the algorithm, which involves calculation with secret, has different execution profile depending on input, it might give away the secret.

To have a better understanding, let us consider an algorithm of a login system.

Algorithm 6 Login System

```
1: if (User exists in the system) then
2:     if (Password Match) then
3:         return "success"
4:     else
5:         return error "An error occurred"
6:     end if
7: else
8:     return error "An error occurred"
9: end if
```

The above algorithm returns the same error if either of username or password is incorrect. Looking at the error one cannot decipher which one is incorrect. However, timing analysis can leak out such information and one can infer if a user exists in the system or not. Figure 1.2 makes it clearer. Just to inform the readers that one of the basic step in hacking is to list down the existing users. Hackers can take advantage of such timing differences.

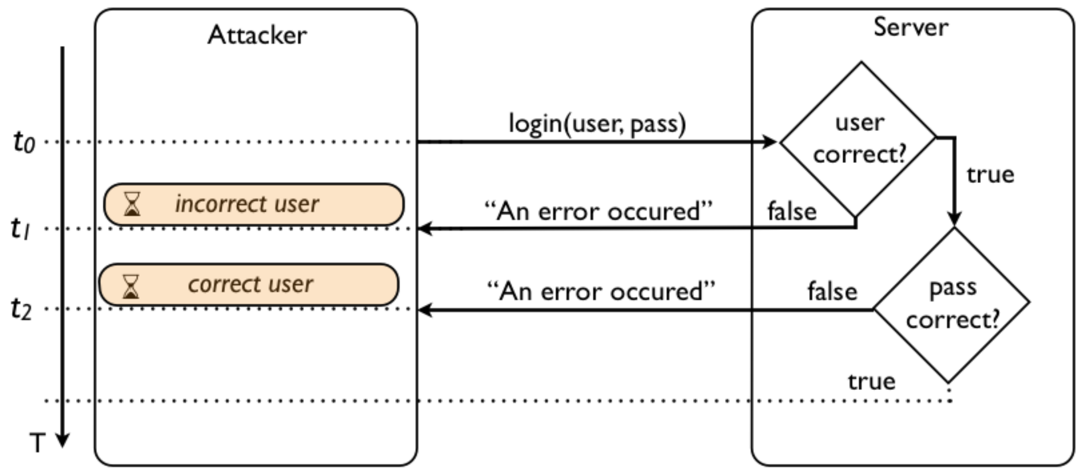


FIGURE 1.2: Timing Analysis of Login System

Our work focuses on the timing analysis of RSA algorithm. Let us discuss RSA in the light of timing side channel analysis.

1.5 Basic Idea behind Timing Attack on RSA

In previous sections, we have discussed how RSA decryption is implemented in OpenSSL. In this section we will critically examine the implementation for timing differences that depends on input data. This would make the basis of our attack.

Researchers have identified two areas of the algorithm where decisions are made based on input data. First is **Montgomery Multiplication** and second is the choice of two **different multiplication routines**. Let us talk about Montgomery Multiplication (MM) first. We have seen that for the calculation of $g^d \bmod q$, MM performs an **extra reduction** step to keep the product in range $[0,q)$. Werner Schindler observed that the probability of an extra reduction during exponentiation $g^d \bmod q$ depends on how close g is to q. He gave a formula for the probability as below[11] :

$$Pr[\text{extra reduction}] = \frac{g \bmod q}{2R} \tag{1.1}$$

According to equation 1.1, the number of extra reductions increases as g approaches to q from below. When g is a multiple of q, $g \bmod q = 0$ which shows that number of extra reductions drops dramatically at multiples of q. The same is true for other prime factor p. This behavior is shown in figure 1.3. This figure 1.3 has been taken from Brumley and Boneh paper [12]. So, by number of extra reductions one can infer that how close g is to q. In other words, when g is near q such that $g < q$, then number of extra reductions would be greater than that when $g > q$. This suggests that **decryption time of $g < q$ would be longer than that of $g > q$.**

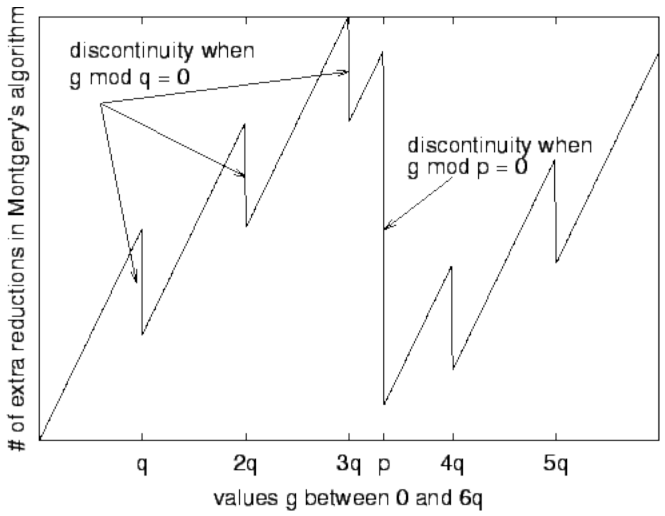


FIGURE 1.3: Number of extra reductions as a function of input g in Montgomery Multiplication

The second point in the algorithm that makes the timing difference is the choice of two different multiplication routines - Karatsuba and Normal. When g is near q such that $g < q$, then operands of the multiplication, mostly, has equal word size and OpenSSL uses fast Karatsuba multiplication. When $g > q$, $g \bmod q$ is small and most multiplication would be of different word sizes. OpenSSL would use normal multiplication, which is slower, for such cases . This suggests that **decryption time of $g < q$ would be shorter than that of $g > q$.** The same has been shown in figure 1.4, taken from [12].

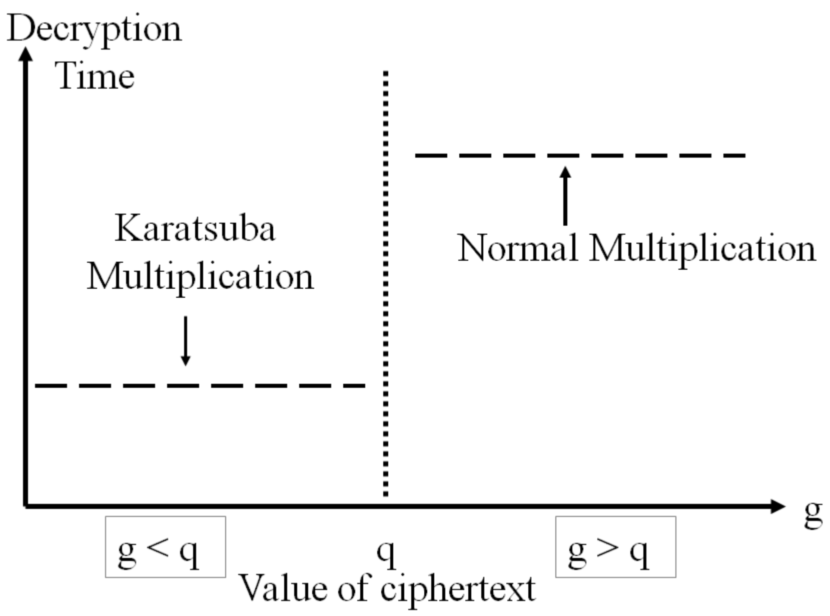


FIGURE 1.4: Two multiplication routines

The two effects are opposite and counteract each other. However, the key point is that one of them would always dominate the other. As suggested by Brumley & Boneh [12], the domination is determined by the exact environment (hardware, compiler options etc.).

Chapter 2

Problem Description

In Chapter 1, we introduced RSA and OpenSSL as an implementation of RSA. We also discussed the implementation in the light of Timing Side Channel Attack. In this chapter, we would discuss the implications of such vulnerability and motivation behind performing such attacks.

2.1 System Model

The vitality of *Information Security* is clear to every individual and organization. Every organization in the world is connected to Internet and provides its services or uses others' services using web. In such scenario, client-server architecture is inevitable. Almost every organization would be hosting or using web servers and we, as an internet user, daily visit web servers or websites for our needs. SSL/TLS protocols are the one, which provides security in such scenario, and OpenSSL is most widely used library for that. Thus, almost all web servers or websites uses OpenSSL to provide HTTPS (Secure HTTP). OpenSSL uses other algorithms also such as Elliptic Curve and Diffie-Hellman (DH). Preferred ciphers use these algorithms, which are known not to have such vulnerabilities so far. However, one can change the available ciphers of the client and make the web server to use RSA.

Not only computers but also most of the computing devices are now part of the Internet infrastructure. Mobile device, Tablet, Smart watch, Sensor, Automobile and many more devices are now connected to Internet giving rise to the infrastructure, which is called **Internet of Things (IoT)**. Either they use an Internet resource or offer one. Due to this infrastructure *Machine 2 Machine (M2M)* communication is rising rapidly. IoT is one of the growing technologies and according to experts this is the future. There is tough competition between organizations to grow in this area that makes this field more alluring. With increasing M2M communication, the need to secure such communication and the identity of the devices is also rising. They also use RSA to achieve secrecy and security.

As Organizations are growing, they are realising the business need to integrate systems, departments, and offices. They are often running different technologies. Due to this need **Service Oriented Architecture (SOA)** is heavily being used for integration. In SOA, services are offered as an Internet resource and it also involves client-server architecture. Cloud computing is a state-of-the-art too. Many organizations are offering services such as applications, platform, and infrastructure as a web service. All such growing technologies require security for their web servers.

2.2 Motivation

The world is full of people who want and strive to breach the privacy of others. People come up with solutions to increase security and privacy, at the same time, millions of hackers are trying to break it. All they want is to compromise the security of others. There might be several reasons to that e.g. war or competition between countries or organizations, jealousy, increasing own security, just for fun etc. We always hear news of such attacks. Stealing of email addresses and passwords from email servers are quite common. Celebrities' pictures have been stolen by compromising the security of cloud servers. Recently, Sony servers have been attacked and were made inaccessible to valid users. So, attacks are common and hackers would always exist.

We have already mentioned in previous section how client-server architecture is inevitable in current computing world. Also, it is bound to increase in future. So, the ubiquity of RSA makes it cynosure of cryptanalysts and attackers. The history of RSA has witnessed several attacks. The very recent one is *Heartbleed*. Cryptanalysts strive to look for vulnerabilities in such a ubiquitous algorithm to make it stronger.

Unlike classical attacks, Side Channel Attacks do not require an attacker to have the possession of systems. The *side channel information* is visible to everyone. These attacks can easily be performed remotely. Side channel vulnerabilities often creep into codes because developers are not aware of such issues. Moreover, our testing strategies do not consider such issues/vulnerabilities and they get overlooked for the same reasons. However, these issues are real and it can lead to serious security threats and implications could be devastating. For example, with our attack one can steal the private key of the server and take its identity. Now consider that a hacker is able to recover the private key of Google mail server and, now, he/she can impersonate Google. Trillions of people using *gmail service* would lose their private information and organizations/businesses would be at risk. It can create a havoc that we could just imagine. The worst part is that you would not even realize that your security/privacy has been compromised.

Bosch is a growing organization in the area of Internet of Things (IoT). Every electrical or mechanical device is running some application. It needs to talk to other devices giving rise to Machine2Machine (M2M) communications. These scenarios make Bosch interested in side channel analysis of her devices or applications. Due to these factors, Bosch is keen to setup a lab where she can critically examine her devices or applications for side channel vulnerabilities. They already have the setup for Power Analysis and Fault Analysis. This effort would make them setup the tools for Timing Analysis.

2.3 Goal of the Thesis

As mentioned in previous section that Bosch is interested in setting up a lab, where she can look for side channel vulnerabilities in her products. My work would be a part of it and it would set up things for doing timing analyses. As a preliminary step, it started to implement known timing attack on a widely used cryptosystem, RSA. This work would enable them to perform a sophisticated timing attack and to carry out advanced statistical analysis. Moreover, they would be able to examine a piece of code from timing difference perspective such that one could figure out if the code could leak out sensitive information. Summing up, the goal of the work would be to figure out what is necessary to perform timing attacks, set up tools or infrastructure for that and implement the known timing attack on RSA to retrieve the private key of a web server.

Chapter 3

Challenges

Unlike classical attacks, the attacker measures the physical properties such as power, timing etc. of the implementation in SCA. These attacks require proper steps to be taken to measure those parameters. An attacker encounters several challenges in performing such attacks. Since one measures the timings in Time Attack, we will discuss the challenges that comes in one's way in carrying out such attacks.

3.1 Precise Time Measurement

Theoretically, it sounds easy to measure the time and infer the secret by comparing the timing differences. However, practically, this is too complicated to carry out and it needs sophisticated approach. Non-determinism in execution time in real life makes the process quite challenging. The execution time of a piece of code varies significantly for a particular input. We need very precise measurement of time. In practice, we can only measure the round trip time of the message that also has variation in it called *jitter*. Moreover, there are factors that interfere with the measurement. So, the main challenges are to reduce the jitter and control the factors that affect the measurement.

3.1.1 Fine Grain Timer

The precision of measurement is very important in timing attacks. We need a timer, which is very precise and independent of system processes. One such timer is **Time Stamp Counter (TSC)**. Every processor has one such counter, which counts the CPU cycles. The counter increments per clock tick of the processor independent of actual instructions being issued. Thus, the precision depends on the actual frequency of the processor. For example, If a processor has 2 GHz frequency, it will give 2 billion ticks per second, which means 1 tick is equal to 0.5 nanoseconds. Thus we can measure up to 0.5 nanoseconds on 2 GHz machine using TSC. Below is the formula to calculate the precision.

$$Precision = \frac{1}{CPUFrequency} \text{ seconds}$$

RDTSC (read time-stamp counter) is the assembly instruction that is used to read TSC for Intel processors. The output is the 64-bit integer. The higher 32 bits are stored in EDX register and lower 32 bits are stored in EAX register. This instruction gives the cycle counts, which can be converted into time units as below.

$$\#\text{ seconds} = \frac{\#\text{cycles}}{CPUFrequency}$$

Intel processors also support out-of-order execution of instructions for optimization. It means that the statements can be executed in a different order than it appears in the source code. This feature could interfere with the measurement and would lead to incorrect number of cycles. To prevent this out-of-order execution, we need a serializing instruction. A serializing instruction forces a processor to complete all previous instructions before going forward. **CPUID**, another assembly instruction, is the solution. This instruction is basically used to identify the processor. However, we will use this instruction to stop out-of-order execution. A detailed description about RDTSC and CPUID could be found here [13].

We have used following code 3.1 to read the TSC.

```
unsigned int hi, lo;
uint64_t cyclecount;

//Read TSC
  __asm__ volatile(
     "cpuid\n\t"
      "rdtsc\n\t"
     "mov %%edx, %0\n\t"
     "mov %%eax, %1\n\t":"=r"(hi),"=r"(lo)::"%rax","%rbx","%rcx","%rdx"
  );
  cyclecount = ( (uint64_t)hi << 32 ) | lo;
```

Listing 3.1: Sample Code to read TSC

3.1.2 Factors Affecting Measurement

No matter, how precise timer we get, there will always be a lot of things that contribute to variation in time measurement. We need to identify all such factors and analyse the impact before we proceed to actual measurement. This section discusses all such factors that affect the time measurement.

3.1.2.1 Power Management

Every Operating System incorporates techniques to optimize the power usage. These management strategies focus on the usage of power only when it is necessary. One such technique is *Dynamic CPU Frequency*. Current CPUs do not always run on a single frequency. It can run on a set of frequencies, which will be decided, based on load by Operating System. For example, when it states that CPU frequency is 2.0 GHz, it only means that the computer can run at a maximum frequency of 2.0 GHz. It usually has a set of frequencies like 1.3 GHz, 1.6 GHz, 1.9 GHz, and 2.0 GHz. These different frequency states are also referred as *performance state* or ”P-state”. The Operating System decides the current frequency, based on management policy, at run time. The TSC is tied to the CPU frequency. Therefore, dynamic CPU frequency causes variation in cycle counts.

Another strategy puts CPUs in different idle states. These states are called ”C-states”. When CPUs are active, it uses power. But, at times it might be idle and will not be executing any code. So, operating system decides to put CPUs in these states to save power. This strategy impacts measurement because it takes time to recover the CPUs from these states. This article [14] gives a good insight of CPU C-States.

3.1.2.2 Interrupt Coalescing

Interrupt Coalescing is the property of Network Interface Cards (NICs). When NIC sends or receives a packet, it informs the operating system using an interrupt. After receiving the interrupt operating system processes the packet. NIC might send or receive multiple packets for a message. In such scenario, it is often an overhead if NIC sends an interrupt per packet. To optimize this process NIC collect a bunch of packets before sending an interrupt to the operating system. This technique is known as *Interrupt Coalescing or Moderation.* With this approach, NIC after sending or receiving one packet either waits for a specific number of packets or certain amount of time, and then it sends an interrupt. This method optimizes the overhead, however, it increases the latency. This is clear that this technique will affect the timing measurements of the messages.

3.1.2.3 Multiprocessing

There are a lot of system processes that run in the background. The schedulers are designed to execute all processes based on their priorities such that no process should feel left out. In such scenario, measuring process might be pre-empted by other high priority processes. This causes variation in time measurements.

3.1.2.4 Multi-Core Processor

Today, it is very common to have multi-core systems. In these systems, one process might get executed on one core at one time and on other cores at other times. Every CPU core has its own TSC. Ideally, TSC of each CPU should be in synchronised. However, practically, they differ in cycle counts. So, if the measuring process reads TSC of one CPU before sending the message and reads TSC of another after receiving the reply, this would give incorrect measure of time.

3.1.2.5 Multi-user Environment

Multi-user environment also introduces lot of jitter in the measurements. Background processes are more in a multi-user environment than single-user and they add to jitter. Moreover, GUI environment would also increase the jitter.

3.1.3 Steps to minimize Measurement Jitter

In previous section, we discussed the factors that interfere with the measurement and introduce jitter. Now, it is the time to overcome those factors in order to minimize the jitter. We would discuss the steps in detail that we used during our experiments. Our discussion would be limited to Linux platform since we performed our experiments on Ubuntu 12.04 LTS. We would definitely give an alternative for other platforms if known.

3.1.3.1 Disable Power Management

Since TSC is connected to CPU frequency, it gives varying results in case of dynamic CPU frequency. So, attacking client or measuring process must be run on a computer with fixed CPU frequency. One should fix the CPU frequency to maximum in order to utilise the highest precision. There are utilities that allow you to change the CPU frequency and policies concerning it in Linux. *cpufrequtils* and *cpupower* are examples of such utilities. We have used *cpufrequtils* for the same. So, the commands mentioned here belong to this utility. One can view the current CPU frequency and policy using command *cpufreq-info*. This command tells you about the available frequency steps and governors. There are five governors - userspace, powersave, conservative, ondemand, and performance. By default, *ondemand* is activated which selects one frequency at run-time from available list based on CPU load. The governor *performance* runs the CPU always on maximum CPU frequency. The governor *userspace* runs the CPU on selected frequency. So, one can fix the CPU frequency and set the governor as *userspace* to run at fixed CPU frequency. Alternatively, one may set the governor as *performance* to do the same. Below are few commands to view and set the policies.

View the frequency settings

```
$ cpufreq-info
```

Command to change the governor to run at maximum frequency

```
$ cpufreq-set -g performance
```

Command to change the frequency of a particular core

```
$ cpufreq-set -c <CPU CORE ID> -f <FREQUENCY>
```

Another important step is to disable the CPU C-states. Recall that CPU C-states will put the CPU in different idle states when CPU is inactive, which causes timing variations due to state switching. Kernel parameter ”idle” controls the C-states in Linux. Set the kernel parameter ”idle=poll” to put the CPU always in active state. The technical white paper [15] from Dell comprehensively describes the usage of C-sates in Linux.

3.1.3.2 Disable Interrupt Coalescing

Interrupt Moderation or Interrupt Coalescing introduces jitter in timing measurement. So, we need to disable it to remove the jitter. A Linux tool called **ethtool** enables you to do that. This is a generic tool to get and modify various parameters of Network Interface Card (NIC) on Linux. Below is the generic format of command to use *ethtool*.

```
$ ethtool [OPTIONS] DEVICENAME
```

DEVICENAME is the name of the Ethernet interface e.g. *eth0*. We need to change the Coalescing Parameters for our purpose. The option "-c" is used to view the coalescing parameters, while "-C" is used to modify it. Relevant coalescing parameters are explained in the table 3.1. Set frame parameters e.g. *rx-frames* to 1 and all others to 0. You must turn off the *adaptive-rx* and *adaptive-tx* parameters to avoid dynamic change in coalesce parameters by the network driver. However, this is not the only way to disable it. Different Ethernet drivers have different methods. You must consult the driver documentation to disable it.

3.1.3.3 Set Process Priority

We need to set the process priority to the highest. This will avoid the pre-emption by other processes. Linux has three types or scheduling classes - SCHED_FIFO, SCHED_RR, and SCHED_NORMAL. SCHED_FIFO and SCHED_RR are for real-time processes. Conventional user space processes belong to SCHED_NORMAL (or SCHED_OTHER) class. The conventional process priority varies from -20 (highest) to 19 (lowest). The priority is also referred as *niceness*. The Linux command **nice** and **renice** can be used to change the priority of a process. The command **nice** is used to start a process with a particular priority, while **renice**

Coalesce Parameters	**Description**
adaptive-rx	Dynamically adjust receive coalesce parameters based on network load
rx-frames	Number of packets to be received before generating an interrupt
rx-usecs	Wait specified microseconds after receiving a packet before generating an interrupt
rx-frames-irq	Corresponding delay in updating the status when interrupt is disabled
rx-usecs-irq	Corresponding delay in updating the status when interrupt is disabled
adaptive-tx	Dynamically adjust transmit coalesce parameters based on network load
tx-frames	Number of packets to be sent before generating an interrupt
tx-usecs	Wait specified microseconds after transmitting a packet before generating an interrupt
tx-frames-irq	Corresponding delay in updating the status when interrupt is disabled
tx-usecs-irq	Corresponding delay in updating the status when interrupt is disabled

Table 3.1: Description of Coalesce Parameters

is used to change the priority of an existing process. Usage of these commands is given in 3.2.

```
$ nice -n <priority> your_command
$ renice -n <priority> -p <pid>
```

Listing 3.2: Usage of nice and renice

The above method changes the priority statically. We can use Linux APIs **getpriority()** and **setpriority()** to dynamically retrieve and modify the process priority at run-time. In our implementation, we have used these APIs to achieve the goal. Below is the C code 3.3 that has been used in our implementation to set the highest priority of the attacking process at run-time.

```c
void setProcessPriority()
{
//Get Process ID
int pid = getpid();

int which = PRIO_PROCESS;

//Get Process Priority before the change and Print it
fprintf(stdout,"\nPriority Before : %d", getpriority( which , pid ) );

//Set Process Priority to highest
if( setpriority( which , pid , PRIO_MIN ) < 0 )
    fprintf(stderr,"\nPriority set error");

//Print Process Priority after the change
fprintf(stdout,"\nPriority After : %d\n", getpriority( which , pid ) );
}
```

Listing 3.3: Sample Code to set Priority

3.1.3.4 Set Processor Affinity

In multi-core systems, different cores can execute one process at different times during its execution. This feature introduces jitter due to mismatch in TSC counts of different cores. We need to execute our process only on one core in order to avoid this problem. The ability of Operating System to bind a process to certain processors is called Processor Affinity. The term *Processor Affinity* refers to a value, which indicates that the process can be executed on specified number of processors or cores. Operating System chooses one of the available CPU by looking at the Processor Affinity at the time of execution.

CPU Affinity or Processor Affinity is a bitmask of n-bits, where n is the bitsize of Operating System. For example, on a 32-bit OS, bitmask would be of 32-bits. These bits (from right to left) represent individual processors. Bit value 1 means that the process can be executed on that processor, while value 0 signifies otherwise. By default, a process can be executed by all available processors. For example, let us assume a 32-bit computer with four cores or processors. The Processor Affinity of a process on such system would look like below 3.4.

```
00000000000000000000000000001111
This states that the process is bound to all four processors.
```

Listing 3.4: Processor Affinity of a Process

The Linux command **taskset** can be used to view and modify CPU affinity of a process. You can start a command with certain CPU affinity or you can modify it for a running process.
Below 3.5 is the example to execute a command with specified CPU affinity:

```
 $ taskset mask <YOUR_COMMAND>
 where mask is integer equivalent of the bitmask
```

Listing 3.5: Execute a command with fixed processor affinity

Retrieve the CPU affinity of a running task as follows:

```
 $ taskset -p <pid>
```

Set the CPU affinity of a running task as follows:

```
  $ taskset -p mask <pid>
```

Linux also provides two system calls *sched_getaffinity()* and *sched_setaffinity()* to get and set the CPU affinity respectively at run-time. I have used these system calls to do the same. Below is the C code 3.6 that has been used in our implementation to set the CPU affinity of the attacking process to first core at run-time.

```c
void setCpuAffinity(){
  //Set mask to 1 for first core
  unsigned long mask = 1;
  unsigned int len = sizeof(mask);

  //Set pid for current process
  int pid = 0;

  if( sched_setaffinity( pid , len , &mask ) < 0 ){
      fprintf(stderr,"\nError in setting Processor Affinity");
      return;
    }
}
```

Listing 3.6: Sample Code to set CPU Affinity

One can also bind the attacking process to one core and all other processes to other cores. You can dedicate the CPU to your attacking process in this manner.

3.1.3.5 Stop Background Processes

Application load may cause additional jitter. System background and scheduled processes interfere with timing measurements. So, it is advisable to stop such processes. On Linux, stop the *cron* scheduled processes. You can view the scheduled jobs using the bash command `$ crontab -l`. It is also advisable not to use GUI mode. One can use ”single user mode” in Linux by booting it with boot parameter ”single”.

3.1.3.6 Use Wired Network

Wireless network could also introduce noise because it is not as fast as wired network. So, using wired network would improve the measurements.

3.1.3.7 Cache Warm Up

There is a peculiar property of RDTSC instruction that it needs to be executed 3-4 times before the actual measurement starts. This behavior is attributed to cache warm up. We call the RDTSC instructions 4 times at the beginning of the attack to warm up the cache. It is also advisable to throw away initial measurements (around 1000) because they have more noise due to cache warm up.

3.1.3.8 Alternate Measurement

When we record the decryption time for two different messages, say A and B, we take n samples first for message A and then for message B. After that we filter out one value from n-samples for both messages and compare them. This measurement approach is shown below.

Measurement for message A : $[A_1, A_2, A_3, ..., A_n]$.
Measurement for message B : $[B_1, B_2, B_3, ..., B_n]$.

In this approach, the time at which the measurement is done differs for both datasets. Consequently, both datasets have different jitter. As a result, this approach does not give proper results. We need an approach in which both datasets are impacted by approximately same jitter. The measurement is done alternately in order to achieve the goal. Both datasets have approximately same jitter in this approach. The new approach is shown below.

Measurement for message A and B : $[A_1, B_1, A_2, B_2, A_3, B_3, ..., A_n, B_n]$.

After the measurement, we separate out the samples for A and B and carry out the usual approach of comparison.

3.2 Filtering

Ideally, processing time of a message should be a single value. However, you get different values each time you repeat the process for the same message. So, you get multiple values for the same message in real life. Thus, we must take out a single value from multiple values as the effective processing time for the message. So, we must have a filtering technique in place.

Before we proceed with selection of appropriate filter, let us have a look at the distribution of the time measurements.

3.2.1 Time Measurement Distribution

Time Measurements vary a lot in real environment. It is important to study the distribution before choosing a filter. Many studies have been done for this and it has been shown that it is not normal-distributed. Rather, it comes out to be highly asymmetric. In most of the cases, it is right-skewed. The distribution looks like that in the figure 3.1.

3.2.2 Filter

A naive filter could be *Mean*, *Median*, or *Mode*. However, these filters are suitable when data is normally distributed. Since the data is not normal-distributed, it would not work well for our case. Moreover, "Mode" could be more than one value in an asymmetric distribution, which makes it inappropriate to use. Among these filters, Median could be used, but it would still have jitters. Actually, initial

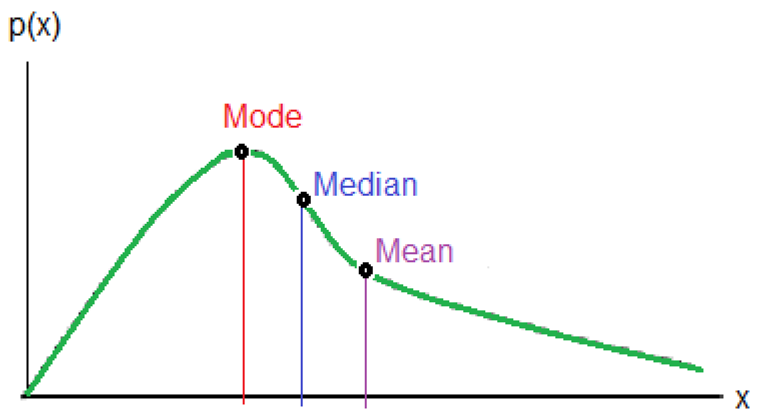


FIGURE 3.1: Asymmetric Distribution

researchers of Remote Timing Attacks like Paul Kocher and Brumley & Boneh have used Median.

Our data is asymmetric and we have filters that do not perform well for such distribution. Now the question arises, how can we perform the statistical analysis? These questions motivated the work of Crosby et al [16]. Their work shows how to measure processing time over network and perform the statistical analysis on the captured data. In their research, they compared various filters and pointed out that **Low Percentile Filters** are best suited for asymmetric distributions.

3.2.2.1 Low Percentile Filter

Percentile or Quantile simply refers to a value below that a certain percentage of the dataset lies. If we say a value as *n-percentile*, this means that there are *n-percent* values that are less than this value. Median is 50-percentile. So, 50 percent values lie below the Median. In Low Percentile Filters, we consider low percentile values, typically 3-10 percentile.

Crosby et al [16] came up with the formula 3.1 that breaks up the actual response time over network into processing time, propagation time, and jitter. Our target is to effectively measure the processing time by reducing the other elements. However, we could not get the exact processing time due to add ups. We must minimize the other elements and then select a value that has best correlation with the processing time.

$$ResponseTime = ProcessingTime + PropagationTime + Jitter \tag{3.1}$$

Intuitively, one can suggest that 0-percentile value (Minimum value in the dataset) should have best correlation and it would be the best filter. On the contrary, the experiments by Crosby et al show that 0-percentile filters have more noise and they do not perform as expected. According to their work, 3-10 percentile filters turn out to be good filters and they were able to differentiate between timings in the range of nanoseconds using these filters. After analysing our dataset, we chose 3-percentile value as the filter since it has less variation. We could differentiate well between distributions of two different messages using it.

Chapter 4

Related Work

Remote Timing Attacks (Side Channel Attacks, in general) are not new type of attacks. These types of attacks have been demonstrated by researchers long ago on hardware. There was a time when people believed that timing attacks are impossible to carry out on software because of non-deterministic behavior and uncontrollable environment. On the contrary, Researchers surprised the world by actually performing such attacks. They showed that it is possible and they are big threat to security because developers tend to overlook such issues. We will discuss the work, which is the basis of our implementation and few others related works in this section.

4.1 Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems - Paul C. Kocher

Paul C. Kocher can be safely assumed to be the pioneer in demonstrating the timing attack on cryptographic implementation. In 1996, he demonstrated a timing attack [17] on implementation of cryptographic algorithm such as Diffie-Hellman, RSA, DSS, and others. He showed that it is possible to find fixed Diffie-Hellman exponents, factor RSA keys, and break other systems, if an attacker can precisely measure the cryptographic operations.

In RSA, receiver must solve the decryption equation $M = C^d \bmod N$, where d is private key, to get the message M. There are several implementations of RSA, one of them uses pure *square and multiply* technique to solve the modular exponentiation of the form $R = y^x \bmod n$, where x is of w bits. *Square and multiply* technique is an optimized solution to modular exponentiation. This algorithm performs "squaring" in each iteration and "multiplication" depends on the bits of the private key x. The *square and multiply* algorithm has been outlined it algorithm 7. It has been taken from [17, p. 2].

Algorithm 7 Calculate $R = y^x \bmod n$

```
1:  $s_0 = 1$
2:  **for** $i = 0$ to $w - 1$ **do**
3:      **if** (bit $i$ of $x$) $= 1$ **then**
4:          $R_i \leftarrow (s_i \cdot y) \bmod n$
5:      **else**
6:          $R_i \leftarrow s_i$
7:      **end if**
8:      $s_{i+1} \leftarrow {R_i}^2 \bmod n$
9:  **end for**
10: **return** $(R_{w-1})$
```

It is clear from the algorithm that the "multiplication" step depends on the bit of the private key x that introduces the timing difference. This step leaks out the information whether particular bit of x is 1 or 0. Paul Kocher leveraged this vulnerability to recover the private key used in RSA. His Attack failed on RSA implementations that use Chinese Remainder Theorem (CRT) to solve the decryption equation.

He found similar timing differences in other cryptographic algorithms.

4.2 A Timing Attack against RSA with the Chinese Remainder Theorem - Werner Schindler

The contribution of Werner Schindler towards cryptanalysis of RSA has been great. Kocher's attack failed on RSA implementation, which uses CRT. It was believed that RSA-CRT is immune to timing attacks until Werner Schindler showed in year 2000 that RSA-CRT also leaks out information through timing side channel. His great work has been the basis of all other timing attacks on RSA. All researchers have utilized his findings to devise attack algorithms on RSA.

His work finds out the timing vulnerability on RSA implementation, which uses CRT with Montgomery's algorithm. He pointed out that the "extra reduction" during Montgomery multiplication depends on the input cipher text and modulus. Let us recall the Montgomery Multiplication and its use in *square and multiply* algorithm. Refer to algorithm 5 in Chapter 1 for Montgomery Multiplication. The exponentiation algorithm 8 uses *square and multiply* with Montgomery's algorithm to calculate $R = y^x \bmod n$, where $(x_w x_{w-1} x_{w-2} ... x_0)_2$ is binary equivalent of x and $x_w = 1$.

Algorithm 8 Calculate $R = y^x \bmod n$

```
1: $temp = \psi(y)$
2: for $i = w - 1$ down to 0 do
3:     $temp = MM(temp, temp)$
4:     if (bit $i$ of $x$) = 1 then
5:         $temp = MM(temp, \psi(y))$
6:     end if
7: end for
8: return $\psi_*(temp)$
```

$\psi(y)$ converts y into Montgomery form and $\psi_*(y)$ converts back to normal form. The work of Schindler showed that the number of "extra reduction" steps in the above exponentiation algorithm depends on base y, or more precisely on $\psi(y)$. He showed that the probability of "extra reduction" is $\frac{y \bmod n}{2R}$, where R is some power of 2 such that $R > n$ and $gcd(R, n) = 1$. By precisely measuring the time, one can figure out how close is y to n. He did not perform the attack on actual RSA implementation; rather he implemented above exponentiation algorithm in software and demonstrated the attack. However, his work paved the path for other timing attacks because most implementations use CRT. Schindler attack failed when RSA-CRT uses Sliding Window Exponentiation (SWE).

4.3 Remote Timing Attacks are Practical - Brumley and Boneh

Until 2003, timing attacks have been done with respect to hardware security tokens such as smart cards. Moreover, it was believed that these attacks cannot be successfully mounted on software because multiprocessing on computers makes it impossible to measure the cryptographic operations. In the midst of these believes, Brumley and Boneh successfully retrieved the primary key of a web server by performing the timing attack [12]. His attack stunned the world and is considered to be the most sophisticated timing attack in the history. He mounted the attack against a web server, which used OpenSSL to provide SSL security. The attack was done when RSA is used as key exchange cipher. We will refer this attack as BB Attack moving onwards.

OpenSSL uses very optimized implementation of RSA. It uses Chinese Remainder Theorem, Sliding Window Exponentiation, Montgomery algorithm, and Karatsuba Multiplication in its RSA implementation. These optimizations make so far known timing attacks including the Schindler's attack to fail in practice. Brumley and Boneh found two data dependencies [12, p .4] in RSA decryption algorithm - (1) Number of "extra reductions" in Montgomery Multiplication (Schindler's observation[11]), and (2) timing difference due to choice of two multiplication routines - Karatsuba and Normal. The effects of both counteract each other. However, they figured out that one of them will always dominate the other and the exact environment will determine this dominance.

Using the timing attack, they factorized the RSA modulus N of 1024-bit, where $N = pq$ and $q < p$. They recovered the smaller prime factor q(512-bits) by the timing attack. They sent approximations of q as messages to the web server and noted the decryption time to decide a particular bit of q. They retrieve q bit by bit starting from the Most Significant Bit (MSB). This attack can be viewed as a binary search for q. The binary equivalent of q is $(q_{511}, q_{510}, q_{509}, ..., q_0)_2$. For OpenSSL, q_{511} =1.

There are two phases of the attack - (1) Guess top few bits (typically 2-3), and (2) Retrieve the rest of the bits one by one. In phase 1, all combinations of top few bits (typically 2-3 bits) are timed. These timings will have two peaks when plotted which correspond to q and p. The first peak will be for smaller prime factor q. After this phase, we know top few bits of q. Once we know the top few bits, we proceed as explained in algorithm 9 to retrieve further bits one by one.

The algorithm assumes that the attacker has already recovered $i - 1$ bits of q.

The value of **"large"** and **"small"** depends on the exact environment and is decided by looking at the previous values.

We need to recover only half of the bits (256 in this case) of q by timing attack. To retrieve the full 512- bits, we use Coppersmith's algorithm [18].

To demonstrate the attack, Brumley and Boneh performed several experiments. They implemented a server, which accepted a binary string as message and decrypted the message using OpenSSL. Server sent 0 to client to notify the end of decryption. They implemented a client, which followed the algorithm 9 to recover the prime factor q. They also showed the real web server attack using Apache Web Server and mod_ssl.

Attack Parameters

Brumley and Boneh used two attack parameters - Sample Size s, and Neighbourhood Size n. Decryption time for the same message varies in practice. To overcome this they repeatedly sent the same message s times. They took median of these s values as effective decryption time.

Algorithm 9 Retrieve i_{th} bit of q

```
1: Let $g$ be a number with same $i-1$ bits as $q$. Remaining bits of $g$ are 0. Let
   $g_{hi}$ be equal to $g$ with $i_{th}$ bit as 1. So,
   $g = (q_1, q_2, q_3, q_{i-1}, 0, 0, ..., 0)$
   $g_{hi} = (q_1, q_2, q_3, q_{i-1}, 1, 0, ..., 0)$
   The idea is, if bit $i$ of $q$ is 0, then $g < q < g_{hi}$
   otherwise $g < g_{hi} < q$
2: Compute $u_g = g \cdot R^{-1} \bmod N$ and $u_{ghi} = g_{hi} \cdot R^{-1} \bmod N$. This step con-
   verts $g$ and $g_{hi}$ in inverse Montgomery form. This is required because server
   will convert $u_g$ and $u_{ghi}$ into Montgomery form before exponentiation during
   decryption to get $u_g \cdot R = g$ and $u_{ghi} \cdot R = g_{hi}$ .
3: Send $u_g$ and $u_{ghi}$ to the server as messages to decrypt.
4: Measure the decryption time for $u_g$ and $u_{ghi}$. Let $t_1 = DecryptionTime(u_g)$
   and $t_2 = DecryptionTime(u_{ghi})$.
5: Calculate the difference, $t_{delta} =| t_1 - t_2 |$
6: if $t_{delta}$ is "large" then
7:     $g < q < g_{hi}$. So, bit $i$ of $q$ is 0
8: end if
9: if $t_{delta}$ is "small" then
10:    $g < g_{hi} < q$. So, bit $i$ of $q$ is 1
11: end if
```

OpenSSL also uses Sliding Window Exponentiation (SWE). This decreases the gap between **"large"** and **"small"** values. In order to increase the gap Brumley and Boneh used neighbourhood values. With this approach they sent g, $g+1$, $g+2$, ..., $g+n$, where n is neighbourhood size, to server when calculating the decryption time for g. After that they sum all these values to calculate effective decryption time for g and similarly for g_{hi}. The difference is then calculated and decision is made based on **"large"** and **"small"** values.

4.4 Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations - Aciicmez, Schindler, and Koc

BB Attack [12] is the most classic timing attack on RSA so far. Other researchers have either implemented the same attack or did some improvements on it. One of the improvements [19] is presented by Aciicmez, Schindler, and Koc. We will refer this as *Schindler attack* going forward in this document.

While calculating $g^d \bmod q$ with SWE, the algorithm first prepares a table with odd powers of g including g^2. During actual exponentiation intermediate values are multiplied by one of these table entries. BB attack take advantage of the multiplications by g. Schindler et al observed that there are more multiplications by g^2(approx. 15) than g(approx. 6) during the exponentiation[19, p 4]. We could increase the efficiency of BB attack by a factor of approximately 6, if we could take advantage of multiplications by g^2. Schindler et al did exactly the same. The modified algorithm has been outlined in algorithm 10.

Schindler et al introduced one more improvement in decision strategy. BB attack calculated the sum of all neighbourhood values and then took the difference. However, Schindler et al calculated the difference for individual neighbourhood values and counted positive and negative differences. After that they calculated the ratio of maximum of these positive and negative differences and the neighbourhood size. They observed that if the bit is 1, the numbers of positive and negative differences are approximately equal giving a ratio close to 0.5. When the bit is 0, either of them outnumbered the other giving a ratio of more than 0.5. This decision strategy enlarges the 0-1 gap and allows to bring the neighbourhood size parameter down. This strategy results in fewer queries for a successful attack than that of BB attack.

BB attack strategy :

$$\Delta = \left| \sum_{i=0}^{n-1} DecryptTime((g+i)\cdot R^{-1} \bmod N) - \sum_{i=0}^{n-1} DecryptTime((g_{hi}+i)\cdot R^{-1} \bmod N) \right| \tag{4.1}$$

Decision of bit depends on Δ being "small" or "large".

Schindler attack strategy :

$$\Delta_i = DecryptTime((\sqrt{g}+i)\cdot R^{-1} \bmod N) - DecryptTime((\sqrt{g_{hi}}+i)\cdot R^{-1} \bmod N)$$
$$for \quad i = 0, 1, 2, ..., n-1 \tag{4.2}$$

$$\Delta_{ratio} = \frac{\max(\#(\Delta_i < 0), \#(\Delta_i > 0))}{n} \tag{4.3}$$

Decision of bit depends on Δ_{ratio} being "small" or "large".

Algorithm 10 Retrieve i_{th} bit of q

```
1: Let $g$ be a number with same $i-1$ bits as $q$. Remaining bits of $g$ are 0. Let
   $g_{hi}$ be equal to $g$ with $i_{th}$ bit as 1. So,
   $g = (q_1, q_2, q_3, q_{i-1}, 0, 0, ..., 0)$
   $g_{hi} = (q_1, q_2, q_3, q_{i-1}, 1, 0, ..., 0)$
   The idea is, if bit $i$ of $q$ is 0, then $g < q < g_{hi}$
   otherwise $g < g_{hi} < q$
2: Compute $h = \lfloor\sqrt{g}\rfloor$ and $h_{hi} = \lfloor\sqrt{g_{hi}}\rfloor$
3: Compute $u_h = h \cdot {R_2}^{-1} \bmod N$ and $u_{hi} = h_{hi} \cdot {R_2}^{-1} \bmod N$, where $R_2 = \sqrt{R}$
   This step converts $h$ and $h_{hi}$ in inverse Montgomery form.
   Server calculates first power, $y_1 = u_h R = hR_2$ (similarly for $u_{hi}$)
   and second power, $y_2 = hR_2 \cdot hR_2 \cdot R^{-1} = g$ and
   $y_2 = h_{hi}R_2 \cdot h_{hi}R_2 \cdot R^{-1} = g_{hi}$. before exponentiation during decryption.
4: Send $u_h$ and $u_{hi}$ to the server as messages to decrypt.
5: Measure the decryption time for $u_h$ and $u_{hi}$. Let $t_1 = DecryptionTime(u_h)$
   and $t_2 = DecryptionTime(u_{hi})$.
6: Calculate the difference, $t_{delta} =\mid t_1 - t_2 \mid$
7: if $t_{delta}$ is "large" then
8:     $g < q < g_{hi}$. So, bit $i$ of $q$ is 0
9: end if
10: if $t_{delta}$ is "small" then
11:     $g < g_{hi} < q$. So, bit $i$ of $q$ is 1
12: end if
```

Chapter 5

Implementation

This chapter deals with the actual setup and implementation to perform the attack.

5.1 Machine Configuration

All experiments have been done on 64-bit Ubuntu operating system with 2GB of RAM. It has one Intel dual core processor. The attacks on localhost have been performed against OpenSSL version 0.9.7, which does not enable blinding by default. Latest version v1.0.1i of OpenSSL, which enables blinding by default has been used for attack over switched LAN. Blinding was disabled to carry out the attack. We used C language for coding and gcc v4.6.3 as compiler. Keys have been randomly generated using OpenSSL. The table 5.1 lists down the used configuration.

Parameters	**Value**
Operating System	Ubuntu 12.04 LTS
Kernel	3.11.0-15-generic
Processor	Intel® Core 2 CPU T7200 2.00 GHz
Memory	2.00 GB RAM
Compiler	gcc V4.6.3
Crypto Library	OpenSSL-0.9.7 , OpenSSL-1.0.1i

Table 5.1: Machine Configuration for the timing attack

5.2 Jitter Minimisation

To measure the decryption time precisely we followed few steps mentioned here. These steps are necessary to avoid the external interference at the time of measurement. This section simply lists down the necessary steps taken. Please refer section ”Steps to minimize Measurement Jitter” for more details like how things have been done.

Necessary Steps :

- Set CPU frequency to the maximum limit.
- Disable CPU C-states.
- Bind attacking client to one core and all other processes to other core.
- Set the priority of attack client to highest.
- Stop unnecessary user tasks.
- Disable all scheduled tasks.
- Use assembly instruction RDTSC along with CPUID to read TSC.
- Cache warm up before actual measurement.

- Alternate measurement strategy.
- Use Wired Network for LAN attack.
- Disable Interrupt Coalescing or Moderation for LAN attack.

5.3 Network Model

We have performed our experiments on two network models - (1) Interprocess and (2) Switched LAN. In Interprocess attack, server and client were running on the same machine.

In switched LAN, server and client were running on different machines that are separated by one switch. The connection to the switch of both machines were wired. Figure 5.1 depicts the switched LAN model.

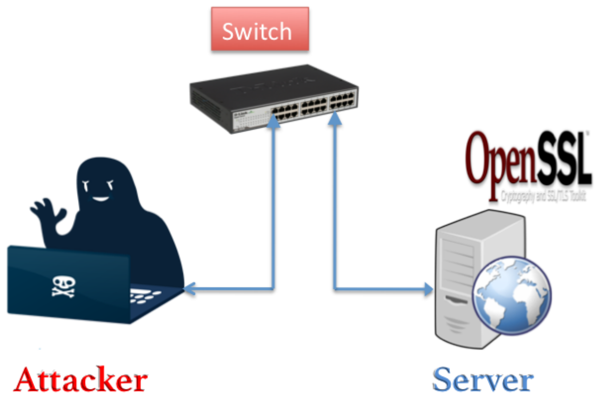


Figure 5.1: Network Model : Switched LAN

5.4 Timing Attack on RSA

In this section, the setup is discussed with respect to server-client perspective. We discuss how OpenSSL is compiled, and server and client are coded.

5.4.1 OpenSSL Compilation

We downloaded the source code of OpenSSL library from the OpenSSL homepage [3]. We configured the library for the machine. The library is compiled using gcc v4.6.3. We used the default optimizations.

5.4.2 Decryption Server

We implemented a simple decryption server. After normal TCP handshake with the client, the server accepts a string and decrypts it using the OpenSSL library. The server replies with -1 to notify the end of decryption. Randomly generated key is used to perform the decryption. Server ends the connection once it receives a message from client to end the connection. The server takes port number as input in order to start the server on that port.

5.4.3 Attack Client

Attack client plays a crucial role in the attack. It incorporates all the attack logic and decision strategy. As a black box, it sends messages to the server to decrypt it and records the decryption time taken by the server. It decides the bits based on the decryption time. It also performs certain steps to overcome jitter effect dynamically. The client is coded in C and compiled using gcc v4.6.3. It uses the same OpenSSL library as the server.

The client also has the same key used by the server in order to compare the guesses and make corrections for incorrect guesses. Attack client assumes that we know initial few bits of prime factor q and, so, it copies those initial bits to recovered list. The search starts from this point. It does not use the key for any other purpose.

The client also performs few steps at runtime in order to reduce jitter. Firstly, it binds the process to a single core (first core) and then, it sets the priority to the highest. It also does four dummy calls to RDTSC instructions for cache warm up. It also calculates the overhead in calling RDTSC instruction this way. This overhead is subtracted from the cycle count to compensate for the delay in calling RDTSC instruction.

The attack client is parameterized. It accepts certain inputs based on which it adjusts the attack. These inputs include $ATTACKTYPE$, $STARTBIT$, $ENDBIT$, $SAMPLES$, $NBSIZE$, $DELTABITGAP$, $DELTARATIO$, and $PERCENTILE$. The type of attack means BB attack or Schindler attack is decided by $ATTACKTYPE$ flag. The attack starts to guess the bits from $STARTBIT$ and goes till $ENDBIT$. $NBSIZE$ refers to the neighbourhood size. The bitgap that should be considered "small" is referred by $DELTABITGAP$ and similarly $DELTARATIO$ for ratio.

The client is adaptable to both BB attack and Schindler attack. It takes a user input to decide which attack algorithm to follow. Please refer to algorithm 9 for BB attack and algorithm 10 for Schindler attack. We have also implemented both decision strategies. The pseudo code of the attack client has been outlined in algorithm 11. We have mentioned only BB attack strategy in the algorithm to keep things simple.

We count the number of correct guesses by comparing the guess with original bit. This allows us to calculate the success rate. Success rate signifies the percentage of bits that are correctly guessed. Equation 5.1 gives the formula to calculate $SuccessRate$.

$$SuccessRate = \frac{NumberofCorrectGuesses}{ENDBIT - STARTBIT + 1} * 100 \tag{5.1}$$

Algorithm 11 Pseudo Code for Attack Client

```
1: Parse the attack parameters & get prime factor $q$ and Modulus $N$.
2: Bind the process to first core & Set the highest process priority.
3: Warm up the cache by making calls to RDTSC. Also, Calculate the $OVERHEAD$.
4: Generate GUESS $g$ of 512 bits and set all bits to 0 & Copy initial known bits to GUESS $g$.
5: for bit $i = STARTBIT$ to $ENDBIT$ do
6:     Set $g_{hi} = g$ & Set $i_{th}$ bit of $g_{hi}$ to 1.
7:     if (ATTACKTYPE == SCHINDLER_ATTACK) then
8:         $g \leftarrow \lfloor \sqrt{g} \rfloor$
9:         $g_{hi} \leftarrow \lfloor \sqrt{g_{hi}} \rfloor$
10:        Compute $R = 2^{256}$ and its inverse $R^{-1}$
11:    else
12:        Compute $R = 2^{512}$ and its inverse $R^{-1}$
13:    end if
14:    for $index = 0$ to $NBSIZE - 1$ do
15:        Compute $u_g = g \cdot R^{-1} \bmod N$ and $u_{ghi} = g_{hi} \cdot R^{-1} \bmod N$.
16:        for ($count = 1$ to $SAMPLES$) do
17:            Send $u_g$ and $u_{ghi}$ to server and record the decryption time.
18:        end for
19:        Compute $percentile$ to filter out effective decryption time $t_g[index]$ and $t_{ghi}[index]$ for $u_g$ and $u_{ghi}$ respectively.
20:        Set $g = g + 1$ and $g_{hi} = g_{hi} + 1$.
21:    end for
22:    Compute $T_g = \sum_{k=0}^{NBSIZE-1} t_g[k]$, and $T_{ghi} = \sum_{k=0}^{NBSIZE-1} t_{ghi}[k]$
23:    Compute $bitgap =\mid T_g - T_{ghi} \mid$
24:    if (bitgap is "small") then
25:        Guess $i_{th}$ bit as 1
26:    else
27:        Guess $i_{th}$ bit as 0
28:    end if
29:    Set all bits of $g$ to 0 and Copy $i$ bits of $q$ to $g$.
30: end for
```

5.4.4 Attack Parameters

Similar to BB attack we have two parameters for the attack - (1) Sample Size s, and (2) Neighbourhood Size n.

Sample Size : We send the same message multiple times to cancel out the jitter. Sample Size s denotes that how many times we will send the same message to server. The variation in the percentile will become lower as we increase the sample size. During our experiments we noticed that sample size of 7 was enough to optimize the variation and carry out the attack successfully.

Neighbourhood Size : We have used neighbourhood values to overcome the effect of SWE. Neighbourhood Size n denotes the same. This means that we send $g, g+1, g+2, ..., g+n-1$ as messages while calculating decryption time for g. We require different neighbourhood size to recover different parts of bits. To keep things simple we fix the neighbourhood size for all bits. Neighbourhood size of 800 is enough to recover most of the bits in interprocess attack. We must increase the Neighbourhood size in order to gain more success.

The number of queries needed to factorize N is decided by these two factors. So, the number of queries is calculated as explained in equation 5.2. Please note that multiplication factor of 2 is because we have to send two messages g and g_{hi} for each bit. We need to recover only half of the bits of prime factor by timing attack. So, we need log of $\frac{N}{4}$. Based on the equation 5.2, we need $2*25*400*256 = 5,120,000$ queries, considering sample size 25 and neighbourhood size 400, to factorize N of 1024-bits.

$$NumberOfQueries = 2 * n * s * \log_2 \frac{N}{4}, \quad \text{where } N \text{ is the Modulus} \tag{5.2}$$

5.4.5 Decision Strategy

We have implemented both decision strategies - BB attack and Schindler attack. We calculate decryption time for individual neighbours for g and g_{hi}. For BB attack strategy, we compute sum of all neighbourhood values and then calculate the absolute difference to get the bitgap. If the resulted bitgap is "small" we guess the bit as 1, otherwise 0.

For Schindler attack strategy, we compute the difference of corresponding neighbourhood values of g and g_{hi}. We count the positive and negative differences. We calculate the ratio as $\frac{\max(positivecount, negativecount)}{neighbourhoodsize}$. If the ratio is "small" we guess the bit as 1, otherwise 0. Please refer to equation 4.1 for BB decision strategy and equation 4.2 for Schindler decision strategy respectively.

5.4.6 Attack in Two Phases

The time attack works in two phases - (1) Find Peak, and (2) Bit-by-Bit Recovery. With the help of first phase, we find initial few bits of prime factors. Once we have the initial bits, we find the other bits one by one in second phase.

5.4.6.1 Find Peak

First step is to find initial few bits (typically 2-3 bits) of prime factor q. This step is required so that we can differentiate between prime factor p and q by looking at initial bits and, then, recover the bits of q by actual attack. During this step, all combinations of initial few bits are timed. Once we plot these decryption times, we get two peaks. First peak corresponds to smaller prime factor q and second peak corresponds to p. This is always true in case of OpenSSL because $q < p$.

We transformed the attack client into a peak finding client by making few changes to it. So, we have two attack clients - one finds the peak and the other recovers the bits one by one. The algorithm 12 explains the pseudo code of peak finder.

Algorithm 12 Pseudo Code for Peak Finder

```
1: Read $numOfBits$, prime factor $q$, and Modulus $N$.
2: Bind the process to first core & Set the highest process priority.
3: Warm up the cache by making calls to RDTSC. Also, Calculate the $OVERHEAD$.
4: Calculate $numOfCombinations = 2^{numOfBits}$ & Generate all combinations.
5: Generate GUESS $g$ of 512 bits as many $numOfCombinations$ and set all bits of $g$ to 0 for all.
6: Set first bit as 1 for all $g$.
7: for (each bit combination) do
8:     Modify corresponding $g$ to reflect the bit combination.
9:     if (ATTACKTYPE == SCHINDLER_ATTACK) then
10:        $g \leftarrow \lfloor\sqrt{g}\rfloor$
11:    end if
12: end for
13: if (ATTACKTYPE == SCHINDLER_ATTACK) then
14:     Compute $R = 2^{256}$ and its inverse $R^{-1}$
15: else
16:     Compute $R = 2^{512}$ and its inverse $R^{-1}$
17: end if
18: for $index = 0$ to $NBSIZE - 1$ do
19:     for each $g$ do
20:         Compute $u_g = g \cdot R^{-1} \bmod N$.
21:     end for
22:     for ($count = 1$ to $SAMPLES$) do
23:         Send each $u_g$ to server and record the decryption time.
24:     end for
25:     Compute $percentile$ to filter out effective decryption time $t_g[index]$ for each $u_g$.
26:     Set $g = g + 1$.
27: end for
28: Compute $T_g = \sum_{k=0}^{NBSIZE-1} t_g[k]$, for each $g$.
29: Write all bits combinations and corresponding $T_g$ to an outputfile.
```

The peak finder generates decryption time for all bit combinations. Later, we use R script to plot the time on a graph to find the peaks. We take the first peak as the initial bits of q.

5.4.6.2 Bit-by-Bit Recovery

This phase recovers the bit of prime factor q one by one starting from MSB to LSB. We recover only half of the bits of q and the other half is assumed to be recovered using Coppersmith Algorithm [18].

5.5 Attack to Find Existing User

Apart from the RSA attack, we also performed simple timing attack to find whether a user exists in a system or not. This was done in order to get the insight of timing attack. In this section we will discuss about the server-client implementation and attack strategy.

5.5.1 Vulnerable Login Server

To set up the environment, a simple login server is implemented. This server accepts a pair of username and password as input and responds if the login is successful or not. If the login is successful, it returns ”success”, otherwise ”username or password incorrect”. The pseudo code of the login server is given in algorithm 13.

The same algorithm has been further explained with the help of the flow diagram 5.2.

It is clear from the algorithm 13 and figure 5.2 that server takes more time to process a valid user than an invalid user. If we can measure precisely the processing time, we can find out whether a user exists in the system or not. There is no locking mechanism enabled in the system on failed login attempts.

Algorithm 13 Pseudo Code for Login Server

```
1: Accept username and password from client
2: if (username exists ) then
3:     if (username is locked) then
4:         return ”username or password incorrect”
5:     else if (username is expired) then
6:         return ”username or password incorrect”
7:     else if (password is correct) then
8:         return ”success”
9:     else
10:        return ”username or password incorrect”
11:    end if
12: else
13:    return ”username or password incorrect”
14: end if
```

5.5.2 Attack Client

The attack client accepts a ”username” to verify whether it is a valid user in the login server. It sends many login attempts to the login server using a dummy password for the user. It is assumed that there is no locking mechanism enabled in case of several failed login attempts. It measures the response time from server and compare to guess the user. We need either a valid user's response time or an invalid user's response time for comparison. It is unlikely that the attacker knows a valid user in the system. However, he can safely assume a non-existing user. We take a dummy user such as ”111111” and assume that it does not exist in the system. The attack client measures the response time for this user. This processing time serves as a base processing time for comparison. If the processing time for the ”username” is ”significantly” more than that of ”111111”, the ”username” is a valid user. What difference should be considered as ”significant”? We can observe this value by comparing the processing times of two dummy users.

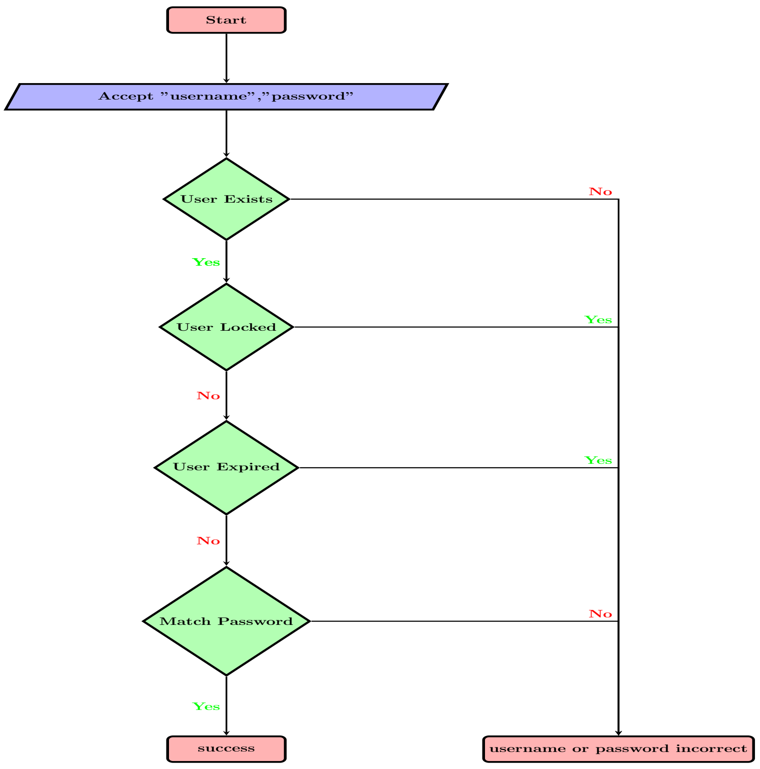


FIGURE 5.2: Flow Diagram of Login Server

The attack client has been implemented keeping the above ideas in mind. The pseudo code is explained in algorithm 14.

Algorithm 14 Pseudo Code to guess a User

```
1: Accept the "username".
2: Generate a dummy user "111111" and a dummy password "password".
3: Bind the process to first core.
4: Set the highest priority.
5: Warm up the cache by making calls to RDTSC. Also, Calculate the
   $OVERHEAD$.
6: for ($count = 1$ to $SAMPLES$) do
7:     Send ("username","password") to server and record the response time.
8:     Send ("111111","password") to server and record the response time.
9: end for
10: Compute percentile to filter out effective response time $t_1$ and $t_2$ for
    "username" and "111111" respectively.
11: Compute $\Delta = t_1 - t_2$.
12: if ($\Delta$ is positive and "significant") then
13:     return "username" is valid
14: else
15:     return "username" is invalid
16: end if
```

Chapter 6

Results

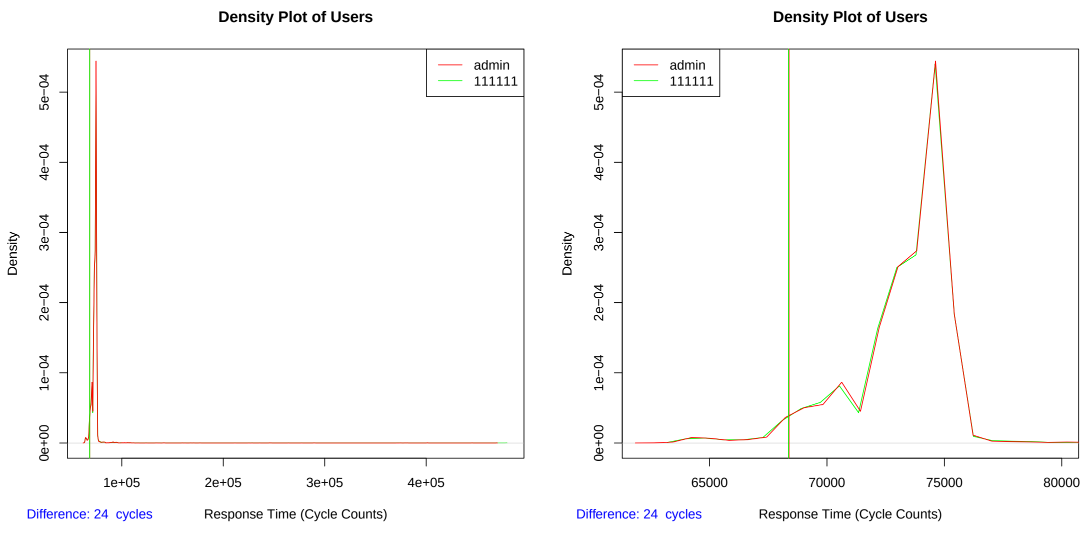
This chapter deals with the experiment results. We performed various experiments to assess the effectiveness of this attack. All experiments have been to factorise RSA keys of 1024-bits. All keys are generated randomly. OpenSSL v0.9.7 has been used during Interprocess attack, while OpenSSL v1.0.1i has been used during switched LAN attack. RSA blinding is turned off in case of OpenSSL v1.0.1i. The results shown are from switched LAN attack unless stated otherwise.

6.1 Guess User in Login Server

With this experiment, we tried to guess a user in login server. The valid user that exists in the database of login server is "mishaukat". We could successfully guess this user using the timing attack. We performed this attack only on Interprocess network model, where server and attack client were running on the same machine. We present the results of the attack in this section. We tried to guess two users "mishaukat" and "admin". We compared the response time of the two users with user "111111" assuming "111111" is not a valid user.

6.1.1 Invalid User

We tried to guess whether "admin" exists in the database of login server. We could successfully guess that "admin" does not exist in the system. We could infer this because the response time for "admin" was close to that of an invalid user, "111111". Figure 6.1 represents the distribution of "admin" and "111111". The cycle difference between their response times is only 24 cycles, which is very close. This suggests that the response time of user "admin" is similar to an invalid user. So, we guess that the user "admin" does not exist in the system. The plot **??** gives a closer look on the difference.

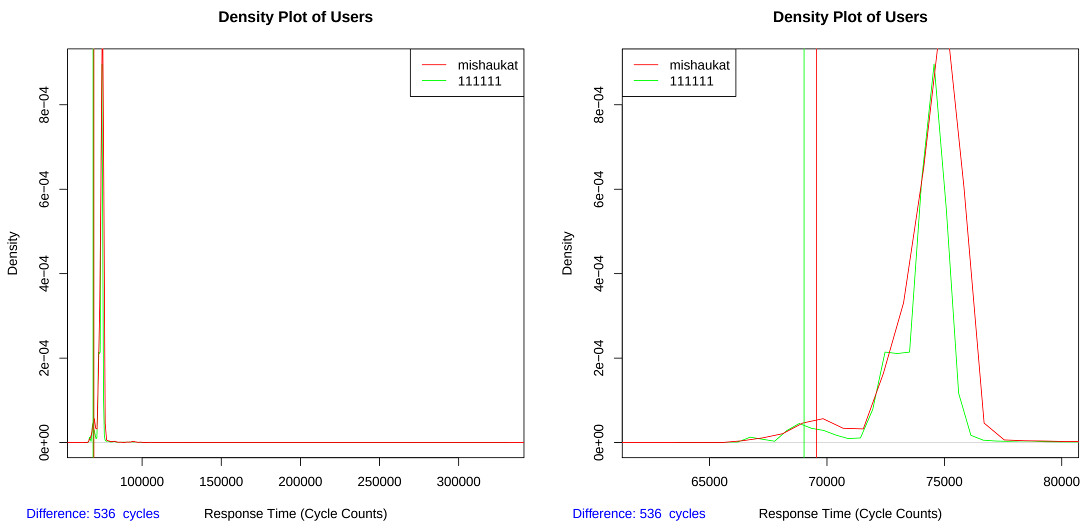


(A) "admin" versus "111111" (B) ZoomIn: "admin" versus "111111"

FIGURE 6.1: Response Time Distribution for an Invalid User

6.1.2 Valid User

We tried to guess whether "mishaukat" exists in the database of login server. We could successfully guess that "mishaukat" exists in the system. We could infer this because the response time for "mishaukat" was larger than that of an invalid user, "111111". Figure 6.2 represents the distribution of "mishaukat" and "111111". The cycle difference between their response times is 536 cycles, which is very large. This suggests that the response time of user "mishaukat" is larger than that of an invalid user. So, we guess that the user "mishaukat" exists in the system. The plot **??** gives a closer look on the difference.



(A) "mishaukat" versus "111111" (B) ZoomIn: "mishaukat" versus "111111"

FIGURE 6.2: Response Time Distribution for a Valid User

6.2 Time Attack on RSA

6.2.1 Sample Distribution and Filter

To cancel out jitter, we take multiple measurements for a particular message and then filter out one value as effective decryption time. As discussed earlier, the distribution of decryption time turns out to be skewed. Median as a filter works well for normally distributed data. So, we cannot take Median as a filter for our case. For such skewed distribution low percentile filters (typically 3-10 percentile) perform better. For our case, we have chosen 3-percentile filter because it has relatively less jitter. Figure 6.3 shows the distribution of decryption time with the help of density plot, while figure 6.4 shows the same with the help of cumulative density plot. It is clear from the plots that the distribution is highly skewed. Both figures show that the rising edge of the graph has less variation. So, we select this region to filter out one value. The vertical line represents the 3-percentile value, which means there are 3 percent values that lie below this value. We consider this 3-percentile value of the data as the effective decryption time for the message.

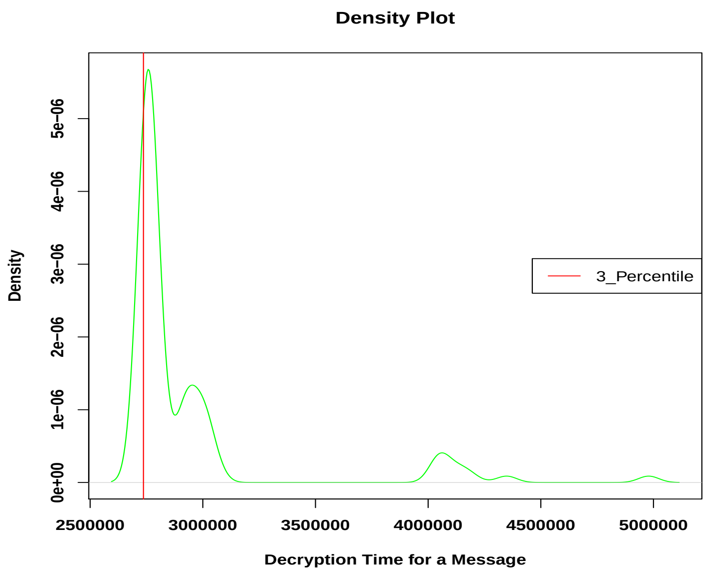


FIGURE 6.3: Density Plot: Distribution of Decryption Time for a message

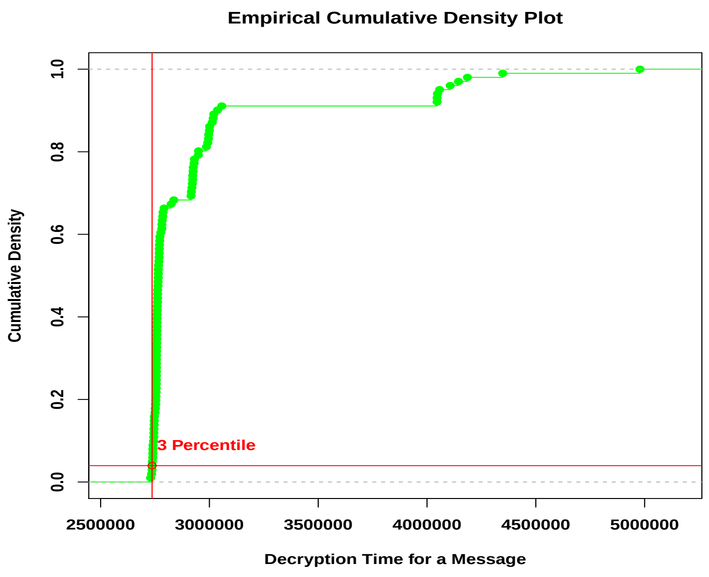


FIGURE 6.4: Cumulative Density: Distribution of Decryption Time for a message

We can use 3-percentile value to differentiate between two messages g and g_{hi}. Figure 6.5 shows the distribution of decryption time for two messages - g and g_{hi}. These two distributions almost completely overlap each other. It appears hardly distinguishable. However, we could use 3-percentile filter to differentiate between them. Figure 6.6 shows zoom-in view of the figure 6.5 so that we can have a better look at the differentiation. Solid vertical lines represent the 3-percentile value while dotted lines represent median. It is clear from the plot that we can differentiate better using the 3-percentile value than Median.

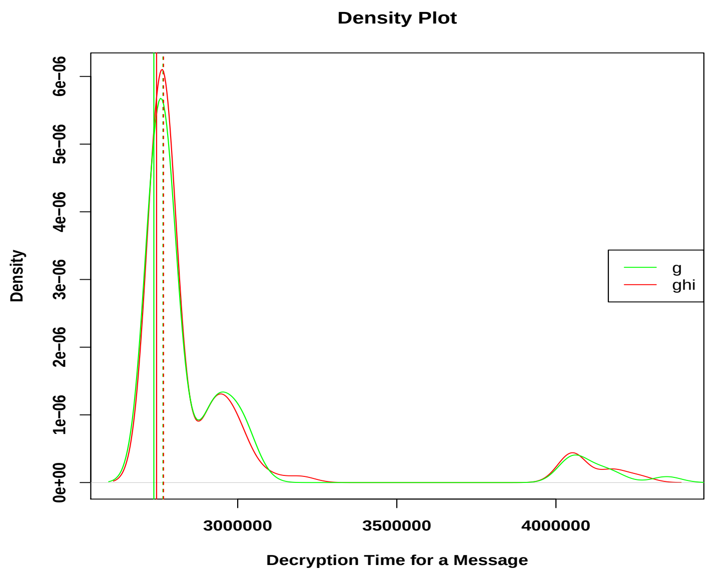


FIGURE 6.5: Distribution of two messages g and g_{hi}

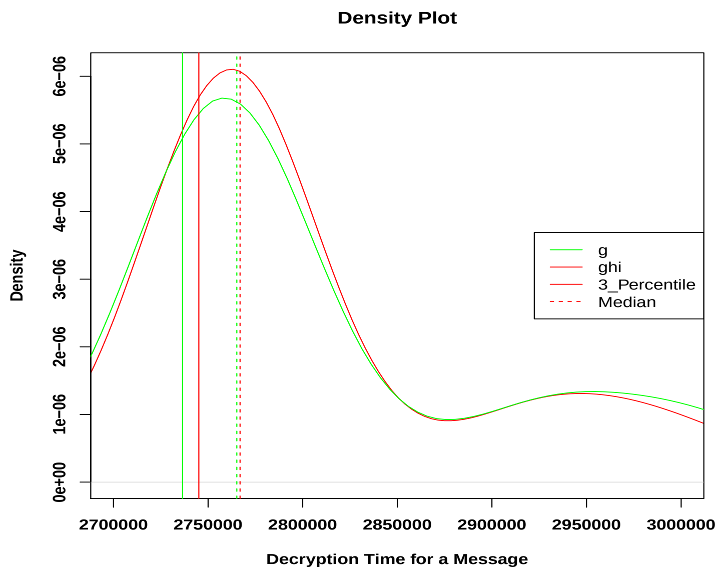
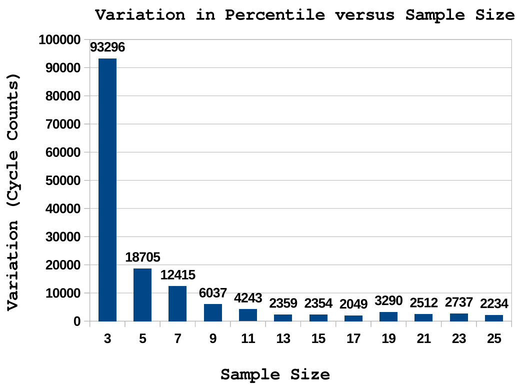
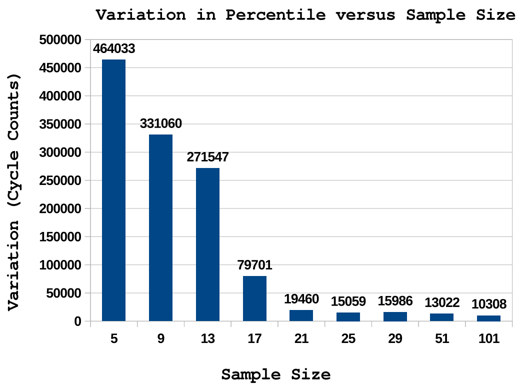


FIGURE 6.6: Zoom In: Distribution of two messages g and g_{hi}



(A) Variation in Interprocess Attack



(B) Variation in Switched LAN Attack

FIGURE 6.7: Decrease in Variation with Increase in Sample Size

6.2.2 Effect of Sample Size

As we discussed earlier, the attack has got two parameters - sample size s and neighbourhood size n. We need to find a way to tune in those parameters. In this section, we try to study the effect of sample size. To cancel out jitter we repeatedly take measurements for a particular message. Sample size s denotes the number of times we do it. It is intuitive that as we increase the sample size the variation will decrease. The experiments establish the same. Surprisingly the number of samples to bring the variation down is very low (7 samples) in interprocess attack. However, we need 21 samples for the same in switched LAN attack.

Brumley and Boneh pointed out that the network with variation under 1 milliseconds is vulnerable to this attack. For our set up, we realized that variation less than 20000 cycles works out well. We could achieve variation under 20000 cycles with 7 samples in interprocess attack and with 21 samples in switched LAN attack. However, for improved results we took 15 samples in interprocess attack and 25 samples in switched LAN attack.

6.2.3 Effect of Neighbourhood Size

SWE makes it hard to distinguish between bit 0 and 1. The parameter *neighbourhood size* is used to distinguish between them. Increase in neighbourhood size for bit 0 increases the bitgap. On the contrary, it does not impact much on the bitgap for bit 1. Figure 6.8 shows this effect. It is clear from the plot that there is no difference between bit 0 and 1 when neighbourhood size is 0. As we increase the neighbourhood size the bitgap for bit 0 increases linearly, however, the bitgap for bit 1 does not change significantly. This makes the difference between bit 0 and 1 prominent. We use neighbourhood size of 400 for our experiment.

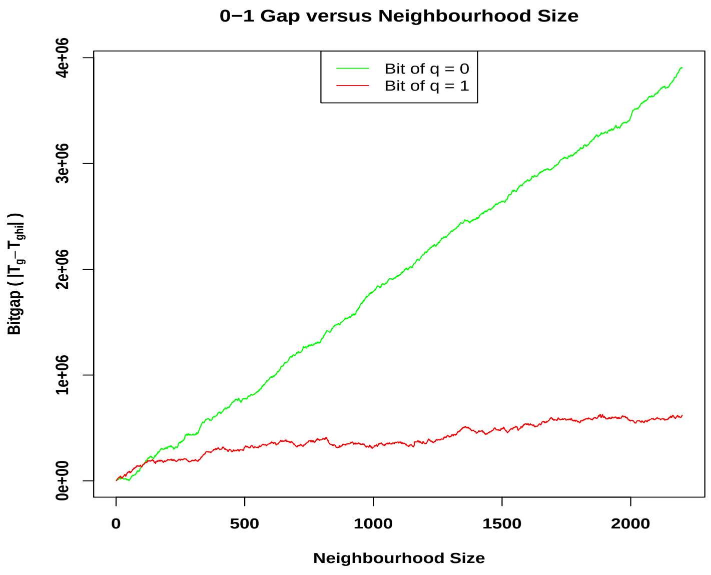


FIGURE 6.8: Effect of Increase in Neighbourhood Size

6.2.4 Recovery of Prime Factor

6.2.4.1 Find Peak

This is the first phase of the attack where we guess initial few bits of the prime factor q and p by identifying the peaks. We tried to guess initial 3 bits of the prime factors. We generated all the combinations of three bits from 000 to 111 and measured decryption time for all. After that we generated the plot and found two peaks that corresponds to two factors. First peak always corresponds to smaller prime factor q and second peak corresponds to larger prime factor p. Figure 6.9 shows the result of the first phase.

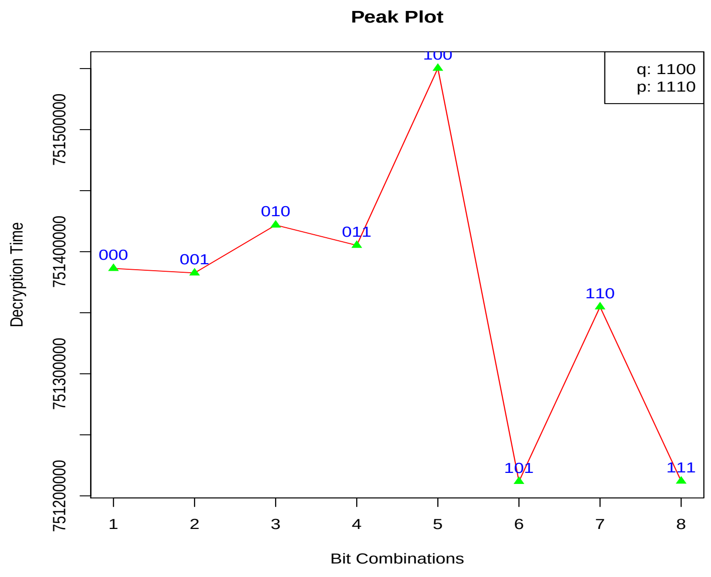


FIGURE 6.9: Peaks for Prime Factor q and p

We clearly see two peaks in the plot for bit combination *"100"* and *"110"*. The prime factors q and p have the same 3 bits "100" and "110" respectively. The plot shows top 4 MSBs of prime factors. The first topmost bit of both prime factors is always 1 as the prerequisite of SWE. This is always true for OpenSSL. So, we need to find the bits from second bit onwards. That makes the initial four bits of q and p as "1100" and "1110" respectively.

We have achieved this result with sample size 25 and neighbourhood size 10000. We need to increase the parameters because initial bits are hard to distinguish with lower values of parameters.

We can also try to find more that 3 bits in first phase. However, it is not recommended because it needs higher parameters, which take time and also it is often hard to identify peaks if bit combinations are more. So, it is suggested to find typically 2-3 bits in first phase of the attack.

6.2.4.2 Bit-by-Bit Recovery

This is the second phase of the attack where we find rest of the bits of smaller prime factor q one by one starting from MSB to LSB. Figure 6.10 shows the result of this phase. The green dots are bit 0 while red dots are bit 1. The horizontal blue line represents the value that we consider *"small"*. This value is derived using the values of initial bits that we have guessed in first phase. So, the bits that are below this line have been guessed as bit 1 and the bits that are above this line have been guessed as bit 0. Thus, all bits are correctly guessed by our attack client except initial few bits that appear in green below the blue line. They are actually bit 0 but guessed as bit 1 by our attack client. This gives us success rate of 96%.

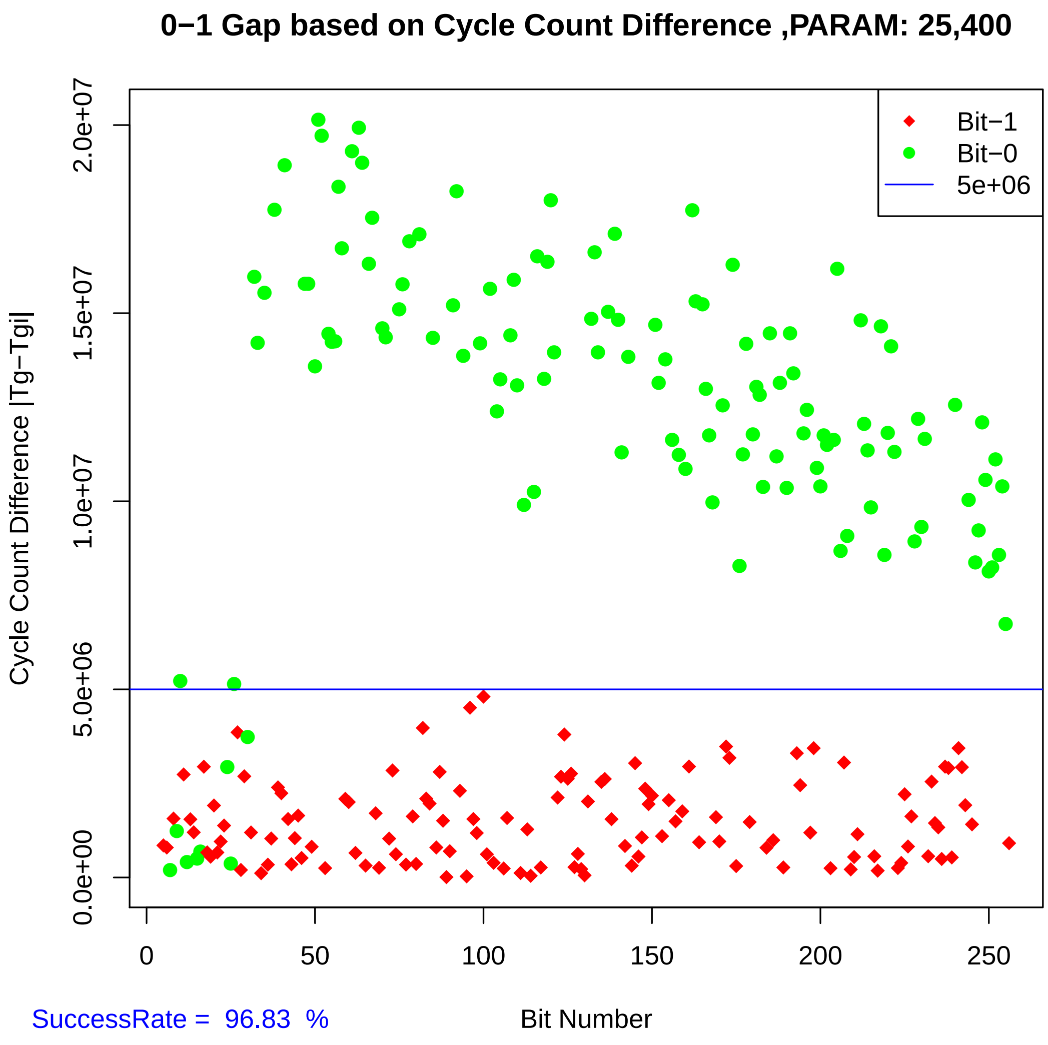


FIGURE 6.10: BitgapPlot: Recovery of bits of Prime Factor q

We have used 25 as sample size and 400 as neighbourhood size for this experiment. This is specified with *PARAM=25,400* in the title of plot. It is hard to distinguish initial bits with low neighbourhood size since the messages g and g_{hi} are quite far. We need to increase the neighbourhood size to classify initial bits properly.

Recall that we need to recover only half of the bits of prime factor using the timing attack. The other half is assumed to be recovered using the Coppersmith Algorithm [18]. That is why the plot shows the bits till 256 (half of 512).

Figure 6.10 shows the result when we considered *bitgap* for decision. Figure 6.11 shows the same result when we considered *positive-negative ratio* for decision. We found with experiments that they approximately give the same results and the plot establishes the same. Both decision strategies give us the success rate of 96%.

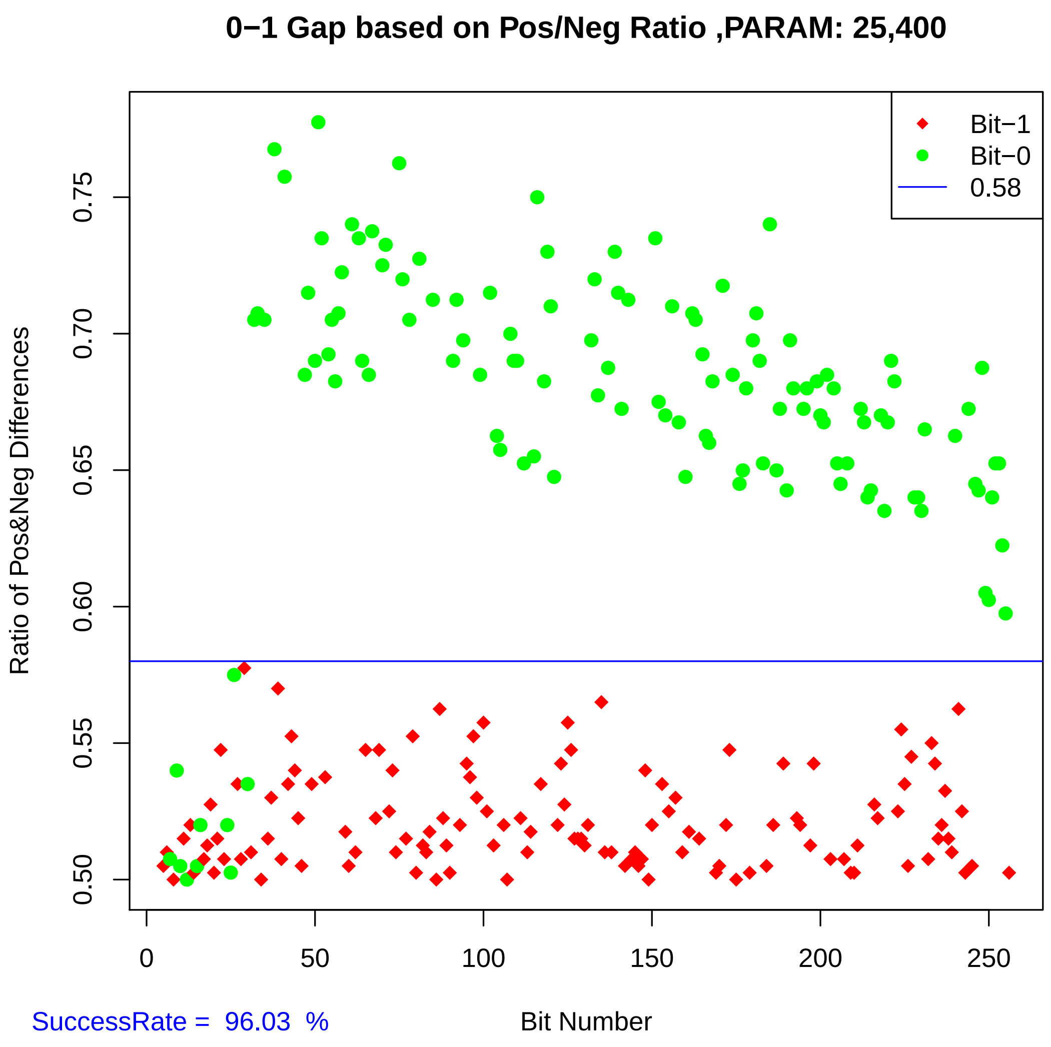


FIGURE 6.11: RatioPlot: Recovery of bits of Prime Factor q

6.2.5 Other Experiments

We tried different experiments to test the efficiency of attacks. Result shown in previous section simply explains the recovery of prime factor for a key. In this section, we present different experiment results.

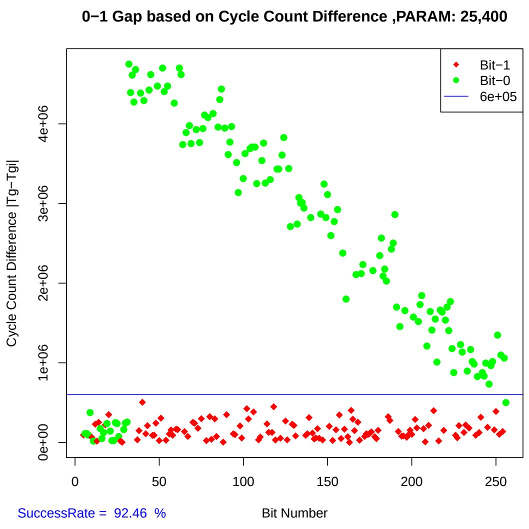
6.2.5.1 Comparison of Two Decision Strategies

As explained earlier, we have implemented two decision strategies - *Bitgap* and *Ratio*. Bitgap is used by BB-attack, while Ratio is used by Schindler-attack. We tried to find out which one provides better 0-1 gap. During experiments we observed that both strategies give more or less same results. Figures 6.12a and 6.12b provide the comparison of both decision strategies. It is clear from the plot that both strategies almost have same results. If we examine the plots more closely, we observe that "Ratio strategy" have larger 0-1 gap than that of "Bitgap strategy". This suggests that we can guess the bits more precisely with "Ratio strategy".

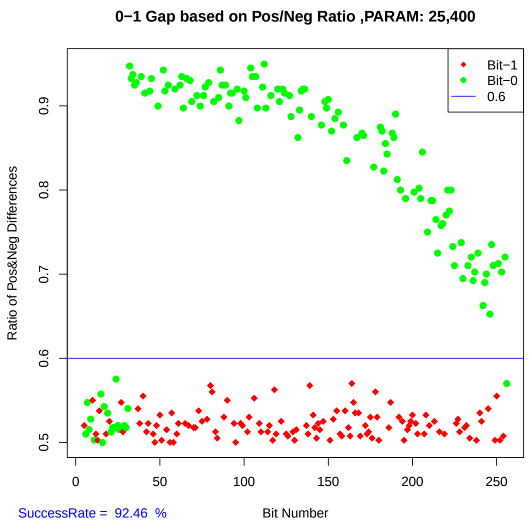
6.2.5.2 BB-attack versus Schindler-attack

This experiment is intended to compare the efficiency of both attacks - BB-attack and Schindler-attack. We kept the parameters same to have a better comparison. According to Schindler et al [19], Schindler-attack provides an improvement with a factor of 10. Also, it gives larger 0-1 gap with low neighbourhood size. We tried to compare the attacks on these aspects. However, we could not establish the claim. On the contrary, Schindler-attack needed higher neighbourhood size than BB-attack to have similar 0-1 gap. Figures 6.12 and 6.13 show the results of bit recovery for the same key using same parameters using BB-attack and Schindler-attack respectively. It appears from the plots that BB-attack could give better results with same parameters than that of Schindler-attack. We could recover 92% of bits using BB-attack, on the other hand, same parameters give 84% using

Schindler-attack. We needed to increase the neighbourhood size from 400 to 800 to have the same result for Schindler-attack.

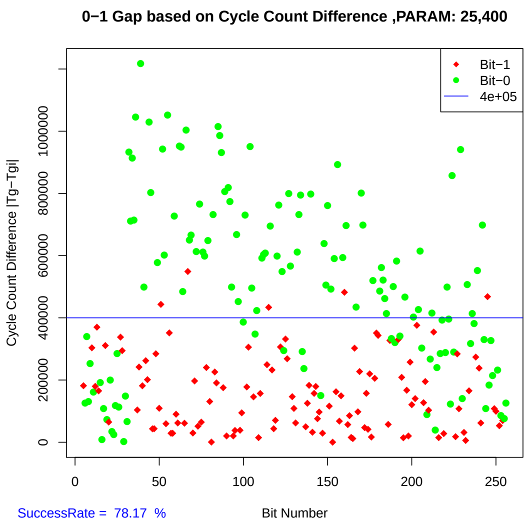


(A) Bitgap decision strategy

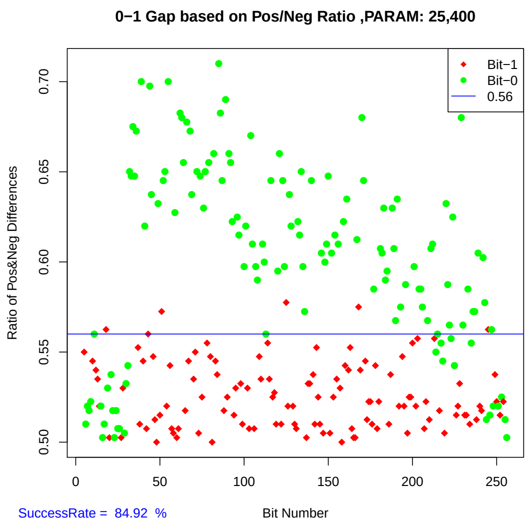


(B) Ratio decision strategy

FIGURE 6.12: BB-attack : Recovery of prime factor q



(A) Bitgap decision strategy



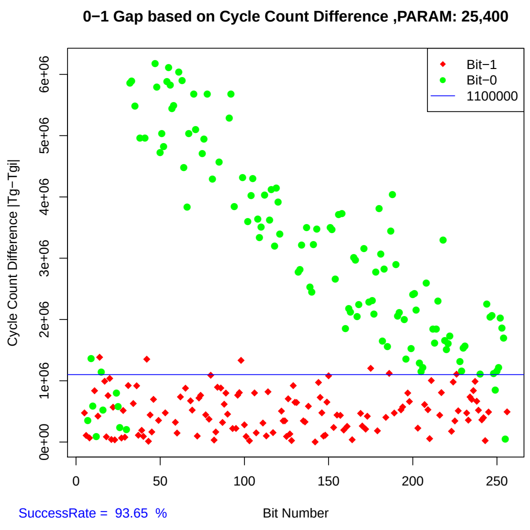
(B) Ratio decision strategy

FIGURE 6.13: Schindler-attack : Recovery of prime factor q

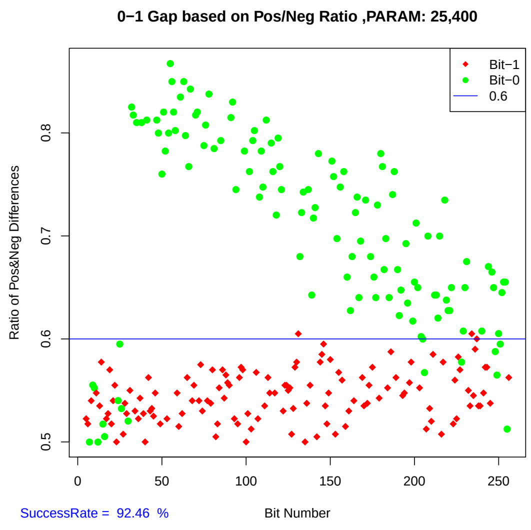
We did observe the efficiency of Schindler-attack over BB-attack during first phase of attack. We could not find the two peaks using the BB-attack. However, we were able to find the two peaks using Schindler-attack.

6.2.5.3 Different Keys

Several experiments have been performed to assess the efficiency of the attack in recovering different keys. The keys are randomly generated using OpenSSL. The success rate did not change for different keys and the results are almost same. We did the experiments on 4 different keys; however, we present here the results for two keys for comfortable comparison. We could achieve up to 95 % for both keys. We have used BB-attack for the comparison. Figures 6.14 and 6.15 show the recovery of prime factor for key1 and key2 respectively. It is clear from the plots that we can recover different keys with almost equal success rate. We might need to adjust the parameters a bit to have clear 0-1 gap.



(A) Bitgap decision strategy



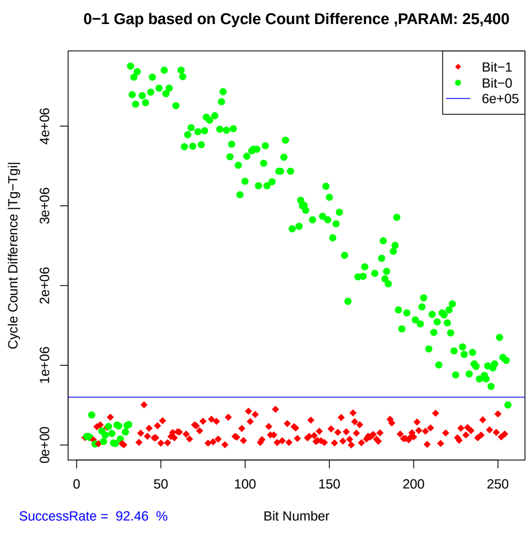
(B) Ratio decision strategy

FIGURE 6.14: KEY 1 : Recovery of prime factor q

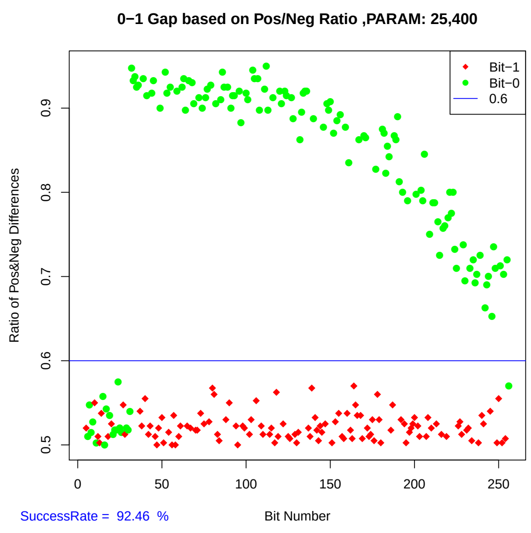
6.2.5.4 Interprocess versus Switched LAN

This section discusses the efficiency of the attack on two network models - (1) Interprocess, and (2) Switched LAN. We could achieve a success rate up to 95-96 % on both network models.

It is clear from the sample study that we needed more samples in Switched LAN attack to cancel out jitter as compared to Interprocess attack. Intuitively, it should



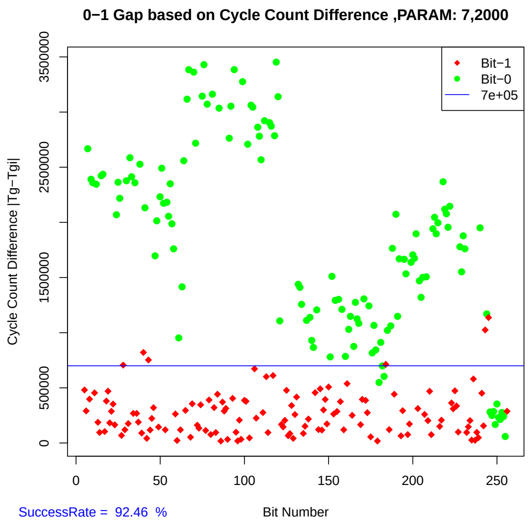
(A) Bitgap decision strategy



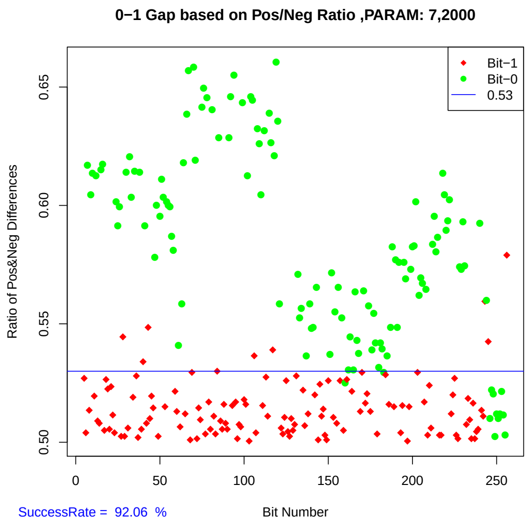
(B) Ratio decision strategy

FIGURE 6.15: KEY 2 : Recovery of prime factor q

be simpler to recover the key in Interprocess attack than Switched LAN. On the contrary, we needed low neighbourhood size in Switched LAN attack to recover the same key. Figures 6.16 and 6.17 represent a comparison between the attacks on two network models. We used sample size of 7 and 25 in Interprocess attack and Switched LAN attack respectively. However, we needed neighbourhood size of only 400 in Switched LAN attack as compared to 2000 in Interprocess attack. In Interprocess attack, the bits 120-180 were not clear with low neighbourhood size. So, we increased the neighbourhood size to make it clear. This suggests that it is not difficult to perform the attack over network. It simply requires precise measurement techniques to cancel out jitter.

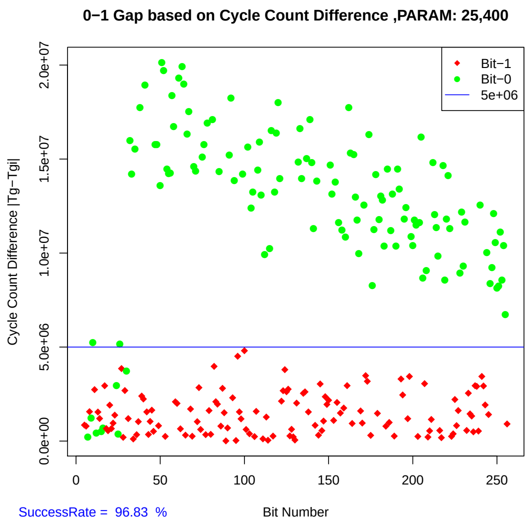


(A) Bitgap decision strategy

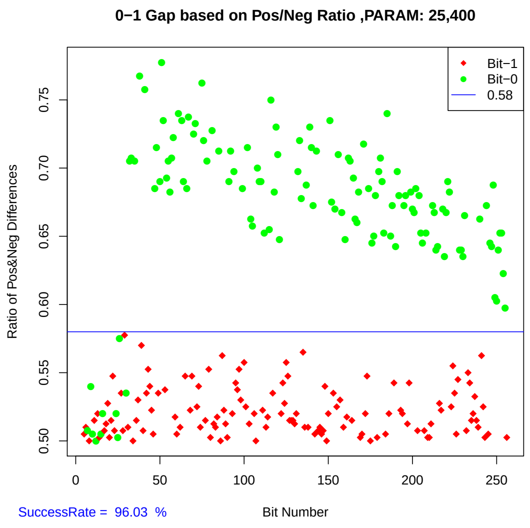


(B) Ratio decision strategy

FIGURE 6.16: Interprocess Attack: Recovery of prime factor q



(A) Bitgap decision strategy



(B) Ratio decision strategy

FIGURE 6.17: Switched LAN Attack : Recovery of prime factor q

Chapter 7

Future Enhancements & Conclusion

We discussed our strategy along with the implementation and results so far. We could achieve success rate of 96% over network. So, scope of improvement is still there. This chapter deals with the scope of improvements in our strategy and implementation. We would also discuss the defenses. We implemented our own server to demonstrate the attack. But we can also perform this attack on real web server. Real life attack would also be discussed here. We will go on to see the feasibility of the attack as well.

7.1 Real Life Attack

We implemented a simple decryption server to demonstrate the attack. However, real life scenario is quite different. The attack would lose its importance if it were not feasible in real life.

We can perform the same attack on real web server. First we need to understand how a real web server and client communicate with each other. Real web server uses hybrid cryptography - combination of asymmetric and symmetric cryptography. Asymmetric cryptography is used to authenticate each other and to exchange a symmetric key. The symmetric key is used for further data encryption.

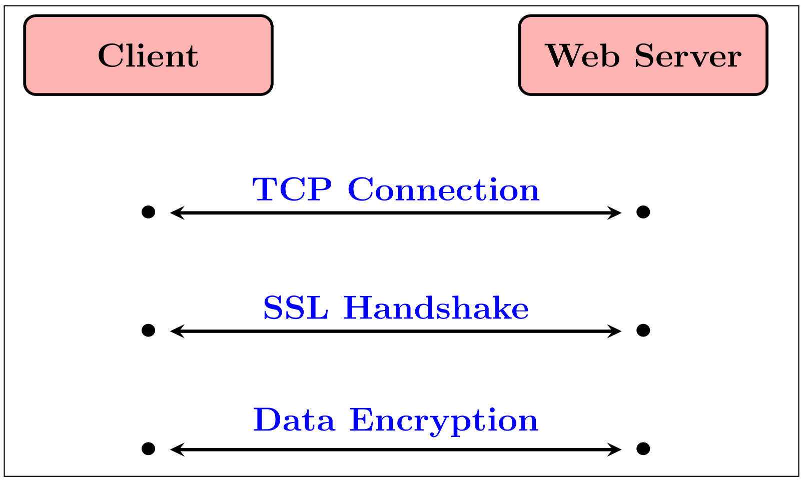


Figure 7.1: Real life Client-Server communication

Figure 7.1 represents the real life communication of a web server and client. First they make a *TCP connection*, which is followed by *SSL handshake*. During *SSL handshake* client and server authenticate each other and share a symmetric key. This symmetric key is used for further encrypted communication. RSA is used during key exchange if the server selects appropriate cipher suite. So, during key exchange step we can attack the real life server.

7.1.1 SSL Handshake

SSL Handshake is performed after making the initial TCP connection. This step is done to authenticate each other and exchange a symmetric key. Figure 7.2 shows the messages exchanged between client and server during this step. We are interested in *ClientKeyExchange* message, highlighted with red rectangle, through which symmetric key is exchanged. RSA is used for key exchange if selected cipher suite includes RSA algorithm as key exchange algorithm. Client generates a "master secret or symmetric key" and encrypts it using public key of server. Client sends the encrypted message or ciphertext to server. Server decrypts the ciphertext using its private key to extract the symmetric key. Only at this instance server uses its private key. Thus, this is the point where we can attack server to recover the private key.

7.1.2 Attack Procedure

We target the *ClientKeyExchange* message during SSL handshake to attack the real web server. We generate message g and replace the *ClientKeyExchange* message with it and send it to server to decrypt it, similarly for g_{hi}. Thus, we measure the decryption time for g and g_{hi} and use our algorithm to recover prime factor.

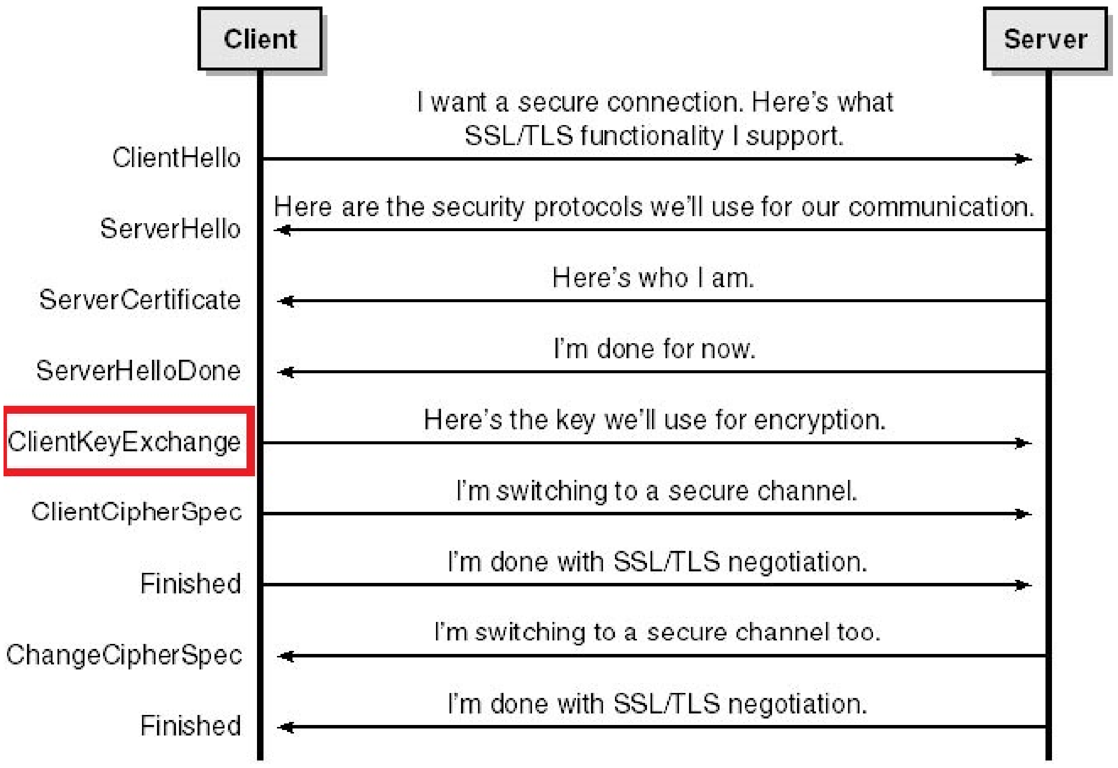

Client
Server
I want a secure connection. Here's what SSL/TLS functionality I support.
ClientHello
Here are the security protocols we'll use for our communication.
ServerHello
Here's who I am.
ServerCertificate
I'm done for now.
ServerHelloDone
Here's the key we'll use for encryption.
ClientKeyExchange
I'm switching to a secure channel.
ClientCipherSpec
I'm done with SSL/TLS negotiation.
Finished
I'm switching to a secure channel too.
ChangeCipherSpec
I'm done with SSL/TLS negotiation.
Finished

FIGURE 7.2: Messages exchanged during SSL Handshake

7.2 Future Enhancements

We could achieve a success rate of up to 96% that means we still need to brute force the other 4% of bits. Also, recovery of bits is dependent on recovery of previous bits. This makes the situation more complicated. Therefore we need error detection and correction scheme. Thus, we foresee little scope of improvements.

7.2.1 Error Detection and Correction

Decision of current bit depends on previously recovered bits. So, it becomes essential that each bit is recovered accurately. So, we need a mechanism to detect error in our guess and correct it.

Error correction and detection scheme is proposed by Chen et al [5]. According to this proposed scheme the *bitgap* or δ will always be *small* after a bit is guessed incorrectly. Assume a bit is guessed incorrectly. Thus both messages g and g_{hi} would either be ”larger than” or ”smaller than” q. Therefore, both case would force us to guess the bit as 1 because the bitgap would always be *small*. We can safely assume that there will not be 20 or more consecutive bits as 1 in prime factor. So, if we find this, we know that there is an error in our guess and, then, we can correct it. There is a drawback with this approach that we would not detect the error immediately but after 20 or more bits. So, we must keep track of this.

We propose this scheme as the future enhancement.

7.2.2 Dynamic Neighbourhood Size

Different parts of prime factor needs different neighbourhood size to get the desired 0-1 gap. For example, initial bits need more neighbourhood size and as we proceed further lesser neighbourhood size is enough. Our implementation fixes the neighbourhood size for all bits. This seems inefficient as it needs more number of queries, however we can perform the attack with fewer queries. We should consider higher value of neighbourhood size for initial bits and then we can decrease the value as it approaches further. The dynamic neighbourhood size approach will give more efficient implementation.

7.2.3 Implementation of Real Life Attack

We implemented the attack on a customised server. We figured out how the same attack could be performed on a real web server. It looks worth to set up the environment where we could perform the same attack on a real web server. It will reinforce the seriousness of the attack. Researches suggest that real life attack on a web server using Apache turned out be simpler.

7.3 Defences

Seriousness of the attack forces us to think about the defence. There is lot of options available as defense. The basic idea behind all the defenses is that there should not be any timing difference based on the input data or the secret. We have listed three such defenses below.

1. Recall the reason behind the timing difference. The reasons are - (1) conditional *extra-reduction*, and (2) two different multiplication routines. So, first defense suggests to always carry out *extra-reduction* and to use only one *multiplication routine*. If *extra-reduction* is not needed, one must carry out a dummy one.

2. We can quantize all RSA operations such that all decryption must take the maximum time of any decryption. This will make all RSA decryptions to take the same amount of time.

3. Other defense is known as RSA Blinding. We consider a random value r, encrypt it, and then perform the decryption. We calculate $x = gr^e \bmod N$ before decryption [12, p. 12]. After that, x is decrypted. The decryption is followed by division by r. Since r is random, this introduces randomness in the decryption time. This is a preferable defines and has performance penalty of 2-10%.

RSA blinding is already implemented in OpenSSL, however it was optional and was disabled by default. It was enabled by default with version 0.9.7b onwards as a countermeasure to the timing attack.

7.4 Summary

The work, presented here, deals with timing attack on a server, which uses OpenSSL, a SSL/TLS library, to provide security. Using the attack it is possible to recover the private key of the server. The work discusses and demonstrates the vulnerability in implementation of RSA decryption by OpenSSL. The attack requires on average 1.5 million queries and takes 2.5 hours to complete. It also presents few defences and proposes one as preferable defence.

It seems needless to emphasise on the seriousness of the attack because private key is the identity of an individual or Organization in the computing world. If private key is compromised it simply implies that the identity is compromised. For example, if an attacker is able to get the private key of a server, say Google server, he can pose himself as Google server and can trick consumers of Google server. One can only imagine the impact of it. It is a big deal. If an attacker recovers the private key of a Certificate Authority, it can corrupt the whole Public Key Infrastructure (PKI). We can go on and on to mention the impact, however we will find ourselves unable to discuss all.

The attack is demonstrated on OpenSSL, which is a widely used SSL/TLS library. It is an open source library. Most web servers are hosted using Apache, which uses OpenSSL for providing security.

The attack leverages the vulnerability in RSA implementation. RSA is an asymmetric cryptography algorithm, which is used now a day by servers to exchange symmetric key with clients. So the server is vulnerable to this attack when it uses RSA as key exchange algorithm.

The attack is demonstrated on a switched network. However, it can be performed on small local networks with few routers. Researchers have pointed it out that the network with less than 1 millisecond variation is vulnerable to such attacks. The attack needs more precise measurements and sophisticated setup on big networks. Nevertheless, it does not completely reject the feasibility of such attack over big networks. Moreover, the attacker can position himself nearby the server and transform it into a local network attack. One can argue that Intrusion Detection System (IDS) would catch those many requests from client and report it as rogue connection deeming the attack impractical. Firstly, there are many ways to trick the IDS and, secondly, the attacker can transform the attack into a local network attack where IDS would not interfere.

7.5 Conclusion

Timing attacks have been demonstrated mostly in the context of hardware. It is a common belief that these attacks are impractical in software context due to factors that impede in precise time measurement. Owing to this notion as well as lack of knowledge regarding timing attacks, developers tend to ignore timing vulnerabilities in their implementation. On the contrary, this sophisticated attack breaks all such notions and establishes the idea as a fact that timing attacks are practical and it can leak big deal of information. Therefore, it becomes necessary to defend against timing attacks in a world where everything is connected to other and the connectivity is growing day by day. There is a saying that "there is no bigger enemy than ignorance". Lack of knowledge will endanger us. Therefore, developers should be made aware of such attacks so that they can critically examine their implementation in light of such timing attacks. I would like to conclude with the statement that remote timing attacks are practical and we must defend against it.

Appendix A

Code Snippets

A.1 Benchmark a Function

```
unsigned int hi, lo;
uint64_t start, end, elapsedCount;
start = end = elapsedCount = 0;

//Record TSC before the function call
  __asm__ volatile(
     "cpuid\n\t"
      "rdtsc\n\t"
     "mov %%edx, %0\n\t"
     "mov %%eax, %1\n\t":"=r"(hi),"=r"(lo)::"%rax","%rbx","%rcx","%rdx"
  );
  start = ( (uint64_t)hi << 32 ) | lo;

//Call the function
  performDecryption();

//Record TSC after the function call
  __asm__ volatile(
      "cpuid\n\t"
```

```
    "rdtsc\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    "cpuid\n\t":"=r"(hi),"=r"(lo)::"%rax","%rbx","%rcx","%rdx"
  );
  end = ( (uint64_t)hi << 32 ) | lo;

//Calculate the difference to get the elapsed time
  elapsedCount = end - start;
```

Listing A.1: Sample Code to benchmark a function

Bibliography

[1] Wikipedia. Cryptography. http://en.wikipedia.org/wiki/Cryptography, Online accessed 01-April-2015.

[2] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. ISBN 0849385237.

[3] OpenSSL Project. Cryptography and ssl/tls toolkit. http://openssl.org/, Online accessed 10-April-2015.

[4] C. Ding, D. Pei, and A. Salomaa. *Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[5] Peter L. Montgomery. Improving timing attack on rsa-crt via error detection and correction strategy. *Information Sciences*, 232(0):464 – 474, 2013. ISSN 0020-0255. URL http://www.ams.org/publications/journals/journalsframework/mcom.

[6] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519 – 521, 1985. ISSN 0025-5718. URL http://www.ams.org/publications/journals/journalsframework/mcom.

[7] GNU Open Source. The gnu multiple precision arithmetic library. https://gmplib.org/, Online accessed 10-April-2015.

[8] Paul Gray. And bomb the anchovies. *Time Magazine*, 1990.

[9] *Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow*, May 2010. IEEE Computer Society. URL http://research.microsoft.com/apps/pubs/default.aspx?id=119060.

[10] Wikipedia. Side-channel attack. http://en.wikipedia.org/wiki/Side_channel_attack, Online accessed 10-April-2015.

[11] Werner Schindler. A timing attack against rsa with the chinese remainder theorem. In *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2000. doi: 10.1007/3-540-44499-8_8.

[12] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251353.1251354.

[13] Intel Developer Support. *Using the RDTSC Instruction for Performance Monitoring*. Intel Corporation.

[14] Gabriel Torres. Everything you need to know about the cpu c-states power saving modes. *Hardware Secrets*. URL http://www.hardwaresecrets.com/article/611.

[15] Stuart Hayes. *Controlling Processor C-State Usage in Linux*. DELL.

[16] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3):17:1–17:29, January 2009. ISSN 1094-9224. doi: 10.1145/1455526.1455530. URL http://doi.acm.org/10.1145/1455526.1455530.

[17] PaulC. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology —*

CRYPTO '96, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61512-5. doi: 10.1007/3-540-68697-5_9. URL http://dx.doi.org/10.1007/3-540-68697-5_9.

[18] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997. ISSN 0933-2790. doi: 10.1007/s001459900030. URL http://dx.doi.org/10.1007/s001459900030.

[19] Onur Aciiçmez, Werner Schindler, and Çetin K. Koç. Improving brumley and boneh timing attack on unprotected ssl implementations. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 139–146, New York, NY, USA, 2005. ACM. ISBN 1-59593-226-7. doi: 10.1145/1102120.1102140. URL http://doi.acm.org/10.1145/1102120.1102140.