ELECTROMAGNETIC SIDE-CHANNEL ANALYSIS ON INTEL ATOM PROCESSOR

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

by

Anh Do

Soe Thet Ko

Aung Thu Htet

Date: April 24, 2013

Approved:

Professor Thomas Eisenbarth, Advisor

Professor Berk Sunar, Co-Advisor

Abstract

Side-channel attacks, in particular, power electromagnetic analysis attacks, have gained significant attention in recent years because they can be conducted relatively easily, yet are powerful. Two types of attacks are investigated in this project: Simple Electromagnetic Analysis (SEMA), which is extremely effective against asymmetric cryptography such as Rivest-Shamir-Adleman Algorithm (RSA) and Differential Electromagnetic Analysis (DEMA), which is commonly implemented against symmetric cryptography such as Advanced Encryption Standard (AES). This project implements SEMA and DEMA attacks based on the electromagnetic radiation from the Intel Atom Processor during its execution of cryptography algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Motivation

This section give a brief background of cryptography, side-channel attacks, and how they motivate us to conduct side-channel analysis on an Intel Atom Processor.

1.1.1 Cryptography

Encryption is the primary cryptographic technique to achieve confidentiality of messages between two communicating parties over public networks. Nowadays, most of the cryptographic applications are conducted over digital media. To cite a few, applications include ATM cards, computer passwords, electronic commerce, etc. In such applications the encryption process typically involves two parties: encryption at the sender side and decryption at the receiver side. Encryption is the process of converting the original message (called *plaintext*) into a secret coded message (called *ciphertext*). Decryption achieves the reverse, i.e. reconstructing the plaintext from the received ciphertext. The encryption algorithm uses a key, i.e. a secret, which when combined with a mathematically defined algorithm renders the plaintext into an unintelligible ciphertext. At the receiving end, the key is used to decrypt the code and restore the original message. The greater the number of bits in the secret key, the more possible key combinations and the harder it would take to break the code using an exhaustive search strategy.

There are two major categories of encryption schemes: symmetric and asymmetric encryption schemes. In the symmetric key category, both the sender and receiver use the same key to encrypt and decrypt. A prominent example is the Advanced Encryption Standard (AES) algorithm which will be described later in Section §1.4. In symmetric ciphers encryption and decryption primitives are much faster than in public key algorithms. However securely delivering the secret key to the recipient in the first place might be a problem. In contrast, primitives in the second category use a key pair. Each recipient has a private key that is kept secret and a public key that is published for everyone. The sender uses the recipient's public key to encrypt the message, and the recipient uses the private key to decrypt the message. The RSA public key scheme, described later in Section §1.2, is a good example of the schemes in this category.

1.1.2 Side-channel Attacks

There are several kinds of attacks on cryptographic devices. They share the same goal, to reveal the secret keys of the devices, but differ significantly in terms of cost, time, equipment, and expertise needed. These methods could be categorized as follows:

- Invasive Attacks: The device is depackaged, and different components of the device are accessed directly using a probing station. These attacks are extremely powerful but require quite expensive equipments and may be very difficult to perform under some circumstances.

- Semi-invasive Attacks: The device is depackaged, but no direct electrical contact to the surface is made. These attacks are powerful, and do not require as expensive equipments as invasive ones. However, the process of locating the right position for an attack requires quite some time and expertise.

- Noninvasive Attacks: Only directly accessible interfaces of the device are exploited, and therefore no evidence of an attack is left behind. These attacks can be conducted with relatively inexpensive equipments, and hence, they pose a serious practical threat to the security of cryptographic devices.

Most of the cryptographic schemes in use have been heavily investigated by cryptography experts to withstand powerful mathematical attacks. In contrast, most of the very same techniques have been shown to be vulnerable to attacks targeting their implementations through so-called *side-channel attacks* [4]. Side-channel attacks, in particular, electromagnetic analysis attacks based on the devices' radiation, have gained quite attention in recent years because they are very powerful and can be conducted relatively easily. These attacks exploit the fact that the instantaneous electromagnetic radiation of a device depends on the data it processes and on the operation it performs [7]. There are two main types of power analysis attack: Simple Electromagnetic Analysis (SEMA), which is extremely effective against asymmetric cryptography and Differential Electromagnetic Analysis (DEMA), which is extremely effective against symmetric cryptography. The detail procedures of these two attacks will be discussed more in Section §1.3 and Section §1.5.

1.1.3 Motivation

Our project involves implementation of side-channel attacks on Intel Atom Processor. In 2008, Intel introduced the processors Silverthorne and Diamondville, which were both released under the name of Atom [1]. Intel targets the Silverthorne for mobile internet devices and Diamondville for personal computers. Intel Atom processors are high speed, and high performance embedded processors, and they are quite resistant to side-channel attacks. The fact that it is challenging to implement side-channel attack on Atom board is one of our project motivations. Another project motivation is the widespread use of Atom processors. They are used in everyday electronic devices, from net books to various embedded applications. The Pine Trail Architecture used in Atom Processors brings considerable power saving and improved performance [2]. Therefore, instead of heavy quad core processors, Atoms Processors are used in devices such as Net books, net-tops, and smartphones, which are used for simple applications such as browsing the internet. With such a wide field of application, Atom processor is a practical application in real-world that is worth investigating for side-channel analysis.

1.2 Rivest – Shamir – Adleman (RSA) Algorithm

The RSA algorithm is currently the most widely deployed public key cryptographic scheme. In practice, RSA is mostly used for encryption of small pieces of data such as key transport, digital signatures and digital certificates on the Internet. Since it is several times slower than symmetric ciphers due to its computational complexity, RSA is often used together with a symmetric cipher such as AES, where AES does the actual bulk data encryption. With the use of both public key and private key, RSA has an important role to securely exchange a key for a symmetric cipher. Moreover, the underlying reason for the RSA algorithm is based on the integer factorization problem: Multiplying two large primes is computationally easy, while factoring the resulting product is very hard. In this section, again all the explanation of the AES algorithm is advised from the book "*Understanding cryptography a textbook for students and practitioners*" [10].

1.2.1 Encryption and Decryption

Both RSA encryption and decryption are performed in the integer ring $\mathrm{Z_n}$ with modular arithmetic as the main computation.

Encryption: Given the plaintext x, and the public key $k_{pub} = (n, e)$, the encryption function is

$$y = e_{k_{pub}}(x) \equiv x^e \mod n$$

Decryption: Given the ciphertext y, and the private key $k_{pr} = (d)$, the decryption function is

$$x = d_{k_{pr}}(y) \equiv y^d \mod n$$

where $x, y \in Z_n$

In practice, x, y, n, and d are very large integers, i.e. usually 1024 bits long or more. The value e is called the public exponent or encryption exponent, while the private key d is called the private exponent or decryption exponent. Based on the algorithm, here are some essential requirements for the RSA cryptosystem:

- It must be computationally infeasible to determine the private key d given the public key e and n
- The maximum bit length that could be encrypted is the size of the modulus n
- A method for fast exponentiation with long numbers is need
- For a given n, there should be many different key pairs

1.2.2 Detail Implementation of RSA

1.2.2.1 Key Generation

During the setup phase, the RSA public key and private key are computed as follows.

- Choose two large primes p and q
- Compute the modulus $n = pq$
- Compute $\phi(n) = (p-1)(q-1)$
- Select the public exponent $e \in \{1, 2 \ldots \phi(n) - 1\}$ such that $\gcd(e, \phi(n)) = 1$
- Compute the private key d such that $d \equiv e^{-1} \mod \phi(n)$

1.2.2.2 Fast Exponentiation

As stated above, RSA requires a fast exponentiation method for both encryption and decryption processes. The square-and-multiply algorithm is the most simple and commonly used for this purpose.

- Scanning the bits of the exponent H from the left (MSB) to the right (LSB)
- In every iteration, square (SQ) the current result
- If and only if the current bit is 1, multiply (MUL) the current result by the base x

In general, for an exponent H with a bit length of t+1, the numbers of operations needed are

$$\#SQ = t,\ \#\overline{MUL} = 0.5t,\ \#\overline{Total} = 1.5t$$

1.2.2.3 Fast Encryption with Short Public Exponents

A surprisingly simple yet powerful trick can be used to accelerate the encryption of a message and verification of an RSA signature is choosing the public key e. For this purpose, the public key is chosen to be a small value with low Hamming weight. Here are the three most common values are shown in Table 1.1.

Table 1.1: Short Public Exponents for RSA

Public key e	e(in binary)	#MUL + #SQ
3	11	3
17	1 0001	5
2^{16}+1	1 0000 0000 0000 0001	17

1.2.2.4 Fast Decryption with the Chinese Remainder Theorem

Unlike the public exponent, the private key d cannot be short since an attacker could simply brute-force all the possible numbers up to a given bit length. In practice, e is often chosen to be short, while d has full bit length. Thus, the Chinese Remainder Theorem (CRT) is applied to accelerate RSA decryption and signature generation. The 2 large primes, p and q, in the key generation phase are used in this transformation.

- In the first step, transformation into the CRT domain proceeds as follows.

$$y_p \equiv y \mod p,\ y_q \equiv y \mod q,\ d_p \equiv d \mod (p-1),\ d_q \equiv d \mod (q-1)$$

- In the second step, we compute the exponentiation in the CRT domain as follows.

$$x_p \equiv y_p^{d_p} \mod p,\ x_q \equiv y_q^{d_q} \mod q$$

- Finally, the inverse transformation is computed as $x \equiv qc_px_p + pc_qx_q \mod n$ where the coefficients are computed as $c_p \equiv q^{-1} \mod p,\ c_q \equiv p^{-1} \mod q$

1.2.2.5 Finding Large Primes

During the key generation process, an important aspect is to find two large primes p and q. The general approach is to generate integers at random then check for primality. This process depends on two important questions, “How common are primes?” and “How fast is the primality check?”. It turns out that even for large value, the density of primes is still sufficiently high. In fact, the probability for a random odd number to be prime:

$$P(\text{p is prime}) = \frac{2}{\ln p}$$

Moreover, the primality tests are also computationally inexpensive. Examples are the Fermat test, the Miller – Rabin test, and variants of them.

1.2.3 RSA in Practice

The RSA system described so far has several weaknesses.

- RSA encryption is deterministic, one-to-one mapping for a specific key
- Plaintexts $x = 0, -1, 1$ produce ciphertexts equal to 0, 1, -1
- Small public exponents and small plaintexts might be subject to attack
- Ciphertexts are malleable. A crypto scheme is said to be malleable if the attacker is capable of transforming the ciphertext into another ciphertext which leads to a known transformation of the plaintext. In the case of RSA, if the attacker computes a new cipher text $s^e y$ with s is some integer, then the receiver decrypts the manipulated ciphertext as $(s^e y)^d \equiv s^{ed} x^{ed} \equiv sx \mod n$. These manipulations might do harm since the attacker is able to get s times the plaintext.

A possible solution to all these problems is the use of padding, which embeds a random structure into the plaintext before encryption. A modern technique, Optimal Asymmetric Encryption Padding (OAEP), for padding RSA messages is specified and standardized in Public Key Cryptography Standard #1 (PKCS #1) [12]. On the decryption side, the structure of the decrypted message has to be verified.

1.3 Simple Electromagnetic Analysis (SEMA) Attacks on RSA

Here are the few common attacks on the RSA.

- Protocol attacks: exploit weaknesses in the way RSA is being implemented. Many of them can be avoided by using standard padding.
- Mathematical attacks: the best method is factoring the modulus. These attacks can be prevented by choosing a sufficiently large modulus.

- Side-channel attacks: exploit information about the private key which is leaked through physical channels such as power consumption and timing behavior. This now becomes a large and active filed of research in modern cryptography.

For this project, we apply a side-channel attack called the simple electromagnetic analysis method. The attacker will be able to find out the private key by observing the decryption process revealed by the devices' radiation, which is the exponential algorithm that has been demonstrated before. Therefore the attacker can decipher the encrypted text with both the public key and the private key. The exponential algorithm has been applied in many RSA cryptographic implementations. As the exponential algorithm has been illustrated in the previous session, there are only 2 different operations, square and multiply. The sequence of these limited operations composing the private key can be revealed by visual observation of the electromagnetic traces.

1.4 Advanced Encryption Standard (AES) Algorithm

The Advanced Encryption Standard Algorithm is the most widely used symmetric cipher today. The AES block cipher is also mandatory in several industry standards and is used in many commercial systems. In this section, all the explanation of the AES algorithm is advised from the book "*Understanding cryptography a textbook for students and practitioners*" [10]. In addition, all the figures are redrawn from the ones in this book.

1.4.1 Overview Structure

A general AES round is shown in Figure 1.1. The input x is a 128-bit block of data or plaintext. The key, denoted by k, can be 128 bits, or 192 bits, or 256 bits. The output y is the 128-bit ciphertext.

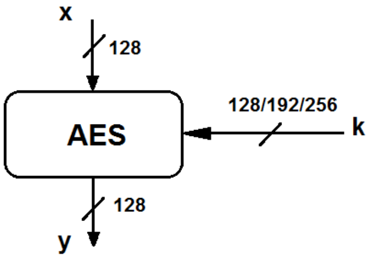


Figure 1.1: Basic AES Algorithm

An AES algorithm consists of several transformation rounds or cycles which convert the input plaintext to the output ciphertext. For different key lengths, the number of rounds varies as shown in Table 1.2. For example, if the key length is 128 bit, there will be 10 AES rounds.

Table 1.2: Number of Rounds as a function of Key Length

Key Length	Number of rounds
128 bits	10
192 bits	12
256 bits	14

The complete AES block diagram is shown below in Figure 1.2. The initial round, or round 0, consists of only one Key Additional Layer. Next, the sequence of AES rounds is consecutively performed, taking the output of the previous round as the current input. Each AES round consists of three so-called layers that manipulate the input data, which are

- Byte Substitution Layer: The data is transformed nonlinearly by using a lookup table, called the S-box, which has special mathematical properties. This nonlinear transformation causes confusion and randomization to the data; in other words, changes in individual state bits spread quickly across data path.

- Diffusion Layer: This layer provides diffusion over all state bits. It has two sublayers: ShiftRows Layer, and MixColumn Layer, both of which are linear transformations. The former performs byte-wise permutation to the data, and the latter performs matrix operation in order to mix or combine bytes of data.

- Key Addition Layer: the 128-bit round key or sub-key is combined with the current state by XOR operation. The sub-key is derived from Key Schedule which will be mentioned in details later in 1.4.4.

The output of the last round is also the final output of the AES cipher. However, the last round does not include MixColumn sublayer in the Diffusion Layer.

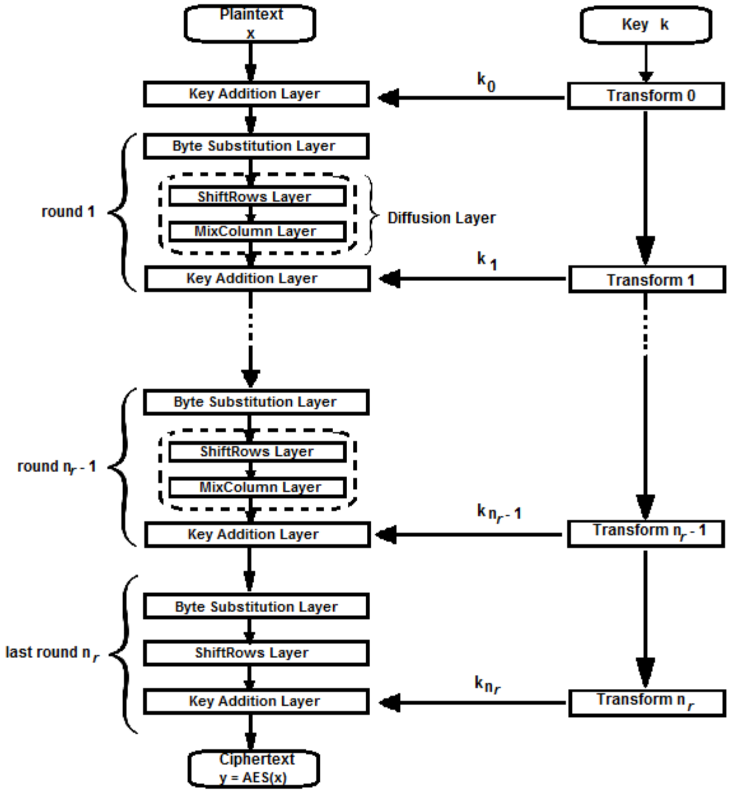
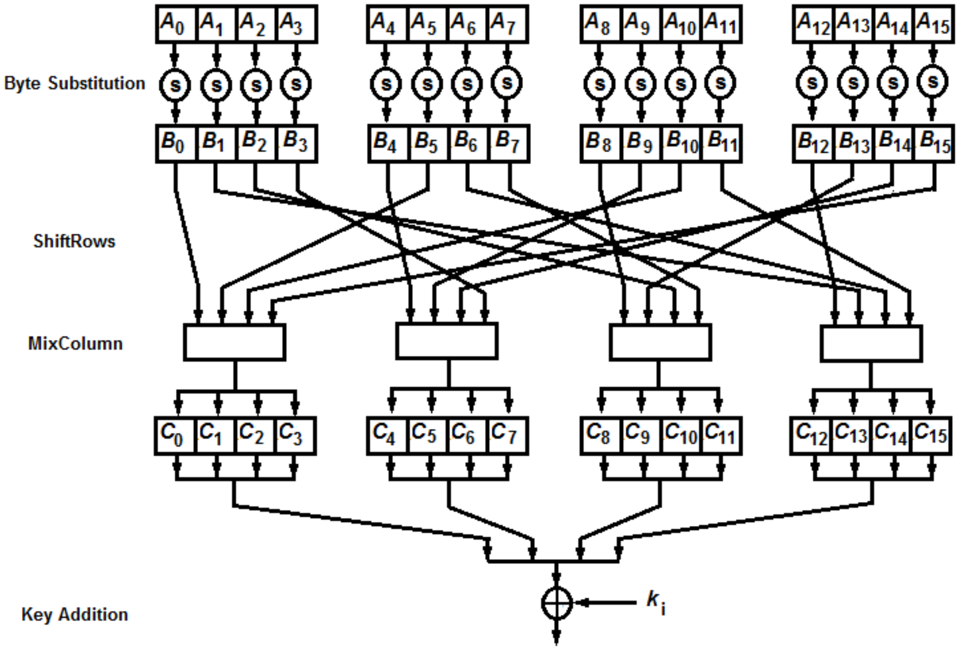


Figure 1.2: Detail Structure of an AES encryption

1.4.2 Internal Structure of AES

The internal implementation of a single round of AES is shown in Figure 1.3. First, the 128-bit plaintext is separated into 16 bytes, from A_0 to A_{15}. Each byte is then fed into the S-Box (Byte Substitution Layer). The 16-byte output from the S-box, from B_0 to B_{15}, is permuted byte-wise in the ShiftRows Layer. The result is mixed and combined in the MixColumn Layer, resulting in the intermediate value, from C_0 to C_{15}.

The intermediate value is then XORed with the 128-bit round key in the Key Addition Layer before going to the next round.

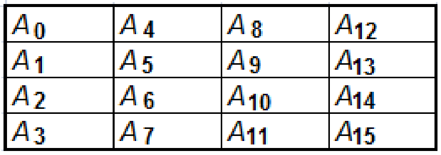


Byte Substitution
ShiftRows
MixColumn
Key Addition
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15
S
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15
ki

Figure 1.3: Internal Structure of an AES round

1.4.3 Detail Implementation of each Layer

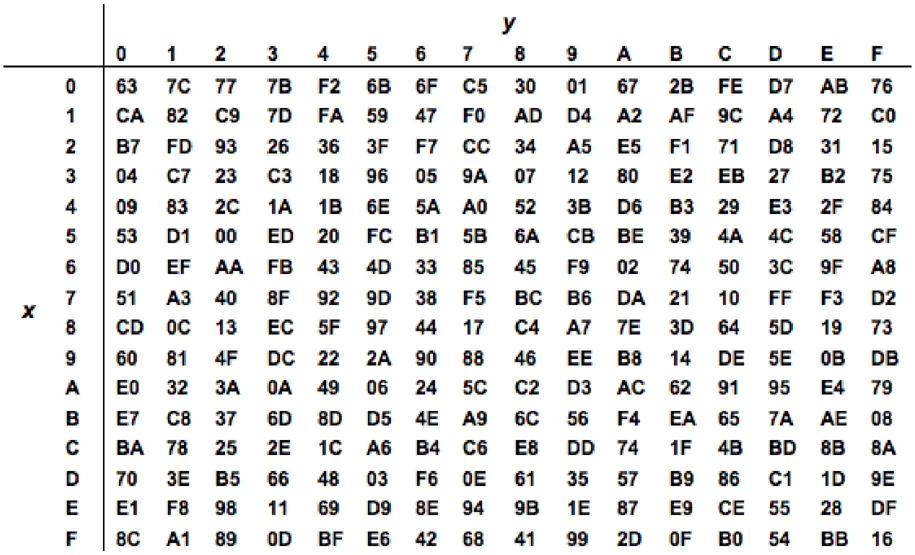
For convenience, the bytes of 16-byte input data is arranged in a 4×4 matrix, as shown in Figure 1.4.



A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

Figure 1.4: Input 4×4 matrix

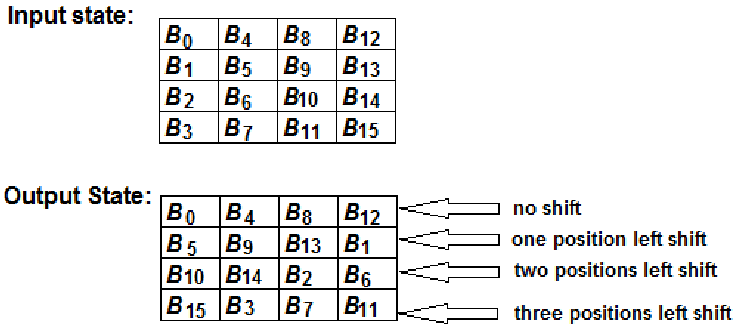
In Byte Substitution Layer, the bytes are transformed nonlinearly using the S-box, shown in Figure 1.5. For example, if the input byte is E9 in hexadecimal, the output is the element in row number E and column number 9, which is 1E in hexadecimal.



									y							
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
x	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 1.5: Lookup Table used in Byte Substitution Layer

The detail implementation of the ShiftRows sublayer is shown in Figure 1.6. The input state, a 4×4 matrix from B_0 to B_{15}, is the output from the Byte Substitution Layer. To achieve the output state, the first row of the matrix is unchanged, the second row is shifted one position to the left, the third row is shifted two positions to the left, and the fourth row is shifted three positions to the left.



Input state:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

Output State:

B_0	B_4	B_8	B_{12}	no shift
B_5	B_9	B_{13}	B_1	one position left shift
B_{10}	B_{14}	B_2	B_6	two positions left shift
B_{15}	B_3	B_7	B_{11}	three positions left shift

Figure 1.6: Input and Output States of the ShiftRows sub-layer

The new 4×4 matrix is fed into the MixColumn sublayer, and the output is calculated using the special matrix operation employing Galois Field Properties, as shown in Figure 1.7. Each column of the output matrix is calculated by performing the matrix operation with the corresponding column of the input 4×4 matrix.

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}$$

Figure 1.7: Matrix Operation in MixColumn sub-layer

Finally, the Key Addition Layer takes in two inputs: the current 16-byte state matrix, which is output from the MixColumn sublayer, and a round key. The output is obtained by performing XOR addition of these two inputs.

1.4.4 Key Schedule

There are different key schedules for three AES key sizes of 126, 192, and 256 bits. As all of the schedules are fairly similar, only the schedule for 128-bit key length will be discussed in detail. For the 128-bit key, there are 10 rounds in the AES encryption operation. However, there is an XOR addition of key before the first round. This means we need 11 sub-keys or round keys.

The key expansion process of an AES-128 is demonstrated in Figure 1.8. The input is the 128-bit original key, represented as bytes from K_0 through K_{15}.Note that the key schedule is word-oriented, and the 11 sub-keys are stored in a key expansion array as words from W[0] through W[43]. The first sub-key is obtained by directly copying the original key from the four elements W[0], W[1], W[2], and W[3] of the array. The remaining 10 sub-keys are computed recursively, using an XOR addition and a non-linear function g, which will be discussed later.

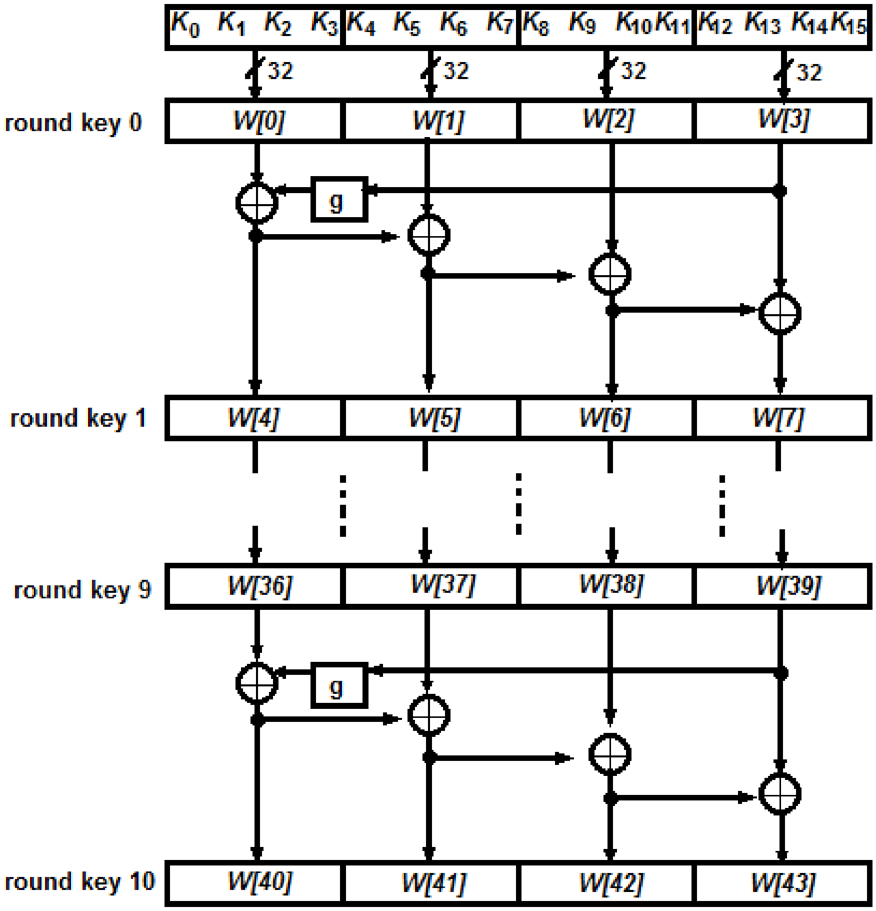


Figure 1.8: Key schedule for AES-128

As stated, function g plays an important role in sub-key derivation as it adds non-linearity to the key schedule. It takes four input bytes and produces four-byte output, as shown in Figure 1.9. It first rotates the input bytes, and then performs a byte-wise S-Box substitution. A round coefficient (RC) is added to the leftmost S-Box output byte. The RC value is an element of Galois field and varies from round to round.

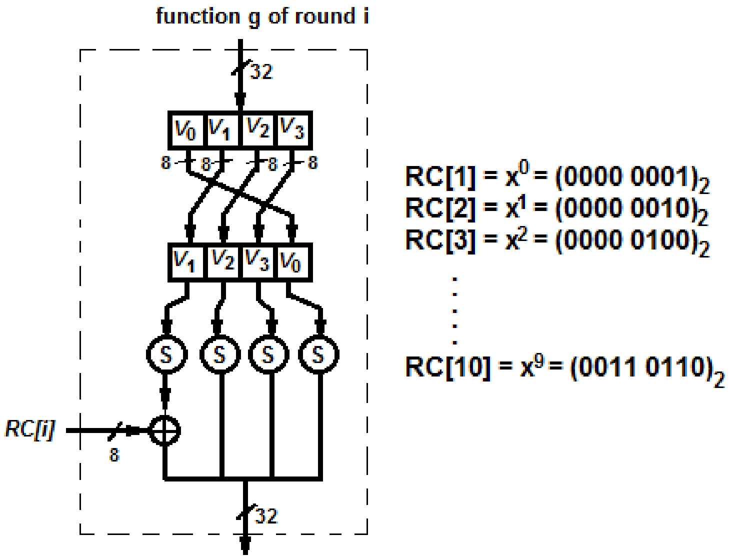

function g of round i
32
V_0 V_1 V_2 V_3
8 8 8 8
V_1 V_2 V_3 V_0
S S S S
RC[i]
8
32
RC[1] = x^0 = $(0000\ 0001)_2$
RC[2] = x^1 = $(0000\ 0010)_2$
RC[3] = x^2 = $(0000\ 0100)_2$
RC[10] = x^9 = $(0011\ 0110)_2$

Figure 1.9: Function g of Round i

1.5 Differential Electromagnetic Analysis (DEMA) Attacks on AES

The Differential Electromagnetic Analysis attack exploits the dependence of power consumption on intermediate values, which is very similar to the Differential Power Analysis attack [5]. Compared to the SEMA attack, the DEMA attack has two main advantages. First, a DEMA attack doesn't require detailed knowledge about the cryptographic device. It is usually sufficient to know the cryptographic algorithm executed by the device. Second, a DEMA attack works even if the recorded electromagnetic traces are extremely noisy. Mainly, the DEMA attack analyzes the dependence of electromagnetic radiation on the processed data at fixed moments of time. The general strategy for a DEMA attack consists of five steps.

- An intermediate result of the executed algorithm is chosen for the attack. It has to be a function of d (plaintext or ciphertext) and k (a small part of the key).
- D different data blocks are fed to the device for encryption or decryption, and the corresponding electromagnetic traces are recorded. The input data blocks are denoted as a vector $d = \{d_1, d_2 \ldots d_D\}$. The electromagnetic trace for each data block is represented as a vector $t_i = \{t_{i1}, t_{i2} \ldots t_{iT}\}$. Then, a

matrix T of dimension D by T is constructed by using the recorded electromagnetic traces.

- Hypothetical intermediate values are calculated for every possible key. The key hypotheses are represented as a vector $k = \{k_1, k_2 \ldots k_K\}$. Then, the data vector d and key vector k are used to compute intermediate values to get matrix $V = [v_{ij}]$, where $v_{ij} = f(d_i, k_j)$.

- The intermediate values in matrix V are mapped to the electromagnetic radiation values, resulting in a new matrix H of hypothetical electromagnetic radiation values. Most common methods for doing so are Hamming Weight Model, Hamming Distance Model and Zero Value Model.

- Hypothetical electromagnetic radiation values in matrix H are compared with the actual electromagnetic traces. The correlation coefficient between each column $\mathrm{h_i}$ of matrix H and each column $\mathrm{t_j}$ of matrix T are calculated and stored in a new K×T correlation coefficient matrix R. The correct key can be retrieved by looking at the index of the largest correlation coefficient value in the matrix R. The row number indicates the correct key and the column number indicates the time.

Chapter 2

Project Setting

This section discusses the experimental settings of our project, including installing the cryptography libraries, compiling the C codes, and capturing electromagnetic traces.

2.1 Overview

The block diagram of the project setting is shown in Figure 2.1. In the center, the Intel Atom Board is operated on UBUNTU 12.04.1 OS, connected to a keyboard and a monitor. These standard I/O devices assist the process of writing and compiling the code. The board itself also has two USB ports, used for copying the library installation files as well as the code from other computers. From the board, either the serial port or the LPT port is connected to a channel of the oscilloscope in order to send the trigger, which will be discussed later in Section §2.3. During the cryptography execution, the electromagnetic waves are collected by a 3-loop coil antenna, with a diameter of about 1 centimeter. The antenna plus an amplifier, which is powered 15V by power supply, can detect electromagnetic leakage traces from Intel Atom Board, and send them directly to a channel of the oscilloscope. The scope model is WavePro 725Zi from LeCroy with the bandwidth of 2.5GHz and the maximum sampling rate of 40 GS/sec [6]. Based on the trigger sent from another channel, the oscilloscope is able to capture the traces from a chosen channel at the right moment, then write to binary files, and save them to the hard disk. The trace files are then transferred to other computers via a USB drive in order to perform signal analysis in MATLAB.

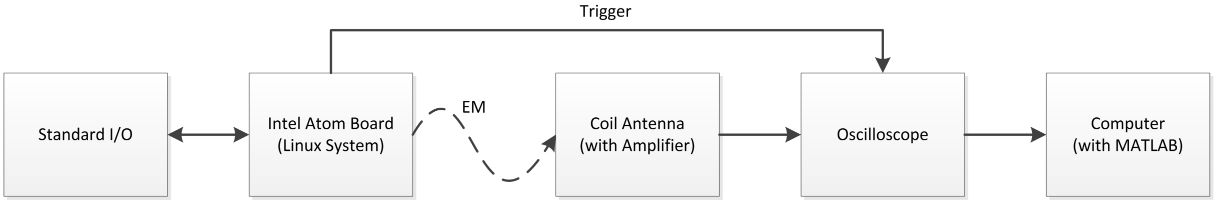


Figure 2.1: Project Setting Block Diagram

Based on the block diagram, the actual project setting is shown in Figure 2.2. The Intel Atom Board is in the middle, run on Linux system. To the left is the monitor and the keyboard, which are the standard I/O connected to the board. The antenna coil with the amplifier is placed above the board in order to capture the electromagnetic waves, and the probe is connected to the serial port, or LPT port of the board. These placements can be seen more clearly in the close-up pictures, Figure 2.3, Figure 2.4, and Figure 2.7. Based on the trigger from a channel, this oscilloscope will collect the electromagnetic traces of the Atom board from another channel and save them to the hard disk. In the back is the power rail supplying for all devices, including the monitor, the Atom board, the oscilloscope, and the power supply.

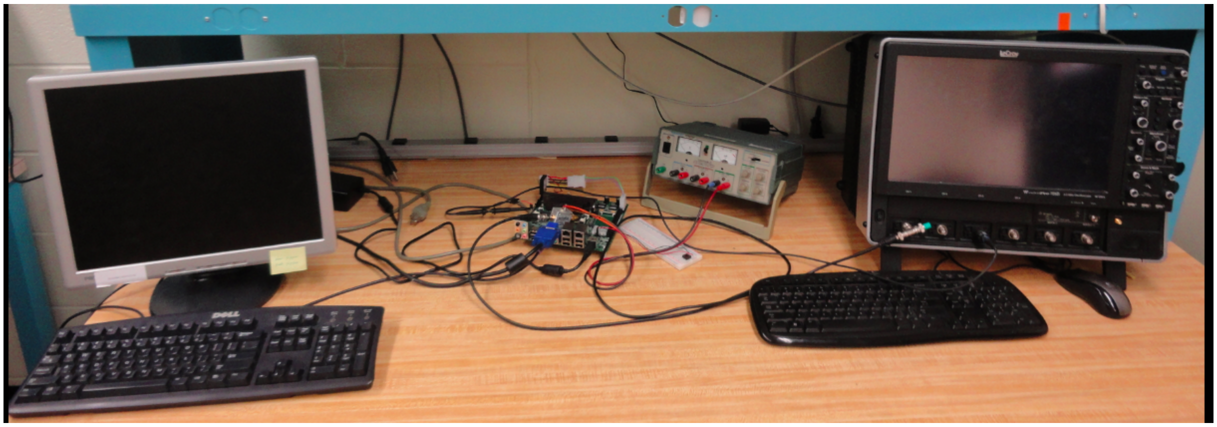


Figure 2.2: General Project Setting

A close-up view on the Intel Atom Board is shown in Figure 2.3. The board uses EMPhase S1 series flash module for memory. To the left, the oscilloscope probe is connected to the serial port in order to detect the generated trigger. In the middle, the coil antenna is placed above the board and under the heat-sink to capture the electromagnetic waves. The optimum position is chosen based on trial-and-error in order to maximize the trace amplitudes while minimize the unexpected noise from other parts. Interestingly, the optimum positions for the RSA and AES traces are different, which makes sense since these two algorithms require different operations of the system. For these two algorithms, the coil antenna measures the electromagnetic radiations from different capacitors in the board.

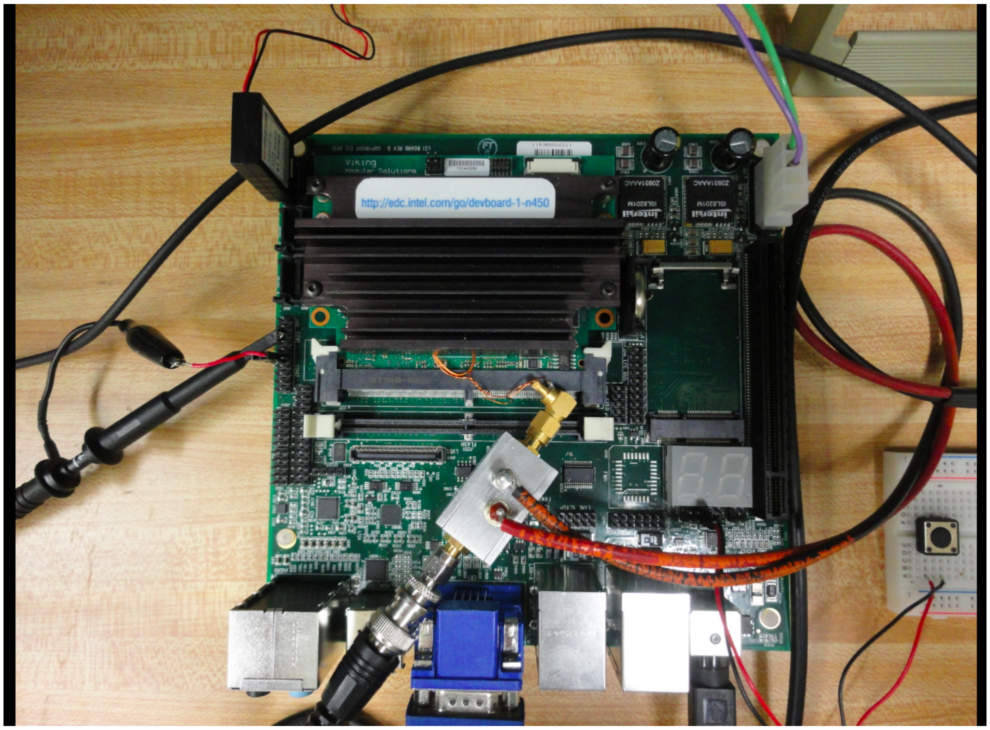


Figure 2.3: Atom Board Setup

The amplifier of the antenna coil is connected to the power supply as shown in Figure 2.4. The amplifier with the model number Mini-circuit ZFL-1000LN+, amplifies the signal inside the range 0.1 MHz to 1000 MHz, before it goes to the oscilloscope. This filter range is sufficient since the RSA and AES operations, as later described, are in the order of 10 MHz to 100 MHz. The amplifier has a minimum gain of 20dB [8]. To the right, the Atom board is connected to a simple button on a breadboard. This button is used to start up or reset the whole system when necessary.

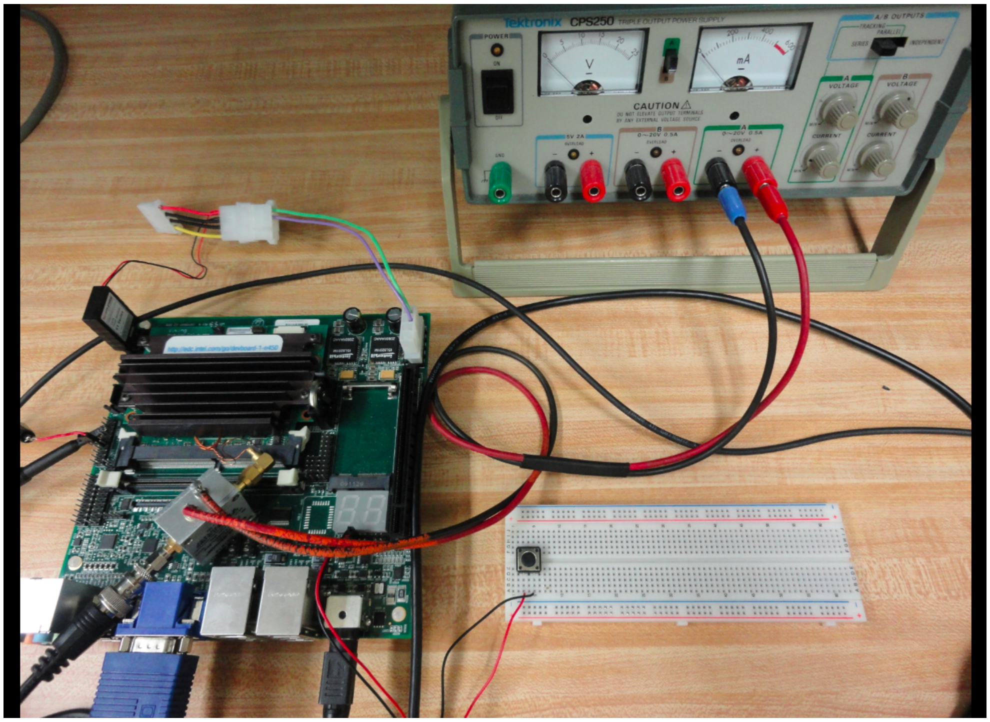


Figure 2.4: Power Supply to the Amplifier

2.2 Writing the C Code

This section explains how we installed cryptography libraries on Linux System, and how we compiled C codes to implement RSA and AES algorithms.

2.2.1 Employing the Libraries

In order to quickly perform the RSA and AES algorithms, it is convenient to employ free public libraries inside the C code. For this project, two popular libraries are chosen: the GNU Multiple Precision Arithmetic (GMP) library version 5.1.0 [13], and the OpenSSL library version 1.0.1c [9]. The GMP library is well-known for the speed when dealing with long integers (*bignum*) using highly optimized assembly language; thus, it will be used to implement the RSA algorithm, both simple multiply-and-square method and modular exponentiation method. The OpenSSL library is an open-source implementation of the SSL and TLS protocol, which implements many basic cryptography functions. Hence, the library consists of both RSA and AES implementations.

These two library packages can be downloaded from their main websites. The installation process is also simple. Here is an example of installing the OpenSSL library. The first step is to extract files from the downloaded package. The second step is to enter the directory where the package is extracted, and to run the configuration file with the default setup. Next step is to compile OpenSSL and to check for any error messages. The last step is to install the OpenSSL library with root command for privileges on destination directory.

```
tar -xvzf OpenSSL-1.0.1c.tar.gz
./config
make
sudo make install
```

Now, the library is ready to be used by including the library header in the beginning of the C code, for example:

```
#include <OpenSSL/aes.h>
```

2.2.2 Compiling the Code

Since the compiler of the system doesn't link the library automatically, we have to specify the library directory manually in each compilation. To compile the code using the GMP library, use the command:

```
gcc -I/usr/local/include test.c /usr/local/lib/libgmp.a
```

To compile the C code using the OpenSSL library, use the command:

```
gcc -I/usr/local/ssl/include test.c /usr/local/ssl/lib/libcrypto.a
```

2.3 Generating Triggers

Generating the trigger is a very important factor in the process of capturing the electromagnetic traces since the oscilloscope need to be triggered based on one of its channel inputs. For both RSA and AES implementations, the atom board first generates the trigger at one of its output port. Since there is always a rising time at the output port when going from low to high, the board is forced to sleep or to perform some delay to accommodate this latency. Then, the cryptographic algorithm is performed, followed by another delay period, either sleeping or counting. The procedure ensures that the whole cryptographic algorithm is captured in the electromagnetic trace, and reduces the interference between two consecutive operations.

2.3.1 From the Serial Port

Since the serial port is quite common in many systems and easy to be written to, we use it as the trigger when capturing the RSA electromagnetic traces. The oscilloscope probe is connected to the serial port as shown in Figure 2.5. The ground is connected to the pin 6 while the probe is connected to the pin 3 of the serial COM port.

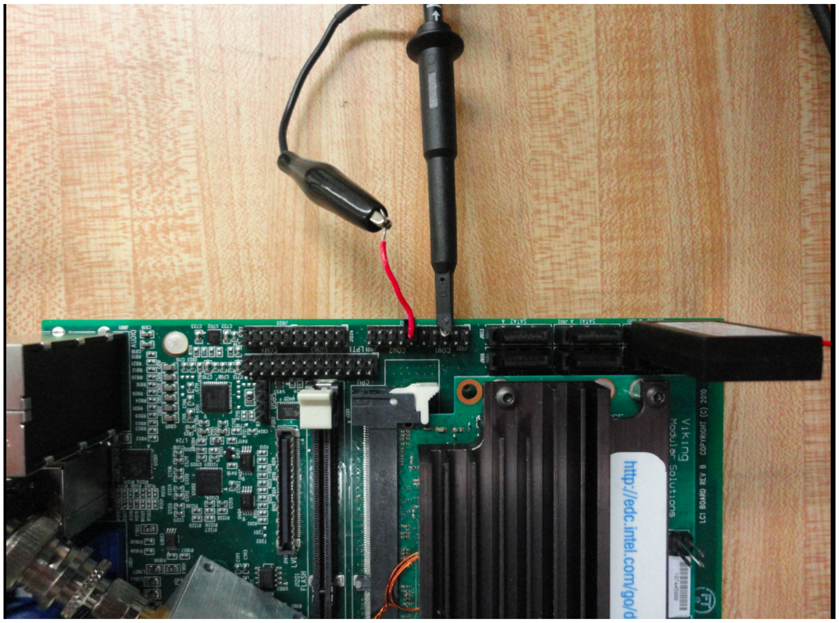


Figure 2.5: Probe at the Serial COM Port

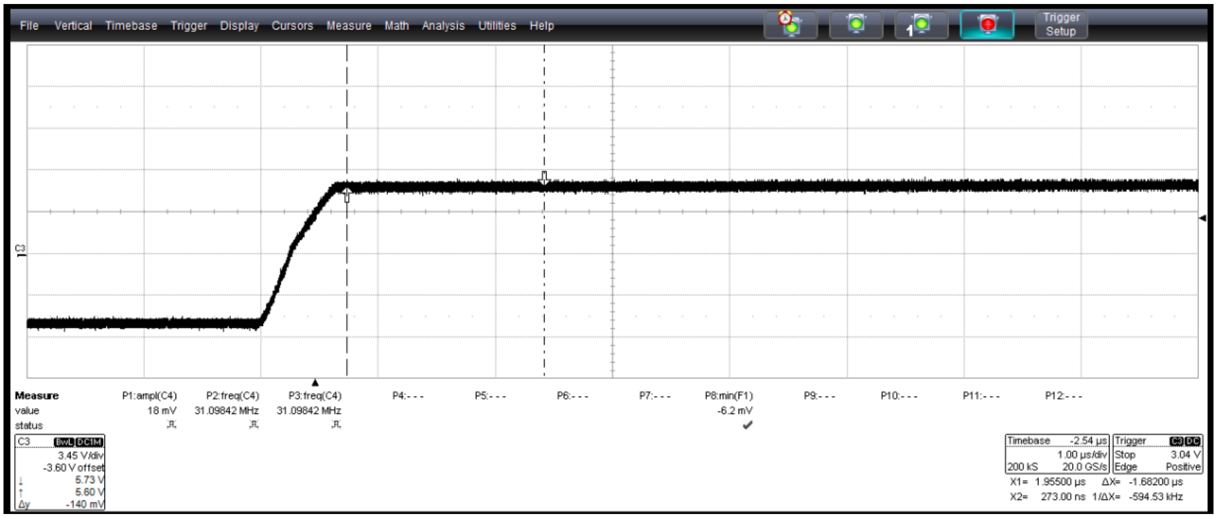
At the beginning of the code, we perform test I/O for this specific port using the command:

```
fd = open("/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
```

Next, every time a trigger is needed, we just have to write a single bit 1 to that port:

```
write(fd,tpstr,1);
```

After a finite time, the output port is back to 0; thus, we don't have to manually turn off the port before the next trigger. The trigger produced at this serial port is shown in Figure 2.6. Since the oscilloscope is set with 1µs per division, the trigger rising time is about 0.75µs, which is significantly slow. However, since the RSA algorithm itself is slow as well, in an order of 10 milliseconds, it is still acceptable to use this trigger when collecting the RSA traces.



```
Measure   P1:ampl(C4)  P2:freq(C4)    P3:freq(C4)    P4:---  P5:---  P6:---  P7:---  P8:min(F1)  P9:---  P10:---  P11:---  P12:---
value     18 mV        31.09842 MHz   31.09842 MHz                                   -6.2 mV
status                                                                                ✓

C3           BwL DC1M
             3.45 V/div
          -3.60 V offset
↓               5.73 V
↑               5.60 V
Δy            -140 mV

Timebase     -2.54 µs   Trigger      C3 DC
             1.00 µs/div  Stop       3.04 V
200 kS       20.0 GS/s  Edge      Positive
X1=  1.95500 µs     ΔX=  -1.68200 µs
X2=   273.00 ns   1/ΔX=  -594.53 kHz
```

Figure 2.6: Trigger Produced at the Serial Port

2.3.2 From the LPT Port

Since a single AES takes about 1.2 µs, the trigger from the serial port, which has a latency of about 0.75µs, is not sufficient. Instead, we need a very fast trigger to capture the AES on the oscilloscope. We researched about Intel Atom Board [3] in order to find a port that can produce a trigger with fast rising time. Finally, we came up with the conclusion of using the LPT (Parallel) port. Although LVDS (low voltage differential signal) port and LPC (low pin count) seem promising as they potentially have lower latencies, producing a trigger at those ports may be difficult due to involving low-level programming such as changing the kernel, and bios setting.

The oscilloscope probe is connected to the data pin of the LPT port as shown in Figure 2.7. The ground is still connected to the pin 6 of the serial COM port while the probe is connected to any pin of the LPT port.

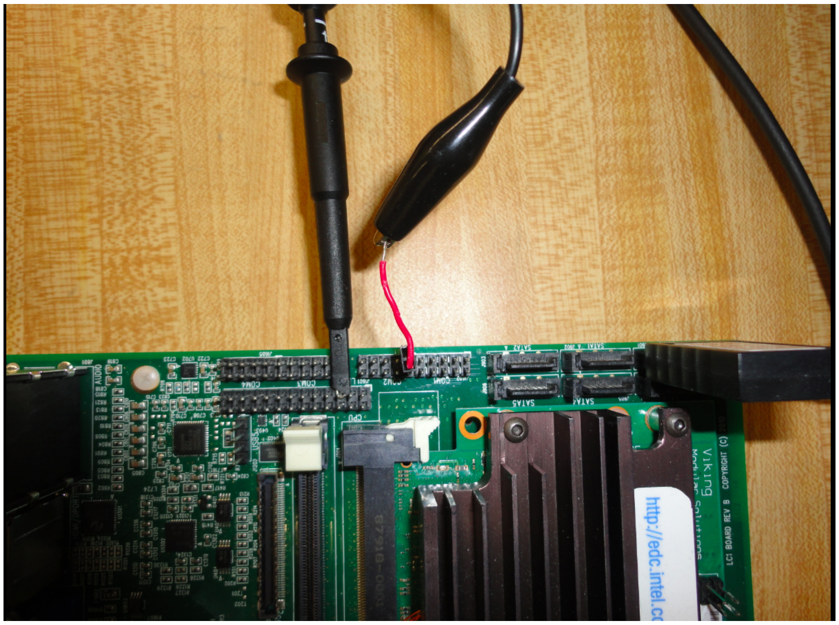


Figure 2.7: Probe at the LPT Port

To write to the data bits of the LPT port, a general procedure is implemented in C. First, the base port address is defined as below:

```
#define BASEPORT 0x378 /*lp1*/
```

Next, we have to set the permission to access the port:

```
if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}
```

Inside the main function, whenever a trigger is needed, we just have to write a sequence of 1 to the chosen port:

```
    outb(255, BASEPORT);
```

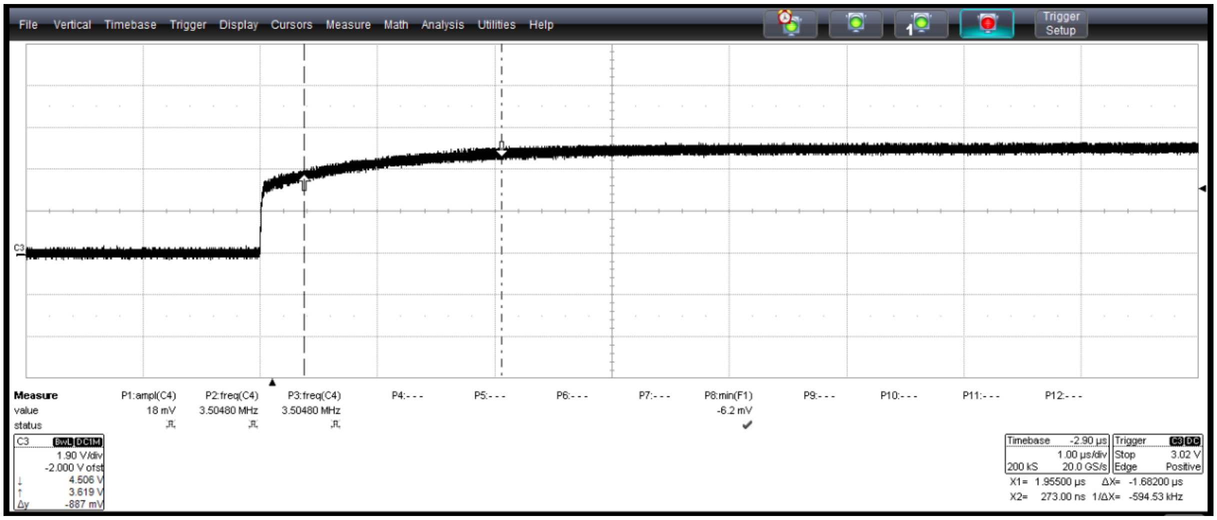
However, since the LPT port doesn't go back to 0 after a finite time, we have to manually set it off:

```
    outb(0, BASEPORT);
```

Finally, for our own safety, we have to take away the permission to access the LPT port at the end of main function in case other codes use the port for different purposes.

```
    if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}
```

The trigger produced at this LPT port is shown in Figure 2.8. The oscilloscope is still set with 1µs per division, and as seen, the slope now is much steeper. The latency is only about 0.05µs, which is acceptable to use as the trigger when collecting the AES traces.



Measure	P1:ampl(C4)	P2:freq(C4)	P3:freq(C4)	P4:- - -	P5:- - -	P6:- - -	P7:- - -	P8:min(F1)	P9:- - -	P10:- - -	P11:- - -	P12:- - -
value	18 mV	3.50480 MHz	3.50480 MHz					-6.2 mV				
status								✓				

C3 BwL DC1M
1.90 V/div
-2.000 V ofst
↓ 4.506 V
↑ 3.619 V
Δy -887 mV

Timebase -2.90 µs
1.00 µs/div
200 kS 20.0 GS/s

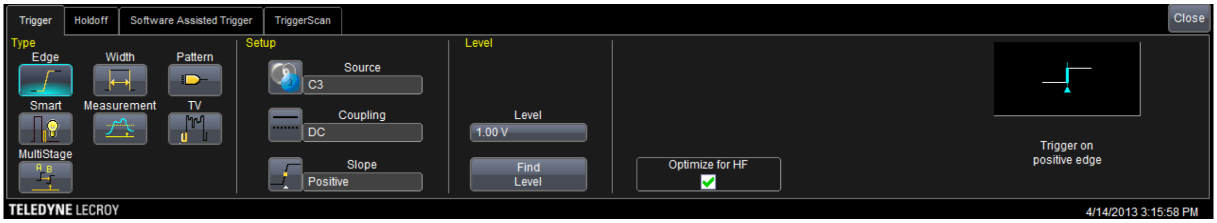
Trigger C3 DC
Stop 3.02 V
Edge Positive

X1= 1.95500 µs ΔX= -1.68200 µs
X2= 273.00 ns 1/ΔX= -594.53 kHz

Figure 2.8: Trigger Produced at LPT port

2.4 Capturing the Traces

In order to successfully capture the interested electromagnetic trace, the scope setup has to be strictly followed. First step is to set up the trigger signal, which can be in any channel. In this case, we chose channel 3, with a positive edge trigger. Detail options are shown in Figure 2.9.



Setup

Source: C3

Coupling: DC

Slope: Positive

Level

Level: 1.00 V

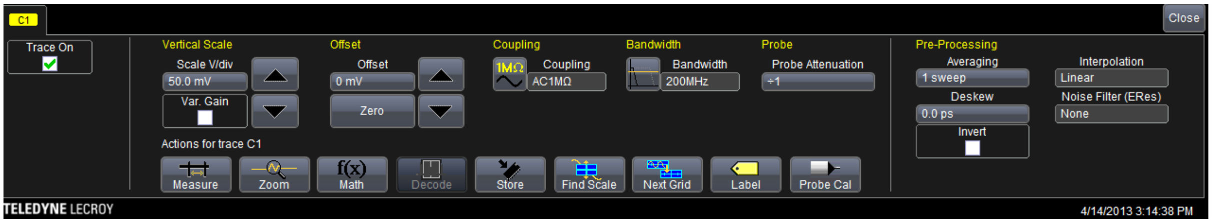
Optimize for HF ☑

Trigger on positive edge

4/14/2013 3:15:58 PM

Figure 2.9: Trigger Channel Setup

Next step is to set up the trace signal, which in this case is chosen to be channel 1. The channel is coupling with AC1MΩ, and the bandwidth is set to 200 MHz, which is enough to cover the RSA and AES operations.



Trace On ☑

Vertical Scale

Scale V/div: 50.0 mV

Var. Gain □

Offset

Offset: 0 mV

Coupling

Coupling: AC1MΩ

Bandwidth

Bandwidth: 200MHz

Probe

Probe Attenuation: ÷1

Pre-Processing

Averaging: 1 sweep

Deskew: 0.0 ps

Invert □

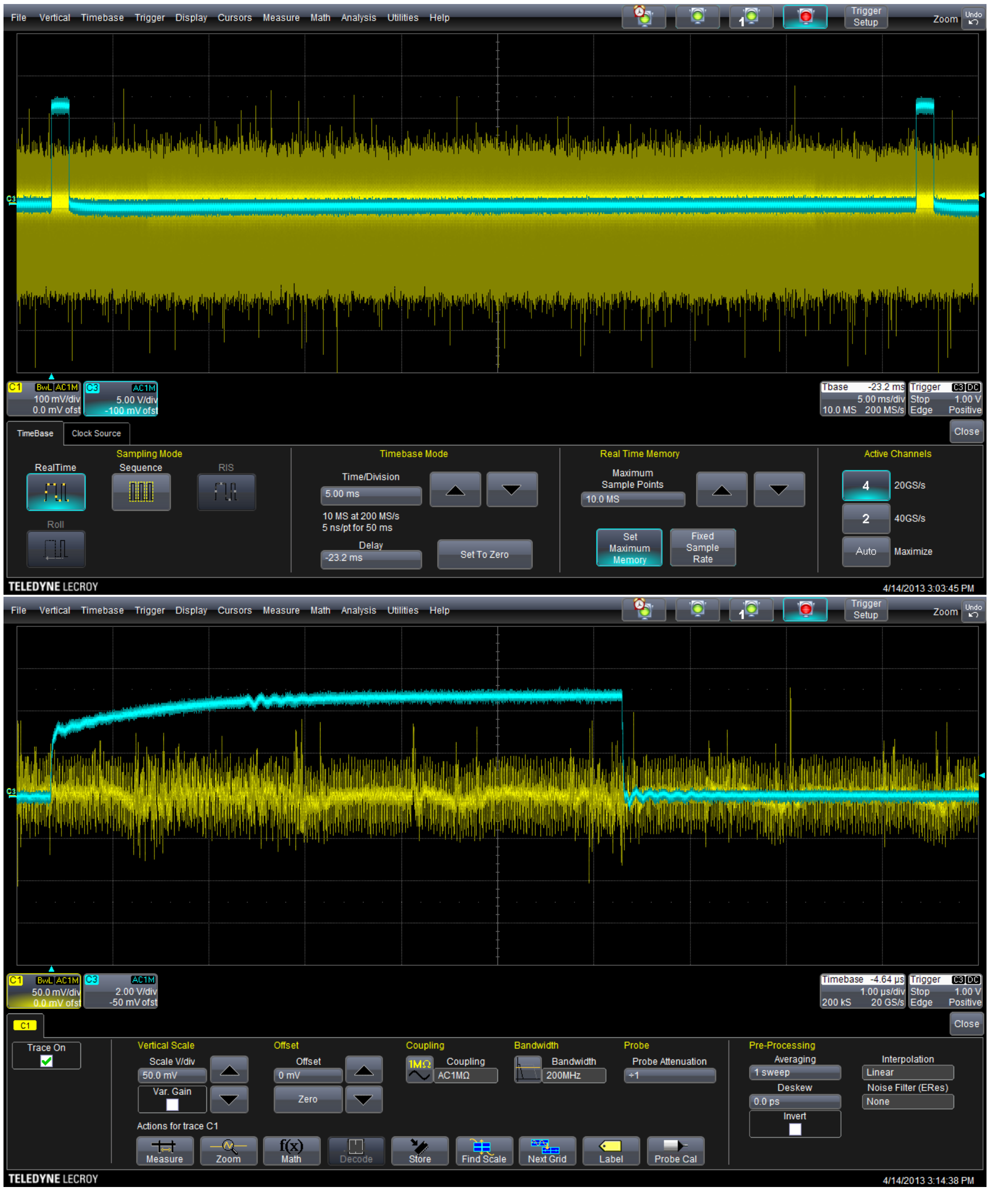
Interpolation: Linear

Noise Filter (ERes): None

4/14/2013 3:14:38 PM

Figure 2.10: Trace Channel Setup

The final step is to set up the appropriate sampling rate in order to capture the whole operation in a single trace. For example, for the RSA 1024-bit key, the sampling rate needs to be 200MS/sec. For shorter or longer key length, the sampling rate would be modified accordingly to accomodate the new RSA operation. Sine the AES operations are really fast, the sampling rate can be set to the maximum of 20GS/sec. The location of the coil attena is chosen to minimize the noises inside each trace. Examples of capturing the RSA and AES traces are shown in Figure 2.11. Finally, press Save to record the wanted traces into binary files, and to later use them for the MATLAB analysis.



C1 BwL AC1M 100 mV/div 0.0 mV ofst

C3 AC1M 5.00 V/div -100 mV ofst

Tbase -23.2 ms 5.00 ms/div 10.0 MS 200 MS/s

Trigger C3 DC Stop 1.00 V Edge Positive

Time/Division 5.00 ms

10 MS at 200 MS/s
5 ns/pt for 50 ms

Delay -23.2 ms

Maximum Sample Points 10.0 MS

4 20GS/s
2 40GS/s
Auto Maximize

4/14/2013 3:03:45 PM

C1 BwL AC1M 50.0 mV/div 0.0 mV ofst

C3 AC1M 2.00 V/div -50 mV ofst

Timebase -4.64 µs 1.00 µs/div 200 kS 20 GS/s

Trigger C3 DC Stop 1.00 V Edge Positive

Trace On ☑

Vertical Scale: Scale V/div 50.0 mV; Var. Gain □

Offset: Offset 0 mV; Zero

Coupling: 1MΩ Coupling AC1MΩ

Bandwidth: Bandwidth 200MHz

Probe: Probe Attenuation ÷1

Pre-Processing: Averaging 1 sweep; Deskew 0.0 ps; Invert □; Interpolation Linear; Noise Filter (ERes) None

4/14/2013 3:14:38 PM

Figure 2.11: Oscilloscope sampling setup for RSA (top) and AES (bottom)

Chapter 3

Attacking RSA

3.1 General Procedure of Analyzing RSA Electromagnetic Traces

This section explains the standard procedure of collecting, filtering, and processing traces to clear unwanted noises and other operations of the system.

3.1.1 Collecting the Traces

As stated, we first set a trigger at the serial port, perform a sleep command, run one RSA operation, and then another sleep command. As the Atom Board executes this procedure over and over again in a loop, the oscilloscope records the traces based on the intended trigger. For each trace, general information of the key, the crypto algorithm, the library, and the scope setting are also recorded. An example trace is shown in Figure 3.1.

- Trace name: sim80_2.5G_2.trc
- RSA Key: 80 bits (*FFFF0000FFFF0000FFFF*)
- RSA algorithm: Simple multiply-and-square, GMP library
- Sampling frequency: 2.5 GS/sec

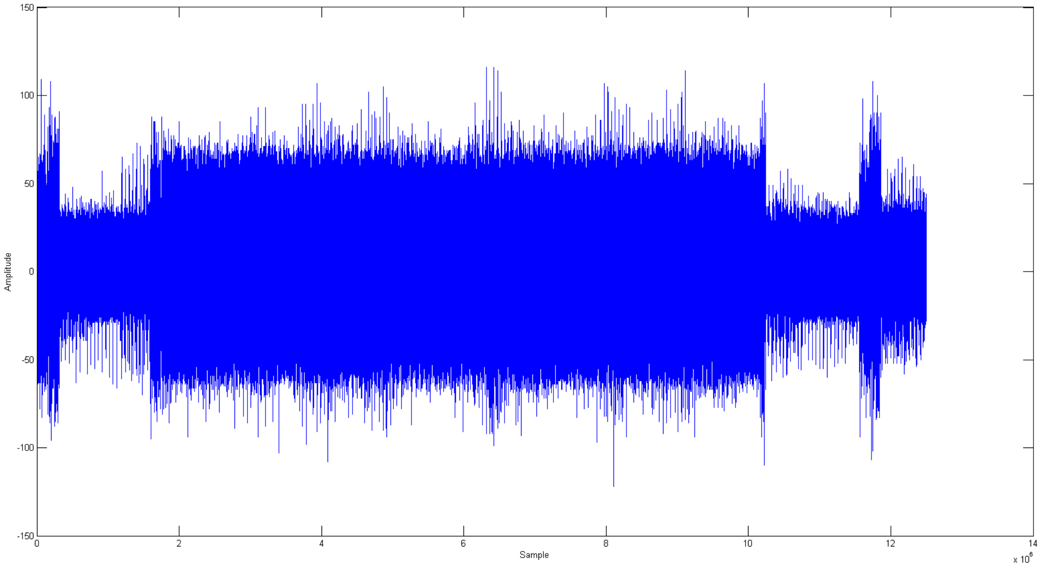


Figure 3.1: Original RSA Trace with 80 bit exponent (sim80_2.5G_2.trc)

Despite the noises, we can see the RSA operation in the middle of the trace, represented by the region with higher amplitude. The two regions with lower amplitude are when the board is performing the sleep commands. However, we cannot see any pattern of multiply and square operations in this raw trace; thus, filtering techniques are required in order to clear out unexpected factors from this trace.

3.1.2 Filtering Technique

Our SEMA attack on RSA utilizes two filters, and now we will describe the concept behind these filters in details. Since there are a lot of noises as well as other system operations running during the cryptography execution, an appropriate band-pass is employed to amplify only the signal inside the range of the interested operations, i.e. RSA or AES. This would be the first filter applied to the collected traces.

The electromagnetic radiation of the target device could be given as $p(t) = P_{const} + p_{dyn}(t)$ where P_{const} is the constant part and $p_{dyn}(t)$ is the dynamic part caused by internal operations. Usually, the dynamic portion is much weaker than the constant part. Thus, the possibility of exploiting leakage heavily depends on the quality of the isolation of $p_{dyn}(t)$.

Since the amplitude of the signal is modulated as $s(t) = p(t)\cos(\omega_r t)$ where ω_r is the carrier frequency, the extraction of $p(t)$ or the weak dynamic portion $p_{dyn}(t)$ could be done using amplitude demodulation. In this project, a very common incoherent technique is employed, which based on rectification, or often called envelope detection. Here is a quick explanation of the method. Let first denote the frequency domain representation of the original signal as $A(j\omega) = \mathscr{F}\{p(t)\}$. Now the spectrum of the rectified signal could be computed [11].

$$\mathscr{F}\{|s(t)|\} = \mathscr{F}\{p(t)\,|\cos(\omega_r t)|\} = \mathscr{F}\left\{p(t)\frac{2}{\pi}\sum_{\nu=-\infty}^{\infty}\frac{(-1)^{\nu}}{1-4\nu^2}e^{j2\nu\omega_r t}\right\}$$

$$\mathscr{F}\{|s(t)|\} = \frac{2}{\pi}\sum_{\nu=-\infty}^{\infty}\frac{(-1)^{\nu}}{1-4\nu^2}\mathscr{F}\left\{p(t)e^{j2\nu\omega_r t}\right\} = \frac{2}{\pi}\sum_{\nu=-\infty}^{\infty}\frac{(-1)^{\nu}}{1-4\nu^2}A(j\omega - j2\nu\omega_r)$$

Thus, the rectified signal is essentially similar to the spectrum of $p(t)$, which however is scaled and repeated at all even multiples of the carrier frequency. Finally, an appropriate band-pass filter at low frequency is employed to isolate the desire signal $p(t)$.

3.1.3 Filtering the Traces in MATLAB

The first step is to plot the spectrogram of the electromagnetic trace, as shown in Figure 3.2, in order to find which frequency range the RSA operations execute at. The *spectrogram* function is available in the MATLAB Signal Processing Toolbox, and the detail script of plotting the spectrogram can be found in the Appendix. Since we add two sleep commands at the beginning and at the end, the RSA operation is expected to stand out clearly.

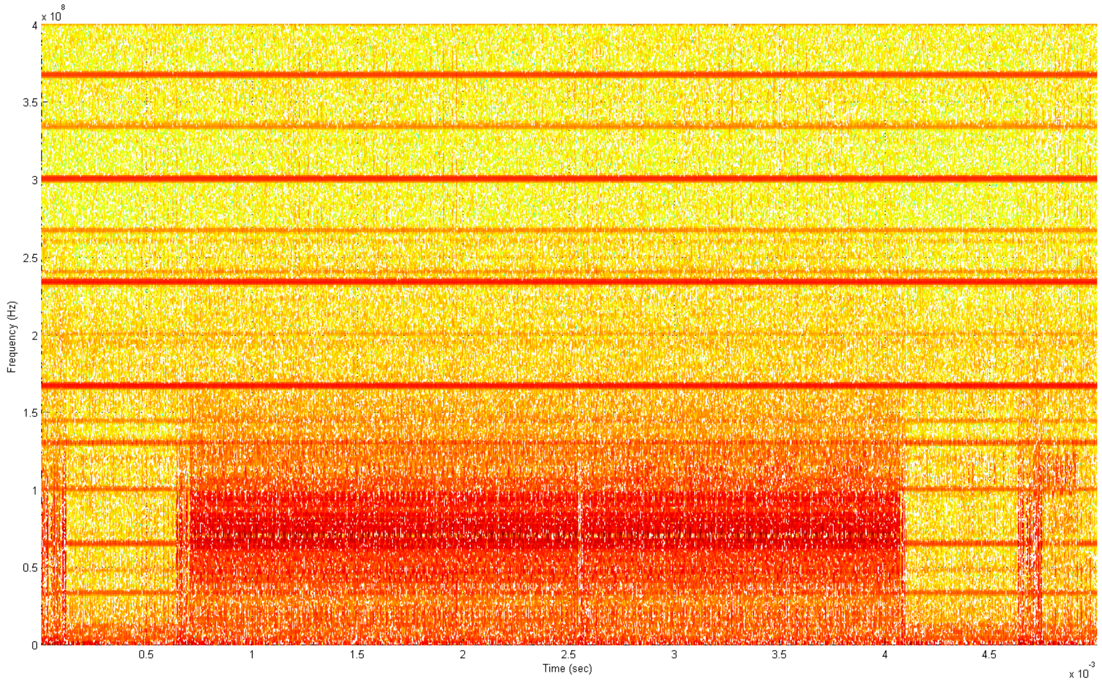


Figure 3.2: Spectrogram of the original trace (sim80_2.5G_2.trc)

As seen, the x-axis is the time in second, the y-axis is the frequency in Hz, and the color at each point represents the amplitude of the specific frequency at the specific time. Based on the spectrogram, we can observe several basic operations of the system, which can be seen as continuous red stripes from the beginning until the end. In other hand, since our RSA operation is set between two sleep commands, it would stand out as a single red stripe but be truncated at the two ends. Applying this concept to the current spectrogram, the RSA operation is the red stripe ranging from 50 MHz to 80 MHz. Moreover, noise can be seen as red spots appearing randomly in the spectrogram. Therefore, the first bandpass filter is chosen to be from 50 MHz to 80 MHz. This filter will attenuate the undesired noise as well as other operations performed by the Atom board during the experiment. The trace after the first filter is shown in Figure 3.3.

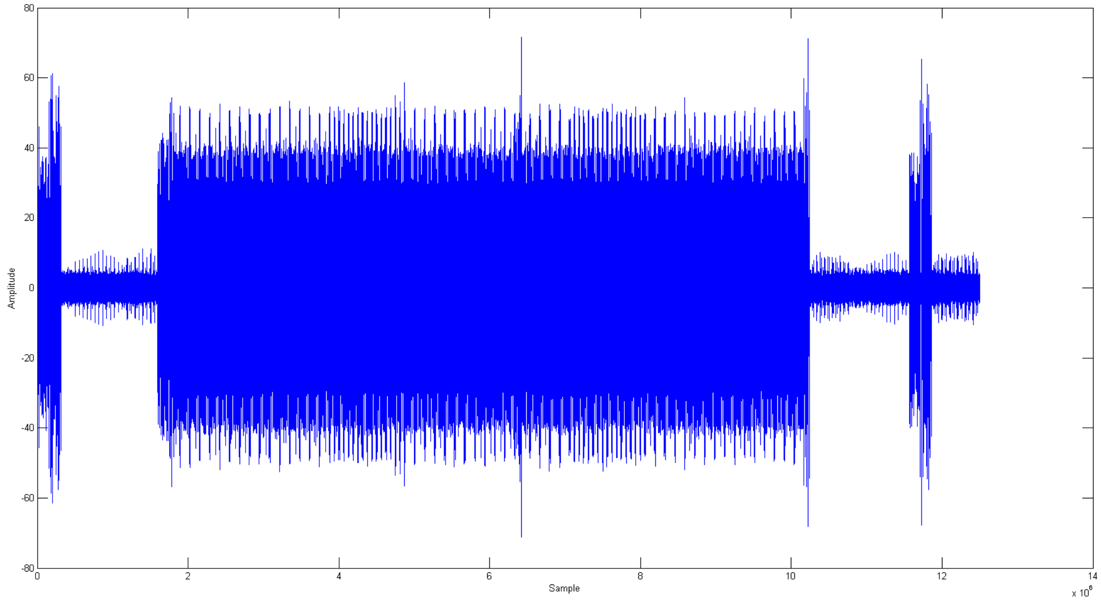


Figure 3.3: Trace after the 1st filter (sim80_2.5G_2.trc)

As expected, the first filter successfully reduces the noise and other operations of the system. The trace amplitude during RSA operation is much higher than the amplitude during sleep time. Also, the overall amplitude is more uniform. However, as mentioned in 3.1.2, the weak dynamic portion $\mathrm{p_{dyn}(t)}$ has to be extracted using amplitude demodulation in order to exploit the information leakage. Therefore, the next step is the envelope detection based on rectification, which removes the carrier frequency from the current trace. After that, the multiply and square operations can be seen and differentiated much more easily. Unfortunately, since the carrier frequency is unknown, we first have to take the absolute value of the trace, which shifts the spectrum center to DC, and then plot the new spectrum in order to obtain some insights about the carrier frequency. The new spectrum is shown in Figure 3.4.

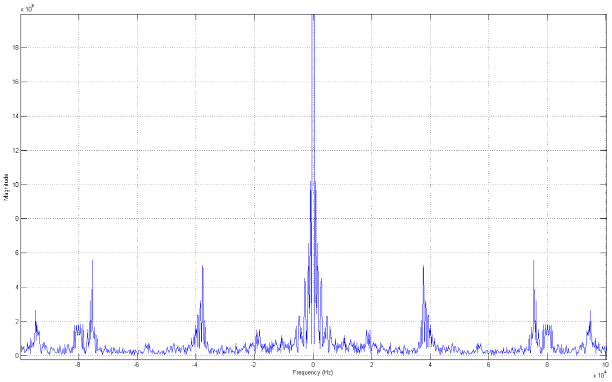


Figure 3.4: Spectrum of the rectified trace (sim80_2.5G_2.trc)

The spectrum of the rectified trace consists of a huge spike at DC and several smaller spikes around; one of them would be the carrier frequency we are looking for. By trial-and-error, we find out the closest spikes to DC, near 40 kHz, is the interested one. Thus, a second bandpass filter is set from 20 kHz to 40 kHz in order to remove the carrier frequency, and the new trace is shown in Figure 3.5.

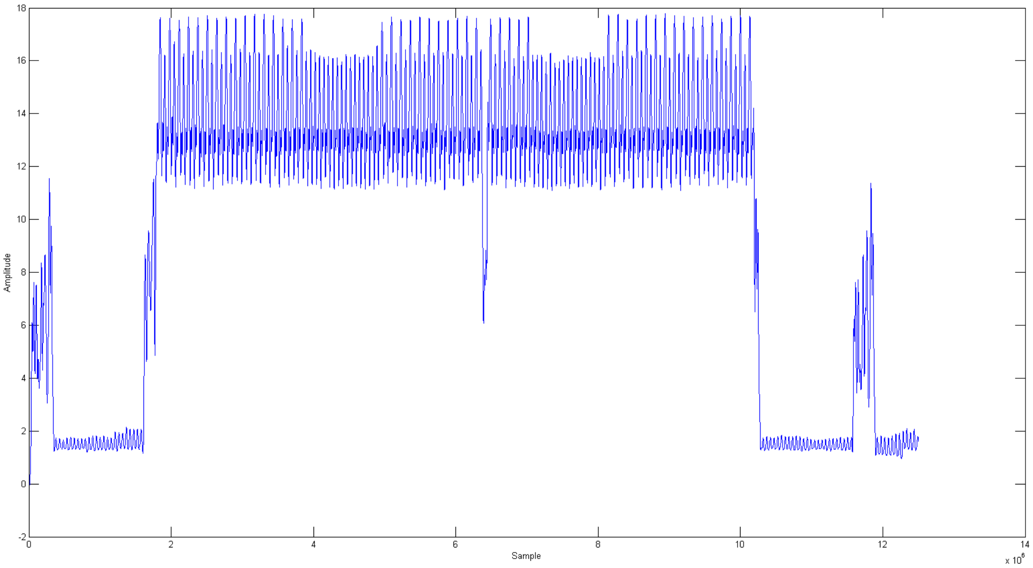


Figure 3.5: Trace after the 2nd filter (sim80_2.5G_2.trc)

3.1.4 Finding the Key

After the envelope detection process, each operation can be visually identified clearly. In this case, the upper peaks are different for the multiply and square operations, and will become the distinguisher. A longer peak represents a multiply operation while a shorter one represents a square operation. It is convenient to do the differentiation procedure automatically in MATLAB. First, we have to cut out the RSA part, perform the peak detection, and then use clustering method to separate between multiply and square operations. For visualization purpose, we draw letter M in red representing a multiply, and letter S in green representing a square near each peak. The processed trace with M and S displayed is shown in Figure 3.6.

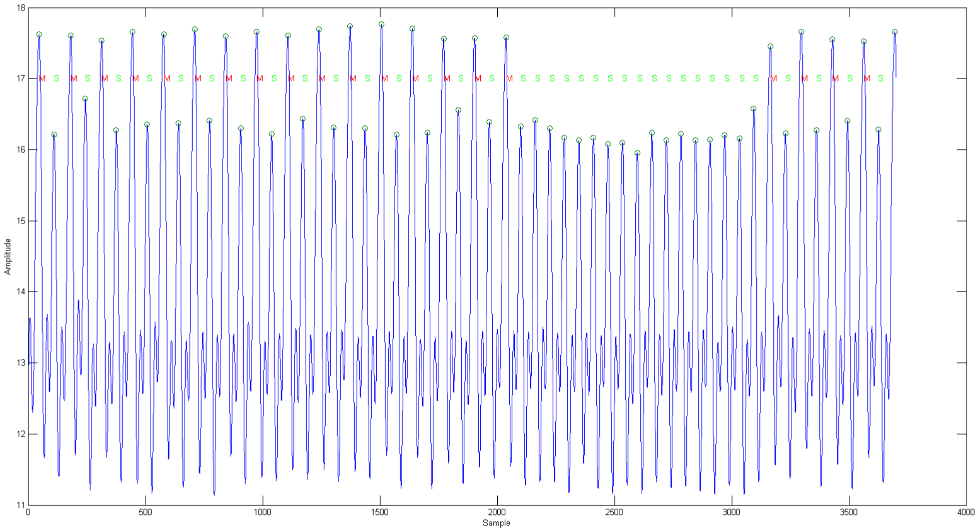


Figure 3.6: Automatically distinguish between MUL and SQ

Finally, the sequence multiply and square can be converted to the RSA key of interest. Since this trace employs the multiply-and-square method, a square followed by of multiply corresponds to a single bit 1, while a single square corresponds to a single bit 0. Again, a MATLAB script is deployed to automatically convert the multiply-and-square sequence to a binary key. The script returns the key *FFFF0000FFFF0000FFFF* in hexadecimal. Knowing the original key, we can then evaluate the performance of our method based on the bit error rate, which is computed by XOR-ing the received key with the original one. In this case, the bit error rate is zero; we successfully retrieve the secret key from the collected electromagnetic trace.

3.2 Simple Multiply-and-Square RSA

In this part, we employ the C code that implements a simple multiply-and-square RSA algorithm using the GMP library. Basically, the board scans the bits of the exponent from the left (MSB) to the right (LSB). In every iteration, it squares the current result. If and only if the current bit is 1, it multiplies the current result by the base. The complete C code can be found under the Appendix. Here we use a 1024 bit key, a standard key length for RSA. Using the trigger from the serial port, we can collect the electromagnetic traces and then start the analysis. Below is the general information of the trace, and the trace itself is shown in the top part of Figure 3.7.

- Trace name: sim1024_250M_2.trc
- RSA Key: 1024 bits {*FA0BFC0DFE09F807F60543210*(x10)*F*(x6)}
- RSA algorithm: Simple multiply-and-square, GMP library
- Sampling frequency: 250 MS/sec

Following the general filtering procedure, we first plot the spectrogram of the received electromagnetic trace to determine the frequency range of RSA algorithm. As usual, the RSA operation stands out clearly compared to the sleep time, ranging from 50 MHz to 80 MHz. Thus, we use these parameters as the first bandpass filter for the received trace. The result trace is shown in the middle part of Figure 3.7.

As seen, the trace is now much clearer and more uniform; the amplitude during RSA operation is much higher than the amplitude during sleep time. Next step is envelope detection, which removes the carrier frequency from the electromagnetic trace. We have to rectify the trace and then plot the spectrum to find the frequency of interest. Similarly, the spikes near 40 kHz is the one we are looking for. Thus, the second bandpass filter is chosen from 20 kHz to 40 kHz, the result trace after this filter is shown in the bottom part of Figure 3.7.

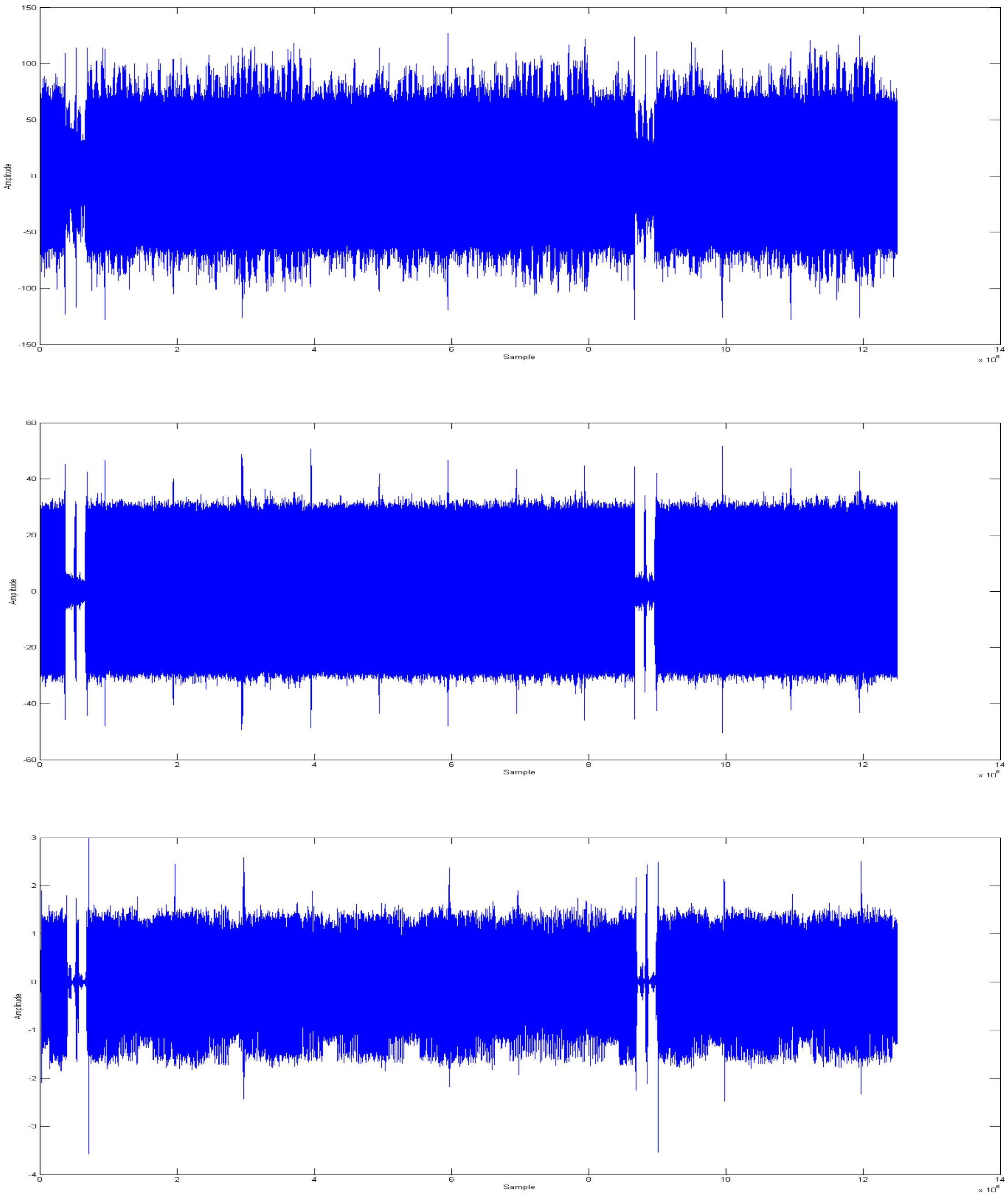


Figure 3.7: Simple RSA 1024 bit traces: Top - Raw trace. Middle - After the 1st filter. Bottom - After the 2nd filter (sim1024_250M_2.trc)

The multiply and square operations are now much clearer. In this case, the lower peaks are different for the multiply and square operations, and will become the distinguisher. A deeper peak represents a multiply operation while a shorter one represents a square operation. It is convenient to do the differentiation procedure automatically in MATLAB. First, we have to cut out the RSA part, perform the peak detection, and then use clustering method to separate between multiply and square operation. The detail MATLAB script can be found in the Appendix. The peak heights are shown in the scatter plot, Figure 3.8. With clustering method, the script automatically separates the multiply operations in red and the square operations in blue.

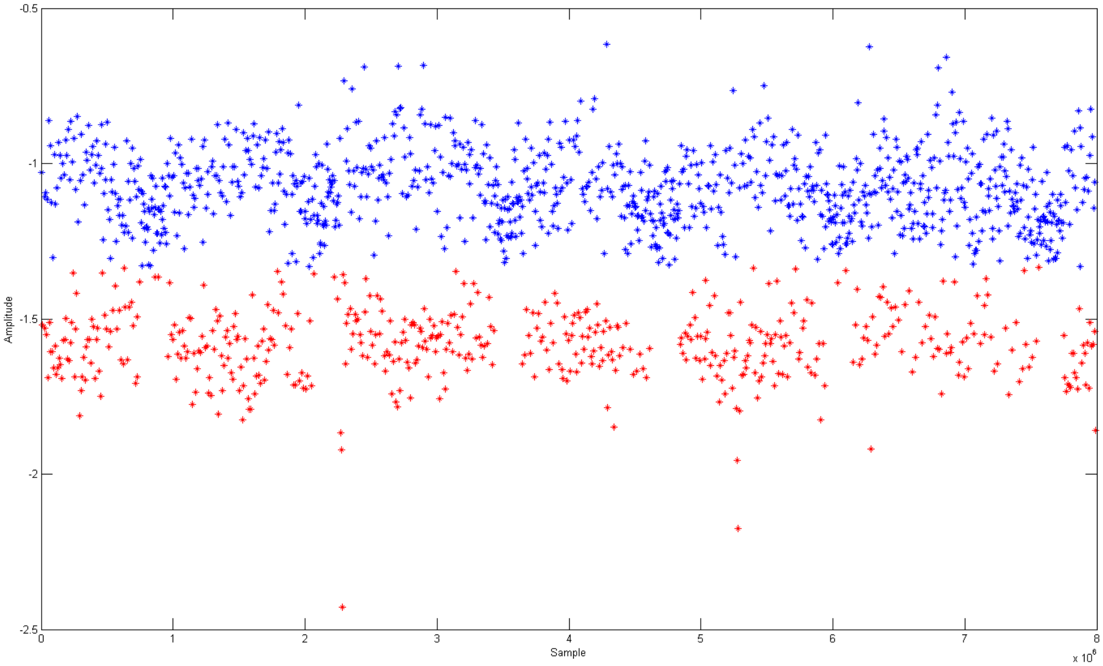


Figure 3.8: Distinguish between MUL (red) and SQ (blue)

Next, the sequence of multiply and square operations can be converted to the RSA key of interest. Since this trace employs the multiply-and-square method, a square followed by of multiply corresponds to a single bit 1, while a single square corresponds to a single bit 0. Again, a MATLAB script is deployed to automatically convert the multiply-and-square sequence to a binary key. Then, the reconstructed key is plotted with red stars together with the original key, which is plotted in blue lines, as shown in Figure 3.9.

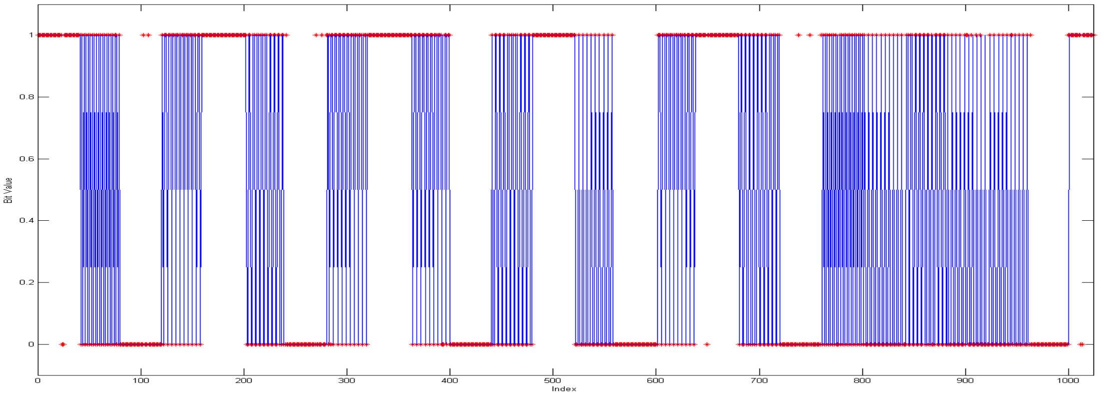


Figure 3.9: Reconstructed key vs. Original key(sim256_250M_2.trc)

As seen, the reconstructed key matches with the original one really well. However, we have to make several small offset adjustments due to the sliding between 0 and 1 since bit 0 corresponds to a single operation while bit 1 corresponds to two consecutive operations. The bit error rate is pretty small, only about 4%, which mostly came from the noise and the unexpected behaviors of the board. Performing the attack several times and calculating the key based on the majority rule can help us to improve the bit error rate.

3.3 Modular Exponentiation (RSA) - GMP Library

In this part, we perform an attack on the RSA algorithm of the GMP library, in particular, the modular exponentiation function. Instead of the simple multiply-and-square method, this library employs the sliding window algorithm. Generally, sliding window exponentiation computes the power by looking at a number of exponent bits at a time, called the *window size*. Different from the simple method, the sliding window method requires pre-computation of several values.

First step is to compute x^2 from the plain text x. Next, the system consecutively computes x^3, x^5, up until x^{2k-1} where k is the window size. These values will be used for the multiplications in the next part. Similar to the simple multiply-and-square method, the exponent bits are scanned from left to right. If the bit is 0, the current value is squared. Otherwise, if the bit is 1, the system finds the longest bit-string whose length is less than or equal to the window size, and the ending-bit is also 1. This bit-string will equal to one of the pre-computed values from the previous part; and the current value is multiplied directly with this bit-string. After scanning all the exponent bits, the final exponentiation is returned.

Despite the extra time at the beginning due to the pre-computations, the sliding window exponentiation is much faster the simple one. Multiplying with a stored bit-string instead of a base only saves a lot of computation time. The goal of our attack is still to distinguish between multiply and square operations; however, we might have to make an additional effort to separate between multiplying of different bit-string values. Otherwise, there would be several unknown bits inside the reconstructed key. For each multiplication, only the first bit and the last bit in the window is ensured to equal 1, while the middle bits could either be 0 or 1.

3.3.1 Different Key Sizes

As expected, the GMP library employs different window sizes for different RSA key lengths. In this section, we explore the window size as well as the number of pre-computations as a function of the key length. For the convenience, the key is chosen to be consisting of just *FFFFF* and *00000* in order to quickly identify the window size on the filtered traces. A sequence of consecutive 1's makes sure the system to always use the maximum of its window size; and a sequence of consecutive 0's helps to distinguish the multiply and square operations. The same two-stage filtering process is applied for all the collected traces in order to remove the unwanted factors such as noise, other operations, and carrier frequency. After that, we have to look into the source code to verify our window size result.

3.3.1.1 80-bit key

We start with a 80-bit key RSA, and the general information of the trace is shown below. Similarly, the two-stage filter is applied and the result trace is shown in Figure 3.10. Since this is a short key, a small window size and a few number of pre-computations are expected.

- Trace name: gmp80_2.5G_1.trc
- RSA Key: 80 bits {*FFFF0000FFFF0000FFFF*}
- RSA algorithm: Sliding window, GMP library
- Sampling frequency: 2.5 GS/sec

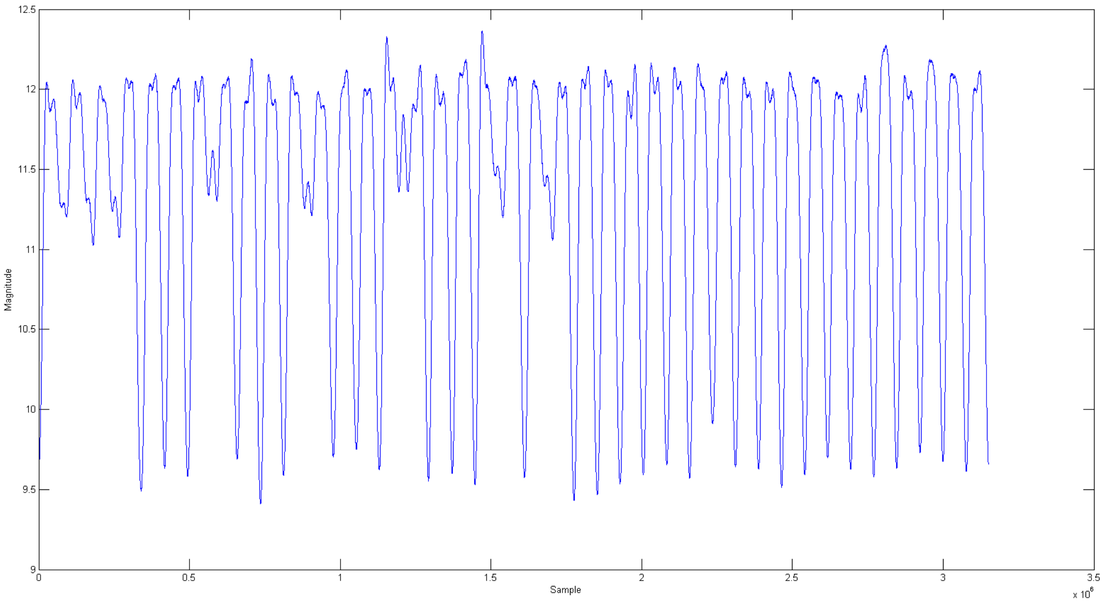


Figure 3.10: 80-bit key RSA filtered trace - GMP library

Based on the sequence of 0's, we conclude that the lower peaks are the distinguisher between different operations. A deeper spike represents a square operation while a shorter one represents a multiply operation. During the sequence of 1's, we can see three square operations followed by a multiply operation; thus, the window size is 3. In the beginning of the traces, there are 3 multiply operations, which is unusual since they never stand next to each other due to the algorithm. However, these are the 3 pre-computations, calculating x^3, x^5, up until x^7.

3.3.1.2 200-bit key

This part deals with a 200-bit key RSA, with the general trace information shown below. Similarly, the two-stage filter is employed, and the result trace is shown in Figure 3.11. Since this is a longer key than before, we expect a larger window size and a more number of pre-computations at the beginning.

- Trace name: gmp200_1G_1.trc
- RSA Key: 200 bits {*FFFFF00000FFFFF00000...*}
- RSA algorithm: Sliding window, GMP library
- Sampling frequency: 1 GS/sec

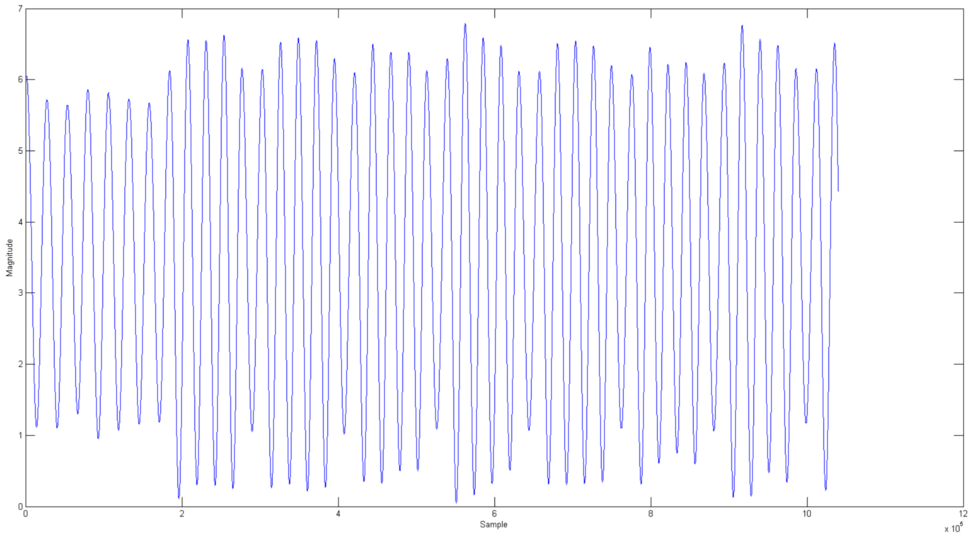


Figure 3.11: 200-bit key RSA filtered trace - GMP library

Based on the trace structure, we know that the lower peaks are still the distinguisher between different operations. A deeper spike represents a square operation while a shorter one represents a multiply operation. However, the differences are not as clear as in the case of 80-bit key. During the sequence of 1's, we can see four square operations followed by a multiply operation; thus, the window size is 4. In the beginning of the traces, there are 7 multiply operations, and these are the 7 pre-computations, calculating x^3, x^5, up until x^{15}.

3.3.1.3 400-bit key

This part explores a 400-bit key RSA, with the general trace information shown below. Again, the two-stage filter is used to remove unwanted factors, and the result trace is shown in Figure 3.12. Since this is an even longer key than before, a larger window size and a more number of pre-computations at the beginning are a reasonable assumption.

- Trace name: gmp400_1G_1.trc
- RSA Key: 400 bits {*FFFFF00000FFFFF00000...*}
- RSA algorithm: Sliding window, GMP library
- Sampling frequency: 1 GS/sec

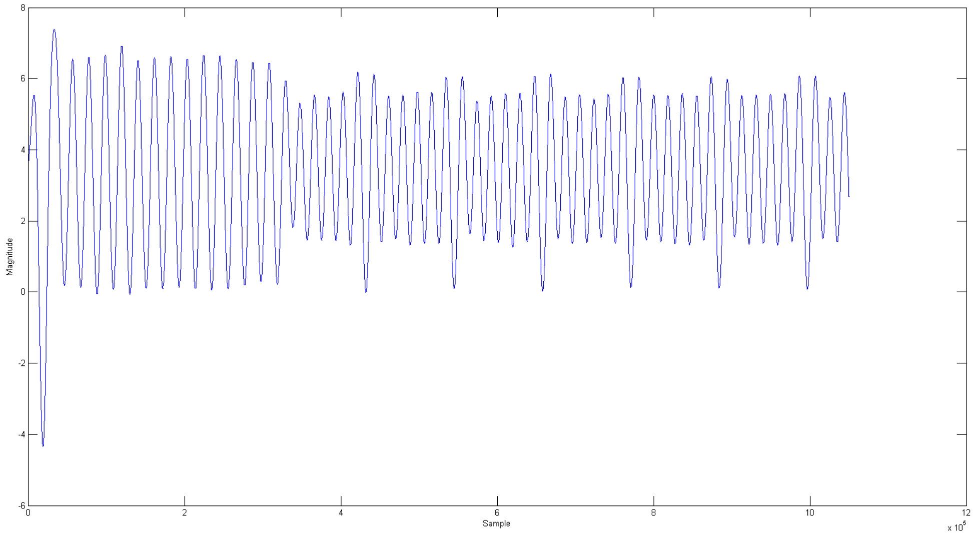


Figure 3.12: 400-bit key RSA filtered trace - GMP library

Exploring the trace structure, we conclude that the lower peaks are still the distinguisher between different operations. However, a deeper spike now represents a multiply operation while a shorter one represents a square operation. The difference might be resulted from the parameter of the two filters, or from the different sampling rate of the oscilloscope. Moreover, the differences are as clear as in the case of 80-bit key. During the sequence of 1's, we can see five square operations followed by a multiply operation; thus, the window size is 5. In the beginning of the traces, there are 15 pre-computations, calculating x^3, x^5, up until x^{31}.

3.3.1.4 1024-bit key

Finally, we look at a 1024-bit key RSA, which is a very common key length, and the general trace information is shown below. Again, the two-stage filter is used to remove unwanted factors, and the result trace is shown in Figure 3.13. Since this is a really long key, a large window size and a big number of pre-computations at the beginning will be our assumption.

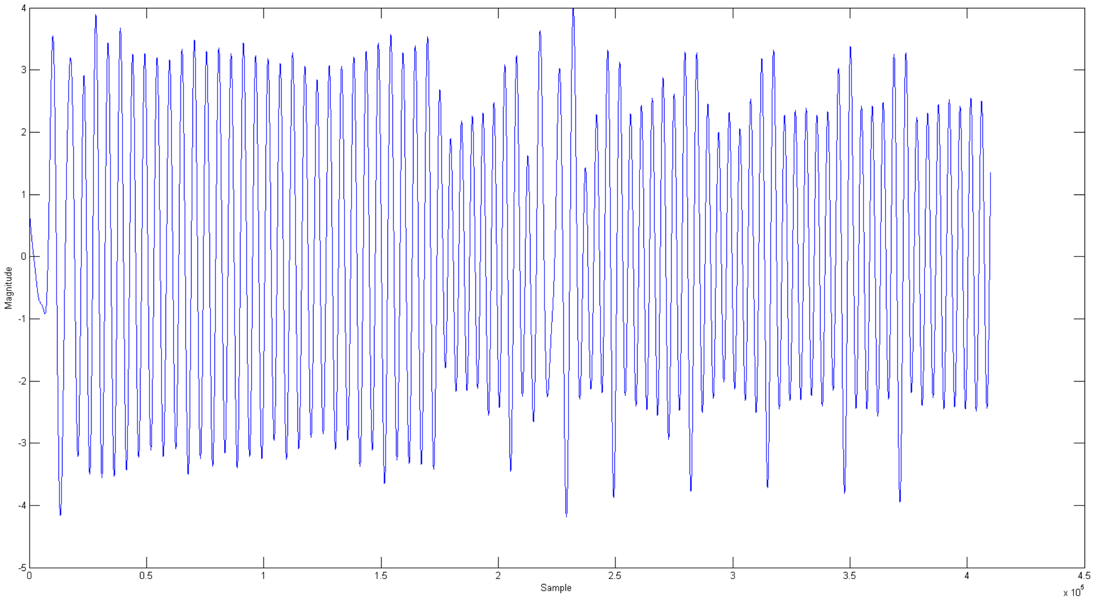


Figure 3.13: 1024-bit key RSA filtered trace - GMP library

- Trace name: gmp1024_250M_1.trc
- RSA Key: 1024 bits $\{FFFFF00000FFFFF00000\ldots\}$
- RSA algorithm: Sliding window, GMP library
- Sampling frequency: 250 MS/sec

Based on the trace structure, we know that the lower peaks are still the distinguisher between different operations. Similar to the case of 400-bit key, a deeper spike represents a multiply operation while a shorter one represents a square operation, and the differences are very clear. During the sequence of 1's, we can see six square operations followed by a multiply operation; thus, the window size is 6. Moreover, 6 is also the maximum window size of the modular exponentiation in the GMP library. In the beginning of the traces, there are 31 pre-computations, calculating x^3, x^5, up until x^{63}.

3.3.2 Summary

In conclusion, the window size and the number of pre-calculations vary as a function of the key length. The longer the key, the larger the window size in order to compute the exponentiation efficiently. The library fixes the maximum window size of 6; thus, for standard RSA keys, either 1024 bits or 2048 bits, the window size would always be 6. Moreover, for a window size of n, the number of pre-computations required is $2^{n-1} - 1$. Table 3.1 shows the measurements we made to verify the window size of the sliding window method.

Table 3.1: Window size and Pre-computations - GMP Library

Key length	Window size	# Pre-computations
80	3 (verified)	3
200	4 (verified)	7
400	5 (verified)	15
1024	6 (verified)	31

Right now, we successfully distinguish the multiply and square operations in the electromagnetic traces. In the cases of 80-bit key and 200-bit, the square operation has higher amplitude than the multiply one, but in the other cases, the multiply operation has higher amplitude than the square one. The difference might be resulted different sampling rate of the oscilloscope. As the key gets longer, the sampling rate is reduced appropriately to fit the whole RSA key on the oscilloscope screen, which results to the change in the parameter of the two filters. These modifications affect the amplitude of the multiply and square operations.

However, the sliding window method has different types of multiplications, which results in several unknown bits in the middle of the window. As the key gets longer, the number of different types of multiplications, which equals the number of pre-calculations, increases. It means that recovering the full key is even more challenging and time-consuming. We are still in the process of finding a method that can distinguish the different multiplications in order to uncover all the unknown bits in the reconstructed key.

3.4 Modular Exponentiation (RSA) – OpenSSL library

In this part, we perform an attack on the modular exponentiation algorithm of the OpenSSL library. Fortunately, the library also employs the sliding window algorithm, similar to the GMP library; thus, we are able to differentiate the multiply and square operations using the same procedure. In addition, the OpenSSL library employs Montgomery reduction to deal with modular arithmetic, which can enhance the execution time significantly. Therefore, we expect to observe the filtered traces with the same strutter as in the GMP library but with shorter execution time due to the additional improved algorithm.

3.4.1 Different Key Sizes

Similar to the GMP library, the OpenSSL library employs different window sizes for different RSA key lengths. In this section, we explore the window size as well as the number of pre-computations as a function of the key length. For the convenience, the key is chosen to be consisting of just *FFFFF* and *00000* in order to quickly identify the window size on the filtered traces. A sequence of consecutive 1's makes sure the system to always use the maximum of its window size; and a sequence of consecutive 0's helps to distinguish the multiply and square operations. The same two-stage filtering process is applied for all the collected traces in order to remove the unwanted factors such as noise, other operations, and carrier frequency.

3.4.1.1 80-bit key

We start with a 80-bit key RSA, and the general information of the trace is shown below. Similarly, the two-stage filter is applied and the result trace is shown in Figure 3.14. Since this is a short key, a small window size and a few number of pre-computations are expected.

- Trace name: open80_2.5G.trc
- RSA Key: 80 bits {*FFFF0000FFFF0000FFFF*}
- RSA algorithm: Sliding window, OpenSSL library
- Sampling frequency: 2.5 GS/sec

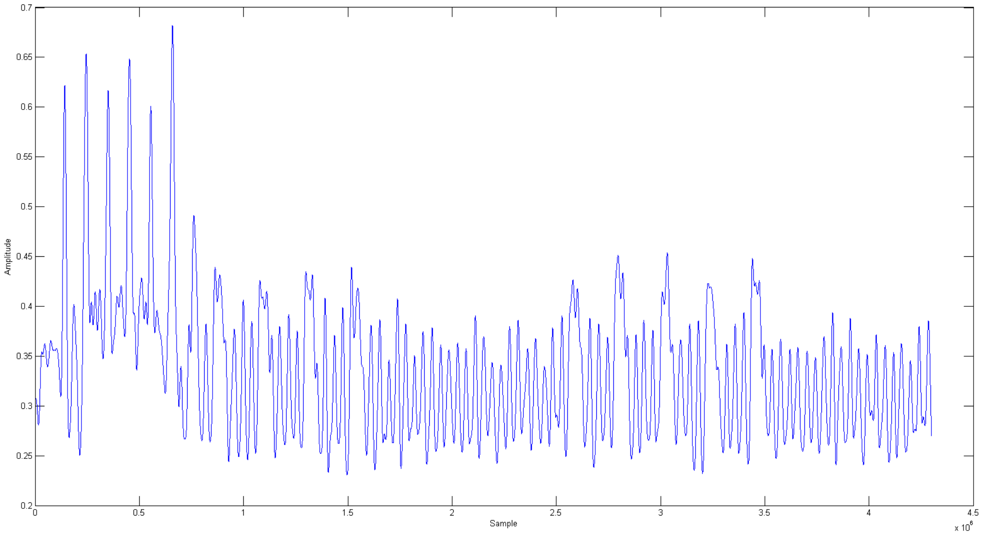


Figure 3.14: 80-bit key RSA filtered trace - OpenSSL library

Based on the sequence of 0's, we conclude that the upper peaks are the distinguisher between different operations. A higher and wider spike represents a multiply operation while a shorter one represents a square operation. However, the filtered trace is not as uniform as the trace of GMP library; thus, it is more difficult to identify each operation. During the sequence of 1's, we can see four square operations followed by a multiply operation; thus, the window size is 4. In the beginning of the traces, there are 7 high spikes, which are the 7 pre-computations when calculating x^3, x^5, up until x^{15}.

3.4.1.2 320-bit key

This part deals with a 320-bit key RSA, with the general trace information shown below. Similarly, the two-stage filter is employed, and the result trace is shown in Figure 3.15. Since this is a longer key than before, we expect a larger window size and a more number of pre-computations at the beginning.

- Trace name: open320_2.5G_1.trc
- RSA Key: 320 bits $\{FFFFF00000FFFFF00000\ldots\}$
- RSA algorithm: Sliding window, OpenSSL library
- Sampling frequency: 1 GS/sec

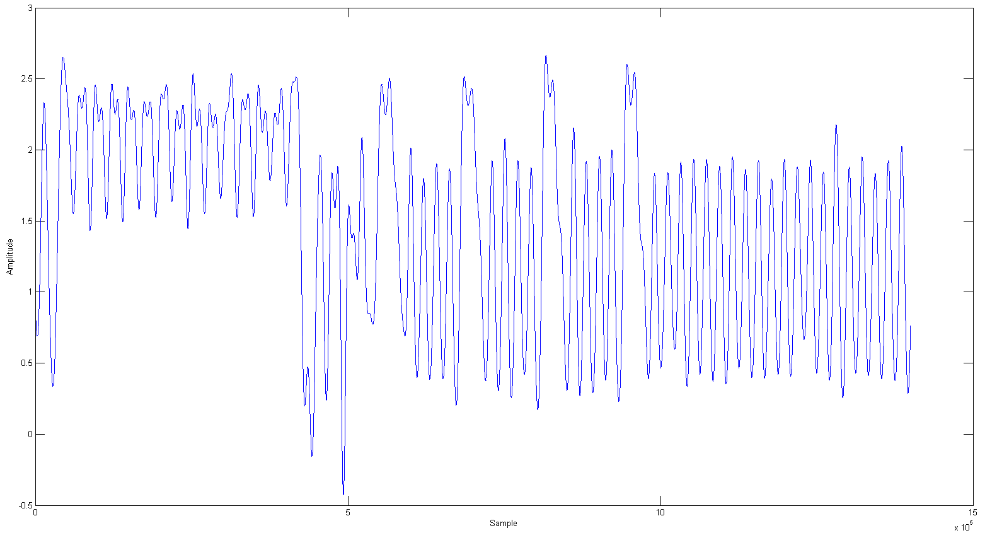


Figure 3.15: 320-bit key RSA filtered trace - OpenSSL library

Based on the trace structure, we know that the upper peaks are still the distinguisher between different operations. A shorter spike represents a square operation while a higher and wider one represents a multiply operation. However, the differences are clearer than in the case of 80-bit key. During the sequence of 1's, we can see five square operations followed by a multiply operation; thus, the window size is 5. In the beginning of the traces, there are 15 multiply operations, and these are the 15 pre-computations, calculating x^3, x^5, up until x^{31}.

3.4.1.3 1024-bit key

Finally, we look at a 1024-bit key RSA, which is a very common key length, and the general trace information is shown below. Again, the two-stage filter is used to remove unwanted factors, and the result trace is shown in Figure 3.16. Since this is a really long key, a large window size and a big number of pre-computations at the beginning will be our assumption.

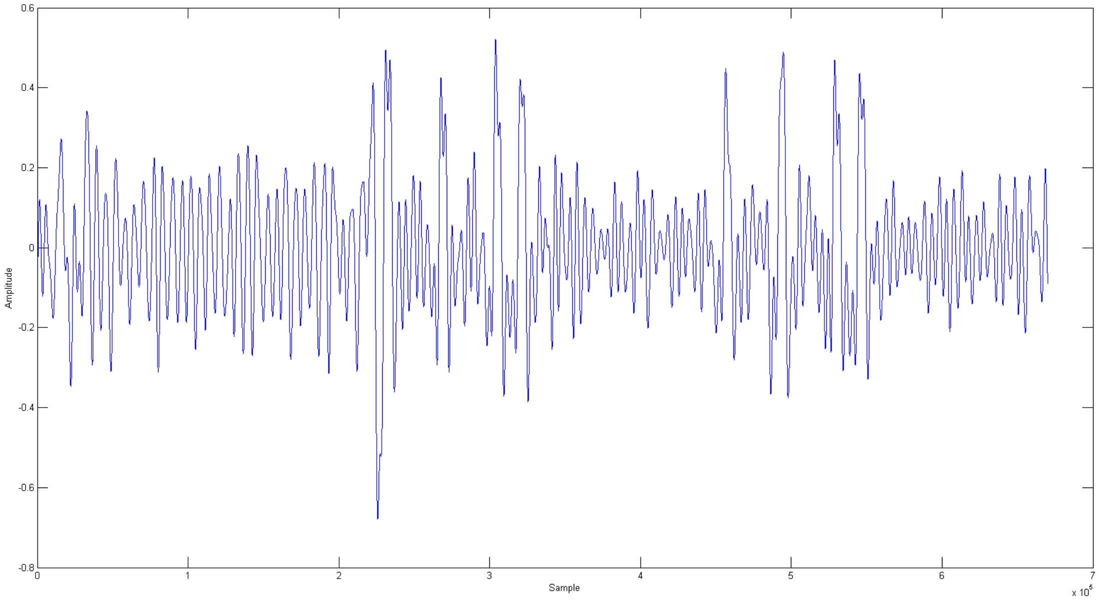


Figure 3.16: 1024-bit key RSA filtered trace - OpenSSL library

- Trace name: open1024_250M.trc
- RSA Key: 1024 bits $\{FFFFF00000FFFFF00000\ldots\}$
- RSA algorithm: Sliding window, OpenSSL library
- Sampling frequency: 250 MS/sec

Based on the trace structure, we know that the upper peaks are still the distinguisher between different operations. Similar to the case of 320-bit key, a shorter spike represents a square operation while a higher and wider one represents a multiply operation, and the differences are very clear. During the sequence of 1's, we can see six square operations followed by a multiply operation; thus, the window size is 6. Moreover, 6 is also the maximum window size of the modular exponentiation in the OpenSSL library. In the beginning of the traces, there are 31 pre-computations, calculating x^3, x^5, up until x^{63}.

3.4.2 Summary

In conclusion, the window size and the number of pre-calculations vary as a function of the key length. The longer the key, the larger the window size in order to compute the exponentiation efficiently. The library fixes the maximum window size of 6; thus, for standard RSA keys, either 1024 bits or 2048 bits, the window size would always be 6. Moreover, for a window size of n, the number of pre-computations required is $2^{n-1} - 1$. Table 3.2 shows the measurements we made to verify the window size.

Table 3.2: Window size and Pre-computations - OpenSSL Library

Key Length	Window size	# Pre-computations
80	4 (verified)	7
320	5 (verified)	15
1024	6 (verified)	31

Right now, we successfully distinguish the multiply and square operations in the electromagnetic traces. However, we are facing the same problem as in using GMP library. Different types of multiplications result in several unknown bits in the middle of the window, which makes the process of recovering the full key to be difficult and time-consuming. Our goal is to find a method that can distinguish the different multiplications in order to uncover all the unknown bits in the reconstructed key.

3.5 Comparing RSA Methods

We have introduced and performed attacks on three different implementations of RSA algorithm, from really simple to more complex. First method is the simple multiply-and-square algorithm using the GMP library. It scans the exponent bit from left to right and carries out the multiply and square operations accordingly. This method is faster than the brute-force exponentiation but still really slow. The second method, sliding window algorithm of the GMP library, is more efficient. It still scans the exponent bit from left to right, and squares the value if the current bit is 0. However, when the current bit is 1, the system multiplies the value with a pre-computed values. The second method is quite an improvement, about one and a half time faster than the first method. The OpenSSL library also employs the sliding window algorithm but with an additional Montgomery reduction to handle modular arithmetic. Thus, the third method is much more efficient than the other two, approximately doubling the speed of the second method. Table 3.3 summarizes the average execution time of three RSA methods performing on the same 1024-bit RSA key.

Table 3.3: Comparing RSA execution time for 1024-bit key

RSA method	Average execution time
Simple multiply-and-square	32 ms
Modular Exponentiation (GMP)	22 ms
Modular Exponentiation(OpenSSL)	12 ms

Chapter 4

Attacking AES

4.1 Collecting the AES traces

The complete C code to generate AES operations can be found in the Appendix. As stated in 2.3.2, we have to use LPT port as a trigger source in collecting AES electromagnetic traces due to the fast operation of AES encryption. We first set the LPTs port to high, which indicates the start of the trigger.

```
outb(255, BASEPORT);
```

Then, two AES encryptions are performed with the same plaintext. With this method, it would be easier to recognize the AES structure inside the collected traces.

```
AES_encrypt(out, temp, &ke);
AES_encrypt(out, out, &ke);
```

After these encryptions, we signal the end of the trigger by setting LPT port to low, ready for the next trigger.

```
outb(0, BASEPORT);
```

The system is then start a counter, which is necessary to allow enough delay for the oscilloscope between two successive triggers.

```
for (j=0; j<1048576; j++);
```

Next, we employ a while loop to automatically generate a thousand AES traces on the scope. The previous ciphertext will become plaintext for the next AES operation. Besides, we incorporate print functions to record the plaintext and ciphertext pair of each trace that we collect in a separate text file, which will be useful for the Differential Electromagnetic Analysis later. The electromagnetic traces are saved in the binary file format, with the general information recorded. Figure 4.1 shows a sample AES trace file captured.

- Trace names: from aes_000.trc to aes_999.trc
- AES Key: 128 bits (*010000...000001*)
- AES algorithm: OpenSSL library
- Sampling frequency: 20 GS/sec

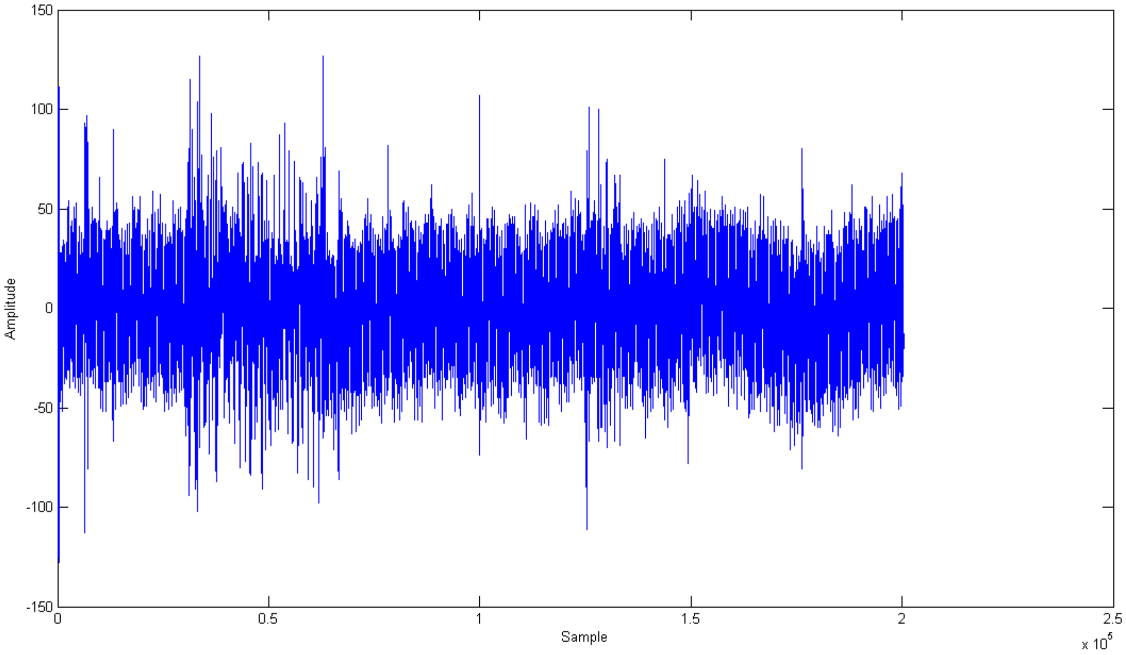


Figure 4.1: A sample AES trace

4.2 Filtering the AES traces

Before performing the DEMA attack, we have to filter them in order to clear out unwanted noises and other operations of the system. The complete MATLAB script for the filtering process can be found in the Appendix. In the raw traces, we can observe significant noise spikes. However, we can see a distinguishable pattern between samples of around 2E4 and 8E4, which we assume to be the region of two AES encryptions. Figure 4.2 shows the processed raw trace after the extraction of the estimated AES region.

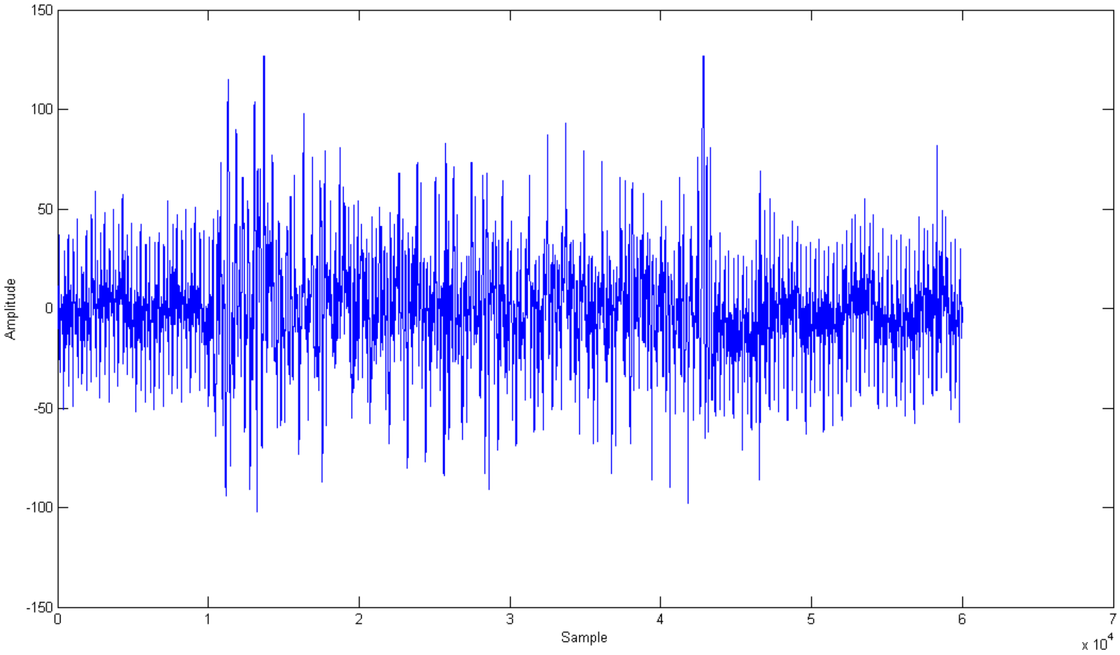


Figure 4.2: Extracted AES region from the raw trace

The trace is now much clearer with the AES operation standing out in the middle. However, there are still a lot of noise spikes in all over the trace. The next step is to plot the spectrum of the electromagnetic trace, as shown in Figure 4.3, in order to find which frequency range the AES operations execute at. Then a bandpass filter can be applied to minimize the noise component and other operations in the trace.

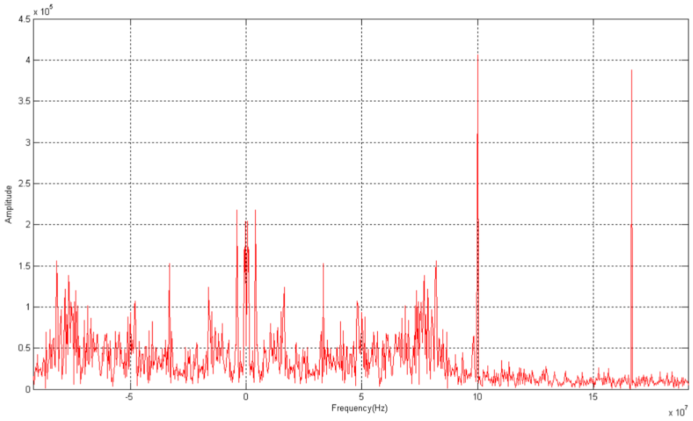


Figure 4.3: Frequency Spectrum Plot of Raw AES

As seen, there are two very high and narrow spikes in 100 MHz and 170 MHz, which represent the other system operations. There are also many frequency spikes resulting from noises. The AES spectrum is estimated to be between 50MHz and 85MHz, represented by a medium clustering spike. These parameters then become the frequency range for our bandpass filter. Figure 4.4 shows the AES trace after applying this filter.

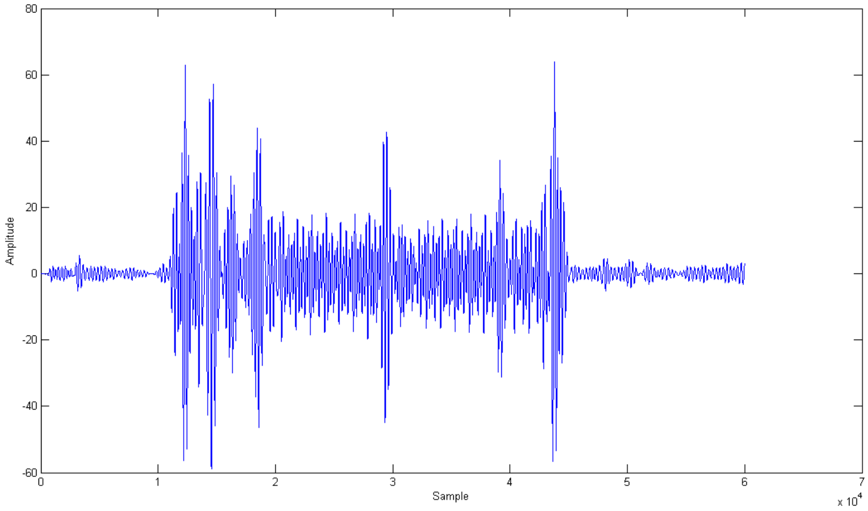


Figure 4.4: Filtered AES

As expected, the filter successfully reduces the noise and other operations of the system. The trace amplitude during RSA operation is much higher than the amplitude during counter time. Since we run two AES encryptions with each trigger, we expect the two similar structures separated by a significant spike between the two AES operations. Thus, the second AES operation is extracted in order to perform the DEMA attack. The process is accomplished using MATLAB peak detection function. Since the AES operations is started and ended with very high peaks, the index of the last peak will be used as a termination point to cut the second AES. For uniform purpose between the AES traces, the 15,000 samples before the last peak are extracted to represent the second AES operation, and the new extracted traces are downsampled by a factor of 16. Figure 4.5 shows the final traces, which has only around 1000 samples.

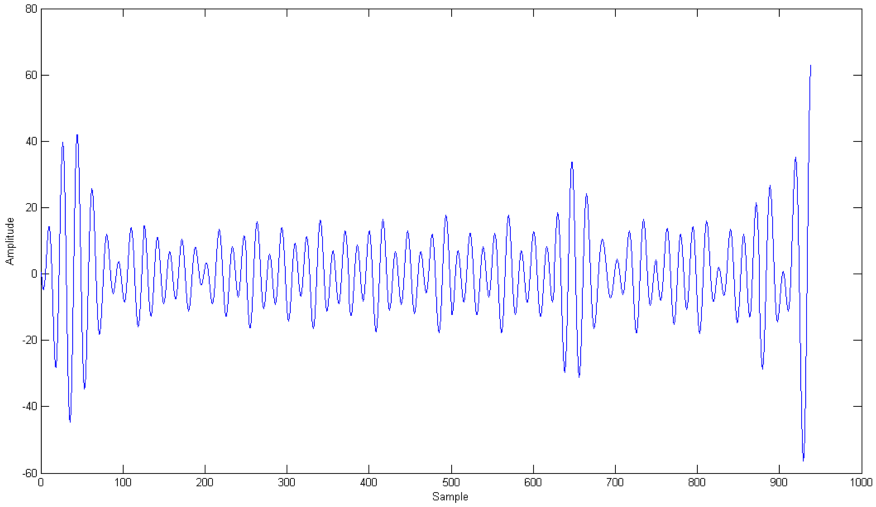


Figure 4.5: Extracted AES Operation

The same process is repeated for all 1000 traces in MATLAB, and nine different AES operations are chosen to show in Figure 4.6. As seen, the same structure can be seen in any AES operation, though the peak locations and heights vary from trace to trace. We managed to make a decent alignment for all collected traces, and hope to be able to perform a DEMA attack on these traces. Finally, the trace data is saved as text files, as shown in Figure 4.7. Each text file contains two columns. The first column is the index, and the second column is the corresponding amplitude of the trace.

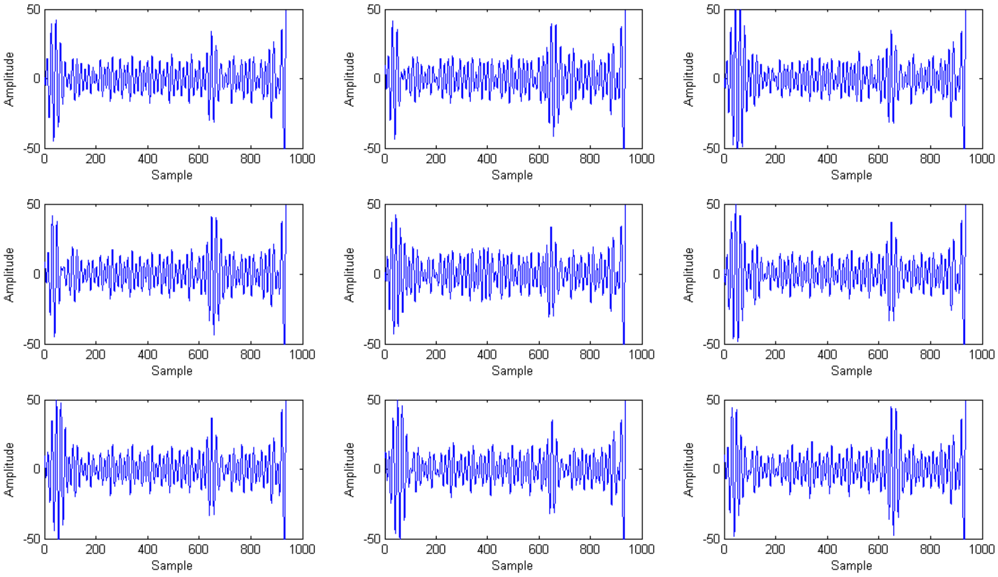
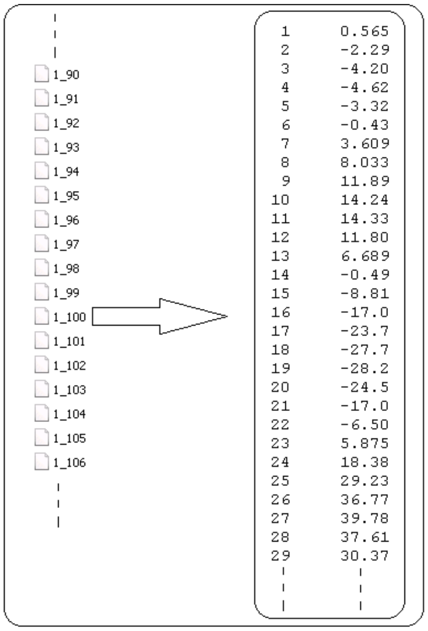


Figure 4.6: Nine different AES operations



```
 1    0.565
 2    -2.29
 3    -4.20
 4    -4.62
 5    -3.32
 6    -0.43
 7    3.609
 8    8.033
 9    11.89
10    14.24
11    14.33
12    11.80
13    6.689
14    -0.49
15    -8.81
16    -17.0
17    -23.7
18    -27.7
19    -28.2
20    -24.5
21    -17.0
22    -6.50
23    5.875
24    18.38
25    29.23
26    36.77
27    39.78
28    37.61
29    30.37
```

Figure 4.7: Downsampled AES written into Text Files

4.3 Differential Electromagnetic Analysis on the AES Traces

The complete MATLAB script for the DEMA attack can be found in the Appendix. First, an intermediate result of the executed algorithm is chosen for the attack. It has to be a function of d (plaintext or ciphertext) and k (a small part of the key). In this case, we chose the intermediate value y after the first S-box: $y = f(x, k) = \text{Sbox}(x \oplus k)$.

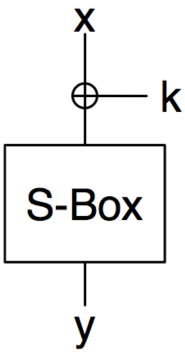


Figure 4.8: Intermediate value y after the first S-box

Here we have D = 20,000 different data blocks are fed to the device for encryption, and the corresponding electromagnetic traces are recorded. The input data blocks are denoted as a vector $d = \{d_1, d_2 \ldots d_D\}$. The electromagnetic trace for each data block is represented as a vector $t_i = \{t_{i1}, t_{i2} \ldots t_{iT}\}$, which is stored in each binary trace file. Then, a matrix T of dimension D by T is constructed by stacking the recorded AES traces.

$$T = \begin{bmatrix} t_1 \\ \vdots \\ t_D \end{bmatrix} = \begin{bmatrix} t_{11} & \cdots & t_{1T} \\ \vdots & \ddots & \vdots \\ t_{D1} & \cdots & t_{DT} \end{bmatrix}$$

All AES traces are plotted together, as shown in Figure 4.9.

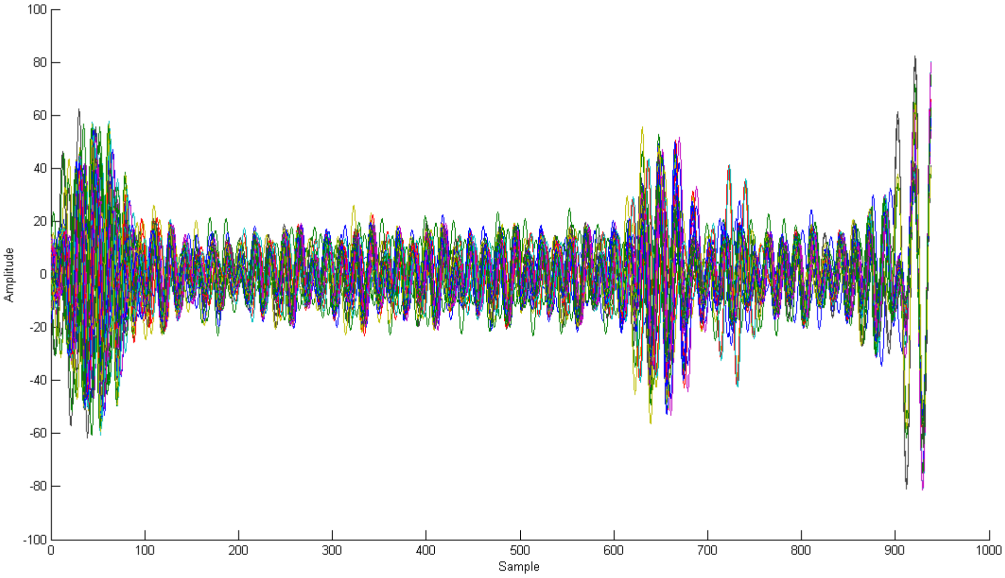


Figure 4.9: All electromagnetic traces

Next, hypothetical intermediate values are calculated for every possible key. The key hypotheses are represented as a vector $k = \{k_1, k_2 \dots k_K\}$. Then, the data vector d and key vector k are used to compute intermediate values to get the matrix $V = [v_{ij}]$, where $v_{ij} = f(d_i, k_j) = \text{Sbox}(d_i \oplus k_j)$.

$$V = \begin{bmatrix} v_{11} & \cdots & v_{1T} \\ \vdots & \ddots & \vdots \\ v_{D1} & \cdots & v_{DT} \end{bmatrix}$$

After that, the intermediate values in matrix V are mapped to the electromagnetic radiation values, resulting in a new matrix H of hypothetical electromagnetic radiation values. Most common methods for doing so are Hamming Weight model, Hamming Distance model and Zero Value model. In this case, we first apply the Hamming Weight model leading to $H = \text{HW}(V)$. Finally, hypothetical power consumption values in matrix H are compared with the actual power traces. The correlation coefficient between each column $\mathrm{h_i}$ of matrix H and each column $\mathrm{t_j}$ of matrix T are calculated and stored in a new K×T correlation coefficient matrix R, with $r_{ij} = \mathrm{corr}(h_i, t_j)$.

$$R = \begin{bmatrix} r_{11} & \cdots & r_{1T} \\ \vdots & \ddots & \vdots \\ r_{D1} & \cdots & r_{DT} \end{bmatrix}$$

Correlation coefficient values for all 256 possible keys are plotted as shown in Figure 4.10.

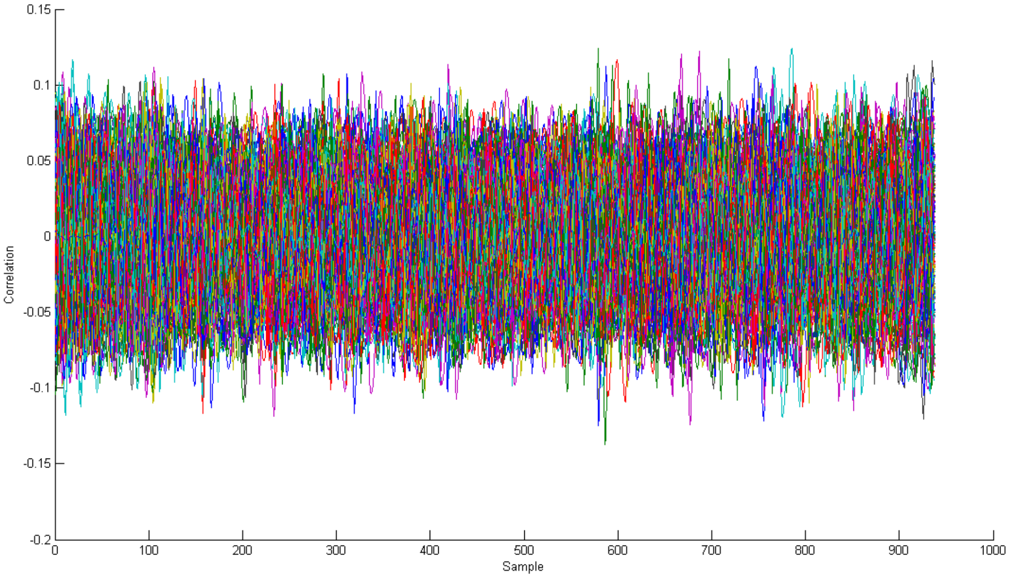


Figure 4.10: Correlation values for 256 possible keys

Unfortunately, we cannot find any leakage from the collected traces; thus, the secret key cannot be identified. After that, we have tried several approaches to improve our differential attack such as increasing the number of traces to 50,000, or cutting out the first AES operation instead of the second one. Different models like Hamming distance, or a single bit are employed but with no success.

4.4 Summary

We believe that the issues regarding AES analysis are alignment of the AES encryptions in different traces, and the inaccuracy of the trigger from the Atom Board. To successfully retrieve the correct AES key, the AES encryptions in electromagnetic traces need to be perfectly aligned. Currently, we use peak detection to align the AES encryptions. Since the peaks themselves are precarious, we might need to find a more reliable way to correctly align the encryptions. Another problem is the inaccuracy of the trigger. In this project, we collected thousands of AES traces automatically using the trigger. In C code implementation of the AES encryptions, a trigger is produced for each encryption. The oscilloscope is set to collect an electromagnetic trace at each trigger. We believe that wrong trigger events are caused by noise. These additional triggers cause a mismatch between the trace and the coresponding input (plaintext). Such a mismatch can render all subsequent traces useless and thereby hinder a successfull attack. An improved measurement setup could solve this problem. Nevertheless, we have successfully identified a way to uncover the AES leakage in the EM trace. Follow-up work should be able to exploit this leakage.

Chapter 5

Conclusion

5.1 Summary

In this project, we performed side-channel attacks on an Intel Atom Processor for RSA crypto scheme by exploiting electromagnetic radiation leakage. The attacks were performed under three implementations: simple multiply-and-square algorithm, exponentiation function in GMP library, and exponentiation function in OpenSSL library.

For simple multiply-and-square algorithm, we use a 1024-bit key, which is a standard RSA key length. The key for simple multiply-and-square algorithm was recovered successfully. However, there is a small error percentage of about 4% due to signal interferences, and we expect to diminish or even eliminate this error by performing the same attack several times.

In addition to simple multiply-and-square algorithm, we performed attack on RSA using modular exponentiation function in GMP library. The modular exponentiation function deploys sliding window algorithm, which performs pre-computations in accordance to the window size, or the number of exponent bits. We discovered that the number of pre-computations varies with key length. We have tabulated the number of pre-computations performed for different key lengths. Although multiply and square operations can be distinguished easily, different number of pre-computations for different key length suggests that recovering the full key is challenging and requires an additional key reconstruction method that is outside of the scope of this project.

Like GMP library, OpenSSL modular exponentiation function makes use of the sliding window algorithm. In addition to the sliding window algorithm, OpenSSL library uses Montgomery Reduction algorithm, which makes it more efficient than GMP library. As in our attack on GMP library, we can recover the squares and multiplies which can be used for a key recovery attack. However, some entropy of the key remains, since our attack can not distinguish precomputed values during the multiplications.

Side-channel analysis on AES algorithm proves to be much more complex, and requires more commitment than the analysis on RSA. Unlike RSA that requires a few electromagnetic traces to perform attack, AES algorithm demands consideration of thousands of traces. Although the chance of recovering the key rises with larger the number of traces considered, due to limited time and resources, we only managed to perform an attack on 20,000 traces. In addition to the large number of traces required, successful AES attack needs a proper alignment of AES operations among traces. We implemented alignment by the same filter technique used in RSA attack and peak detection. However, we have not yet achieved a successful side-channel attack on AES.

Overall, our project is a success as we can recover a standard 1024-bit RSA key for simple multiply and square algorithm with very little error percentage. Besides, we have promising results with modular exponentiation functions of GMP and OpenSSL cryptographic libraries. Although a full key recovery is not possible so far for these implementations, we can recover parts of the key and analyze pre-computations. For the AES implementation under OpenSSL library, a full key recovery is not possible at the current stage due to the limited time frame available. However, we can successfully capture AES traces on the oscilloscope and correctly identity the regions of AES encryptions. Therefore, even high-end embedded platforms are vulnerable to side-channel attacks like EM attacks, and open source implementation are not protected.

5.2 Future Work

The next steps of this project would be to distinguish the different pre-computations for RSA analysis, and to resolve the alignment issue in AES analysis. Currently, only parts of the key can be recovered for modular exponentiation of GMP and OpenSSL libraries. To distinguish among the different pre-computations in electromagnetic traces will require more in-depth research and commitment. The next step regarding side-channel analysis on AES would be to connect the oscilloscope with the network, and set up the oscilloscope so that it properly captures each AES encryption trace while recording the corresponding plaintext and ciphertext pair. This would require proficient manipulation of the oscilloscope using low-level scripts.

5.3 Contribution

Since we could successfully recover a standard 1024-bit RSA key for simple multiply and square algorithm, the major conclusion from our project is that even high-end embedded systems like Intel Atom Processor are vulnerable to side channel attacks. This suggests that there is potential threat in the security of current embedded systems. The results from our project suggest that more extensive improvements are in need for securing future embedded devices against side-channel attacks. Possible areas of research to eliminate electromagnetic radiation leakage might include interfering the leakage signals.

Bibliography

[1] Amoth, Dough. “Intel details next-generation ’Pine Trail’ Atom platform, intros updated ’Moblin’ UI | TechCrunch.” TechCrunch. N.p., n.d. Web. 31 Mar. 2013.
http://techcrunch.com/2009/05/19/intel-details-next-generation-pine-trail-atom-platform-intros-updated-moblin-ui/

[2] Cangelsolo, San. “Intel Silverthorne, Diamondville to hit the streets as Atom | Chips | Geek.com.” Geek.com. N.p., n.d. Web. 31 Mar. 2013.
http://www.geek.com/articles/chips/intel-silverthorne-diamondville-to-hit-the-streets-as-atom-2008033/

[3] Intel® Atom™ Processor N400 and N500 Series: Datasheet.
http://www.intel.com/content/www/us/en/processors/atom/atom-n400-n500-vol-1-datasheet.html

[4] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. *Side-channel vulnerability factor: a metric for measuring information leakage*. In Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA ’12). IEEE Computer Society, Washington, DC, USA, 106-117.

[5] P. Kocher, J. Jaffe, B. Jun, P. Rohatgi: Introduction to differential power analysis. Journal of Cryptographic Engineering (JCEN), Vol. 1, No. 1, Springer, 2011.

[6] LeCroy WavePro 725Zi Oscilloscope
http://teledynelecroy.com/oscilloscope/oscilloscopemodel.aspx?modelid=4717

[7] Mangard, Stefan, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks revealing the secrets of smart cards*. New York, N.Y.: Springer, 2007. Print.

[8] Mini-Circuits ZFL-1000LN+ Low Noise Amplifier Datasheet
http://www.minicircuits.com/pdfs/ZFL-1000LN+.pdf

[9] OpenSSL library.
http://www.OpenSSL.org

[10] Paar, Christof, and Jan Pelzl. *Understanding cryptography a textbook for students and practitioners*. Berlin: Springer, 2009. Print.

[11] Oswald, David, and Christof Paar. "Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World." *Cryptographic hardware and embedded systems–CHES 2011* 13th international workshop, Nara, Japan, September 28-October 1, 2011 : proceedings. Heidelberg: Springer, 2011.

[12] Public Key Cryptography Standard #1
http://www.rsa.com/rsalabs/pkcs/files/h11300-wp-pkcs-1v2-2-rsa-cryptography-standard.pdf

[13] The GNU Multiple Precision Arithmetic Library.
http://gmplib.org

Appendix A

C Code and MATLAB Code

A.1 C Code

A.1.1 RSA simple multiply-and-square

```
#include "stdio.h"
#include "gmp.h"
#include <sys/timeb.h>
#include <sys/io.h>
#include <unistd.h>
#include <termios.h>
#include <fcntl.h>
#include <time.h>
int main( )
{
        /* IO test routine */
        printf("Test IO\n");
        int fd;
        //get permission to access COM1 port
        fd = open("/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
        if (fd == -1) {
                printf("io error\n");
        }
        else {
                fcntl(fd,F_SETFL,0);
        }
        int r;
        char tpstr[10];
        tpstr[0]=0x00;
        //test write to COM1 port
        r = write(fd,tpstr,1);
        if (r < 0) printf("fail␣to␣write\n");
        /* End of IO test routine */
        /* prepare sleep time */
        timespec sleeptime;
        sleeptime.tv_sec = (time_t)0;
        sleeptime.tv_nsec = 1000000;
        /* declare and initialize variables required for RSA */
        mpz_t a,b,c,d;
        mpz_init(a);
        mpz_init(b);
        mpz_init(c);
        mpz_init(d);
        /* Prepare Pseudo-Random Number Generator */
        gmp_randstate_t rnd;
        gmp_randinit_default (rnd);
        unsigned int seed;
        seed = 10000;
        gmp_randseed_ui(rnd, seed);
        /* set a, b and c */
        mpz_urandomb(a, rnd, 2048);
        mpz_urandomb(c, rnd, 2048);
        mpz_nextprime(c, c);
        mpz_set_str(b, "ffffffffffaaaaaaaaaa0000000000bbbbbbbbbbffffffffffcccccccccc
␣␣␣␣␣␣␣␣0000000000ddddddddddffffffffffeeeeeeeeee00000000009999999999ffffffffff
␣␣␣␣␣␣␣␣888888888800000000007777777777ffffffffff666666666600000000005555555555
␣␣␣␣␣␣␣␣44444444443333333333222222222211111111110000000000ffffff", 16);
        size_t bits = mpz_sizeinbase(b, 2);
        /* RSA exponentiation and modular reduction */
        //d = a^b mod c
        while(1)
        {
                int i = 0;
                mpz_set_ui(d,1);
                write(fd,tpstr,1); //send trigger to COM1
                nanosleep(&sleeptime,NULL); //delay
                mpz_mul(d,d,a);
                mpz_mod(d,d,c);
                for(i = bits-1; i >= 0; i--) {
                        mpz_mul(d, d, d);
                        mpz_mod(d, d, c);
                        if(mpz_tstbit(b, i) == 1) {
                                mpz_mul(d, d, a);
                                mpz_mod(d, d, c);
                        }
                }
                nanosleep(&sleeptime,NULL); //delay
        }
        return 0;
}
```

A.1.2 RSA modular exponentiation (GMP library)

```
#include <stdio.h>
#include "gmp.h"
#include <assert.h>
#include <sys/io.h>
#include <unistd.h>
#include <termios.h>
#include <fcntl.h>
#include "stdlib.h"
#include <sys/timeb.h>
#include <time.h>
int main( )
{
        /* IO test routine */
        printf ("Test IO\n");
        int fd;
        //get permission to access COM1 port
        fd = open("/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
        if (fd == -1)
                printf("IO error\n");
        else
                fcntl(fd,F_SETFL,0);
        int r;
        char tpstr[10];
        tpstr[0] = 0x00;
        //test write to COM1 port
        r = write (fd,tpstr,1);
        if (r<0) printf("fail to write\n");
        /* End of IO test routine */
        /* prepare sleep time */
        timespec sleeptime;
        sleeptime.tv_sec = (time_t)0;
        sleeptime.tv_nsec = 500000;
        /* declare and initialize variables required for RSA */
        mpz_t result, b, e, m;
        mpz_init(result);
        mpz_init(b);
        mpz_init(e);
        mpz_init(m);
        /* Prepare Pseudo-Random Number Generator */
        gmp_randstate_t rnd;
        gmp_randinit_default (rnd);
        unsigned int seed;
        seed = 10000;
        gmp_randseed_ui(rnd, seed);
        /* set modulus, base and exponent */
        mpz_urandomb(m, rnd, 2048);
        mpz_nextprime(m, m);
        mpz_urandomb(b, rnd, 2048);
        mpz_set_str(e, "ffffffffff0000000000ffffffffff0000000000ffffffffff0000000000
        ffffffffff0000000000ffffffffff0000000000ffffffffff0000000000ffffffffff
        0000000000ffffffffff0000000000ffffffffff0000000000ffffffffff0000000000
␣␣␣␣␣␣␣␣ffffffffff0000000000ffffffffff0000000000ffffffffff000000", 16);
        while (1){
                write(fd, tpstr, 1); //send trigger to COM1
                nanosleep(&sleeptime,NULL); //delay
                /* RSA modular exponentiation */
                /*result = b^e mod m
                mpz_powm (result, b, e, m);
                nanosleep(&sleeptime,NULL); //delay
        }
        return 0;
}
```

A.1.3 RSA modular exponentiation (GMP library)

```
#include <stdio.h>
#include <openssl/bn.h>
#include <assert.h>
#include <sys/io.h>
#include <unistd.h>
#include <termios.h>
#include <fcntl.h>
#include "stdlib.h"
#include <sys/timeb.h>
#include <time.h>
int main( ) {
        /* IO Test */
        printf ("Test␣IO\n");
        int fd;
        //get permission to access COM1 port
        fd = open("/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
        if (fd == -1)
                printf("IO␣error\n");
        else
                fcntl(fd,F_SETFL,0);
        int r;
        char tpstr[10];
        tpstr[0] = 0x00;
        //test write to COM1 port
        r = write (fd,tpstr,1);
        if (r<0) printf("fail␣to␣write\n");
        /* End of IO Test */
        /* prepare sleep time */
        timespec sleeptime;
        sleeptime.tv_sec = (time_t)0;
        sleeptime.tv_nsec = 500000;
        /* RSA part starts here */
        const char base_hex[] = "FFFF1111222233334444555566667777888899990000AAAABBBB
␣␣␣␣␣␣␣␣CCCCDDDDEEEE";
        /*
        const char exp_hex[] = "FFFFF00000FFFFF00000";
        */
        /*
        const char exp_hex[] = "FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000
        FFFFF00000FFFFF00000FFFFF00000";
        */
        const char exp_hex[] = "FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000
˽˽˽˽˽˽˽˽FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF
˽˽˽˽˽˽˽˽00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000
˽˽˽˽˽˽˽˽FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF00000FFFFF0";
        const char mod_hex[] = "FFFF00005555AAAAFFFF00005555AAAAFFFF00005555AAAAFFFF
˽˽˽˽˽˽˽˽00005555AAAAFFFF00005555AAAAFFFF00005555AAAAFFFF00005555AAAAFFFF00005555AAAA
˽˽˽˽˽˽˽˽FFFF00005555AAAAFFFF00005555AAAAFFFF00005555AAAAFFFF00005555AAAAFFFF00005555
˽˽˽˽˽˽˽˽AAAAFFFF00005555AAAAFFFF00005555AAAAFFFF00005555AAAB";
        BIGNUM *result, *base, *exp, *mod;
        result = BN_new();
        base = BN_new();
        exp = BN_new();
        mod = BN_new();
        BN_hex2bn(&base, base_hex);
        BN_hex2bn(&exp, exp_hex);
        BN_hex2bn(&mod, mod_hex);
        BN_CTX *ctx;
        ctx = BN_CTX_new();
        printf( "Running˽modular˽exponentiation\n");
        while (1){
                write(fd, tpstr, 1); //send trigger to COM1
                nanosleep(&sleeptime,NULL); //delay
                //RSA modular exponentiation
                BN_mod_exp(result, base, exp, mod, ctx);
                nanosleep(&sleeptime,NULL); //delay
        }
        return 0;
}
```

A.1.4 AES 1-in-1

```
#include <openssl/aes.h>
#include <sys/timeb.h>
#include <termios.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/io.h>
#include <sys/types.h>
#include <fcntl.h>
#define BASEPORT 0x378 /* LPT1 */
void printdata (unsigned char* ptr, int number);
//global variables
FILE *file;
int main()
{
        int i, tem;
        unsigned int j;
        /* set permissions to access LPT1 */
        if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}
        /* prepare a text file to record plaintext & ciphertext pairs */
        file = fopen ("file.txt", "w");
        fprintf(file, "Plaintext & Ciphertext Pairs for AES Encryption\n");
        /* AES part starts here */
        unsigned char key[] = {0x00, 0x00, 0xff, 0xff, 0x55, 0x55, 0xaa, 0xaa, 0x00,
        0x00, 0xff, 0xff, 0x55, 0x55, 0xaa, 0xaa};
        unsigned char in[] = {0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01};
        unsigned char out[20];
        unsigned char temp[20];
        AES_KEY ke;
        AES_set_encrypt_key(key,128,&ke);
        printf("running AES\n");
        AES_encrypt(in, out, &ke);
        for (i=0; i< 5000; i++) {
                printdata(out, i+1); //print plaintext
                outb(255, BASEPORT); //set LPT1 port to high (start of trigger)
                /* do 2 AES encryptions with the same plaintext */
                AES_encrypt(out, temp, &ke);
                AES_encrypt(out, out, &ke);
                outb(0, BASEPORT); //set LPT1 port to low (end of trigger)
                printdata(out, i+1); //print ciphertext
                for (j=0; j<1048576; j++); //delay
        }
        fclose(file);
        //take away permissions to access LPT1
        if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}
        printf("end\n");
        return 0;
}
void printdata (unsigned char* ptr, int number) {
        int i;
        fprintf(file, "%4d. 0x", number);
        for (i=0; i<16; i++) {
                fprintf(file, "%x", (*ptr)>>4);
                fprintf(file, "%x", (*ptr)&0x0f);
                ptr++;
        }
        fprintf(file, "\n");
}
```

A.1.5 AES 100-in-1

```
#include <openssl/aes.h>
#include <sys/timeb.h>
#include <termios.h>
#include <time.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/io.h>
#include <sys/types.h>
#include <fcntl.h>
#define BASEPORT 0x378 /* LPT1 */
void printdata (unsigned char* ptr, int number);
//global variables
FILE *file;
int main()
{
        int i, k, l, number, temp;
        unsigned int j;
        number = 1;
        /* set permissions to access LPT1 */
        if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}
        /* prepare a text file to record plaintext & ciphertext pairs */
        file = fopen ("file.txt", "w");
        fprintf(file, "Plaintext␣&␣Ciphertext␣Pairs␣for␣AES␣Encryption\n");
        /* AES part starts here */
        unsigned char key[] = {0x00, 0x00, 0xff, 0xff, 0x55, 0x55, 0xaa, 0xaa, 0x00,
         0x00, 0xff, 0xff, 0x55, 0x55, 0xaa, 0xaa};
        unsigned char in[] = {0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01};
        unsigned char out[20];
        unsigned char temp[20];
        AES_KEY ke;
        AES_set_encrypt_key(key,128,&ke);
        printf("running␣AES\n");
        AES_encrypt(in, out, &ke);
        for (i=0; i< 500; i++) {
                for (k=0; k<100; k++) {
                        printdata(out, number); //print plaintext
                        outb(255, BASEPORT); //set LPT1 port to high (start trigger)
                        /* do 2 AES encryptions with the same plaintext */
                        AES_encrypt(out, temp, &ke);
                        AES_encrypt(out, out, &ke);
                        outb(0, BASEPORT); //set LPT1 port to low (end of trigger)
                        printdata(out, number); //print ciphertext
                        number++;
                        for (l=0; l<4000; l++); //delay
                }
                for (j=0; j<1048576; j++); //delay
        }
        fclose(file);
        //take away permissions to access LPT1
        if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}
        printf("end\n");
        return 0;
}
void printdata (unsigned char* ptr, int number) {
        int i;
        fprintf(file, "%5d.␣0x", number);
        for (i=0; i<16; i++) {
                fprintf(file, "%x", (*ptr)>>4);
                fprintf(file, "%x", (*ptr)&0x0f);
                ptr++;
        }
        fprintf(file, "\n");
}
```

A.2 MATLAB Code

A.2.1 RSA 80-bit key

```
%% Raw Data
clear all
Fs = 2.5e9;   % sampling freq
x = read_binary_tracefile('sim80_2.5G_2.trc',8);
plot(x); xlabel('Amplitude'); ylabel('Sample');
%% Spectrogram
%Compute spectrogram
WINDOW = 1500;
NOVERLAP = 0;
NFFT = 2^nextpow2(WINDOW);
[S,F,T,P]=spectrogram(x,WINDOW,NOVERLAP,NFFT,Fs);
%Change the axis and plot
surf(T,F,10*log10(abs(P)),'EdgeColor','none'); grid on;
axis xy; axis tight; colormap(jet); view(0,90);
xlabel('Time␣(sec)'); ylabel('Frequency␣(Hz)');
ylim([0 4e8]);
%% Fisrt Filter (Bandpass)
b1 = fir1(2000,2/Fs*[50e6 80e6]);
x1 = fftfilt(b1,x);
plot(x1); xlabel('Amplitude'); ylabel('Sample');
%% Rectified Spectrum
plotspec(abs(x1),1/Fs);
%% Second Filter (Rectify and Bandpass)
b2 = fir1(50001,2/Fs*[2e4 6e4]);
x2 = fftfilt(b2,abs(x1));
plot(x2); xlabel('Amplitude'); ylabel('Sample');
%% Extract the RSA part
y = x2(1.8e6:5.5e6);
y = downsample(y,1000);
plot(y);
t = 1:length(y);
%% Find Valleys
[VALS,LOCS] = findpeaks(y,'MINPEAKHEIGHT',15);
plot(t,y,LOCS,VALS,'o'); xlabel('Amplitude'); ylabel('Sample');
%% Decide MUL and SQ
[IDX,C] = kmeans(VALS,2);
```

```
for i = 1:length(LOCS)-1
    if IDX(i)==find(C==max(C))
        text(LOCS(i),17,'M','Color','r');
    else
        text(LOCS(i),17,'S','Color','g');
    end
end
```

A.2.2 RSA 1024-bit key

```
%% Raw Data
clear all
Fs = 250e6;   % sampling freq
x = read_binary_tracefile('sim1024_250M_2.trc',8);
plot(x); xlabel('Sample');ylabel('Amplitude');
%% Spectrogram
% Compute spectrogram
WINDOW = 1500;
NOVERLAP = 0;
NFFT = 2^nextpow2(WINDOW);
[S,F,T,P]=spectrogram(x,WINDOW,NOVERLAP,NFFT,Fs);
%Change the axis and plot
surf(T,F,10*log10(abs(P)),'EdgeColor','none'); grid on;
axis xy; axis tight; colormap(jet); view(0,90);
xlabel('Time␣(sec)'); ylabel('Frequency␣(Hz)');
%% Fisrt Filter (Bandpass)
b1 = fir1(2000,2/Fs*[70e6 80e6]);
x1 = fftfilt(b1,x);
plot(x1); xlabel('Sample');ylabel('Amplitude');
%% Second Filter (Rectify and Bandpass)
b2 = fir1(50000,2/Fs*[2e4 6e4]);
x2 = fftfilt(b2,abs(x1));
plot(x2); xlabel('Sample');ylabel('Amplitude');
%% Extract the RSA part
y = x2(.695e6:8.69e6);
plot(y);
t = 1:length(y);
xlabel('Sample');ylabel('Amplitude');
%% Find Valleys
[VALS,LOCS] = findpeaks(-y,'MINPEAKHEIGHT',0.5);
VALS = -VALS;
plot(t,y,LOCS,VALS,'o');
%% Plot Valleys
[IDX,C] = kmeans(VALS,2);
plot(LOCS,VALS,'*'); hold on;
plot(LOCS(IDX==1),VALS(IDX==1),'*r');
xlabel('Sample');ylabel('Amplitude');
ylim([-2.5 -.5]);
%% Decide MUL and SQ
[IDX,C] = kmeans(VALS,2);
% 0 is SQ, 2 is MUL
```

```
ms = zeros(1,length(VALS));
for i = 1:length(VALS)
    if IDX(i)==find(C==max(C))
        ms(i) = 0;
    elseif IDX(i)==find(C==min(C))
        ms(i) = 3;
    end
end
% Remove consecutive MULs
for i = 1:length(ms)-1
    if ms(i)==3 && ms(i+1)==3
        ms(i+1) = 0;
    end
end
% Convert to binary key
ms = ms;
ms(find(ms==3)-1) = 1;
ms(ms==3) = [];
%% Original key (1024 bit)
% f a 0 b f c 0 d f e 0 9 f 8 0 7 f 6 0 5 4 3 2 1 0 (10 times each) f(x6)
b = [0 0 0 0;0 0 0 1;0 0 1 0;0 0 1 1;0 1 0 0;0 1 0 1;0 1 1 0;0 1 1 1;...
     1 0 0 0;1 0 0 1;1 0 1 0;1 0 1 1;1 1 0 0;1 1 0 1;1 1 1 0;1 1 1 1];
c = [15 10 0 11 15 12 0 13 15 14 0 9 15 8 0 7 15 6 0 5 4 3 2 1 0];
% Generate key
k0 = [];
for i=1:length(c)
    k0 = [k0 repmat(b(c(i)+1,:),1,10)];
end
k0 = [k0 repmat(b(16,:),1,6)];
% plot(k0,'*-');
%% Adjustment
key = ms;
key(1:2) = [];
key = [key(1:100) 0 0 key(101:end)];
key = [key(1:270) 0 0 key(271:end)];
key(400) = [];
key(650) = [];
key = [key(1:775) 0 key(776:end)];
key(868) = [];
key = [key(1:945) 0 key(946:end)];
%% Compare key
plot(key,'*r'); hold on;
plot(k0,'-b'); hold off;
ylim([-.1 1.1]); xlim([0 1024]);
xlabel('Index'); ylabel('Bit␣Value');
% Percentage Error
error = sum(xor(key,k0))/length(k0)
```

A.2.3 AES Write File

```
%% Extract AES
clear all
Fs = 20.0e9;
b = 0;
remov = [];
tic
for a = 10:20
    %file names%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    if(a<10)
        filename = ['C1Test1114120000' num2str(a) '.trc'];
    else if(a<100)
        filename = ['C1Test111412000' num2str(a) '.trc'];
    else
        filename = ['C1Test11141200' num2str(a) '.trc'];
        end
    end
    %read file%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    x = read_binary_tracefile(filename,8);
    x = x(2e4:8e4);
    %filter%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    b1 = fir1(2000,2/Fs*[50e6 85e6]);
    x1 = fftfilt(b1,x);
    %find peaks%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    [PKGS, LOCS] = findpeaks(x1,'MINPEAKHEIGHT', 40, 'MINPEAKDISTANCE', .02e4);
    t1 = 1:length(x1);
    figure;
    plot(t1, x1, LOCS, PKGS , 'o');
    ylim ([-60 80]);
    %cut trace using first and last peaks%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    x2 = x1(LOCS(end)-1.5e4:LOCS(end));
%     figure;
%     plot(x2);
    %downsample%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    x2 = downsample(x2,16);
%     figure;
%     plot(x2);
    if(length(LOCS) >= 14)
       x2 = temp;
       remov = [remov, a];
    end
    %write file for top peaks%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    fileID = fopen( [ '1_', num2str(a)], 'w');
    for m = 1:length(x2)
      fprintf(fileID, '%9.5s␣%9.5s\n',num2str(m), num2str(x2(m)));
    end
    fclose(fileID);
    %store previous trace
    temp = x2;
    %loop count
    display(['count␣is␣', num2str(a)]);
%     % peak check
%     b = b + 1;
%     display([num2str(b),' peak count is ', num2str(length(LOCS))]);
```

```
end
toc
%% %write file for removed traces%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    fileID = fopen( '1_removed', 'w' );
    for m = 1:length(remov)
      fprintf(fileID, '%9.5s %9.5s\n',num2str(m), num2str(remov(m)));
    end
    fclose(fileID);
```

A.2.4 AES DEMA

```
%% Hypothesis
% Get 8-bit in each plain text
clear; clc;
num_trace = 1000;          % num of traces used
num_time = 938;          % num of time data points
pt = zeros(1,num_trace);

fid = fopen('file1.txt');
fgets(fid);                            % skip the 1st line
for i=1:num_trace
    temp = fgets(fid);                 % read the plaint text
    pt(i) = hex2dec(temp(16:17));      % get the 8-bit (CHANGEABLE)
    fgets(fid);                        % skip the cipher text
end
fclose(fid);

% Sbox look up table
Scell = {
    '63', '7c', '77', '7b', 'f2', '6b', '6f', 'c5',...
    '30', '01', '67', '2b', 'fe', 'd7', 'ab', '76',...
    'ca', '82', 'c9', '7d', 'fa', '59', '47', 'f0',...
    'ad', 'd4', 'a2', 'af', '9c', 'a4', '72', 'c0',...
    'b7', 'fd', '93', '26', '36', '3f', 'f7', 'cc',...
    '34', 'a5', 'e5', 'f1', '71', 'd8', '31', '15',...
    '04', 'c7', '23', 'c3', '18', '96', '05', '9a',...
    '07', '12', '80', 'e2', 'eb', '27', 'b2', '75',...
    '09', '83', '2c', '1a', '1b', '6e', '5a', 'a0',...
    '52', '3b', 'd6', 'b3', '29', 'e3', '2f', '84',...
    '53', 'd1', '00', 'ed', '20', 'fc', 'b1', '5b',...
    '6a', 'cb', 'be', '39', '4a', '4c', '58', 'cf',...
    'd0', 'ef', 'aa', 'fb', '43', '4d', '33', '85',...
    '45', 'f9', '02', '7f', '50', '3c', '9f', 'a8',...
    '51', 'a3', '40', '8f', '92', '9d', '38', 'f5',...
    'bc', 'b6', 'da', '21', '10', 'ff', 'f3', 'd2',...
    'cd', '0c', '13', 'ec', '5f', '97', '44', '17',...
    'c4', 'a7', '7e', '3d', '64', '5d', '19', '73',...
    '60', '81', '4f', 'dc', '22', '2a', '90', '88',...
    '46', 'ee', 'b8', '14', 'de', '5e', '0b', 'db',...
    'e0', '32', '3a', '0a', '49', '06', '24', '5c',...
    'c2', 'd3', 'ac', '62', '91', '95', 'e4', '79',...
    'e7', 'c8', '37', '6d', '8d', 'd5', '4e', 'a9',...
    '6c', '56', 'f4', 'ea', '65', '7a', 'ae', '08',...
    'ba', '78', '25', '2e', '1c', 'a6', 'b4', 'c6',...
    'e8', 'dd', '74', '1f', '4b', 'bd', '8b', '8a',...
    '70', '3e', 'b5', '66', '48', '03', 'f6', '0e',...
    '61', '35', '57', 'b9', '86', 'c1', '1d', '9e',...
    'e1', 'f8', '98', '11', '69', 'd9', '8e', '94',...
    '9b', '1e', '87', 'e9', 'ce', '55', '28', 'df',...
    '8c', 'a1', '89', '0d', 'bf', 'e6', '42', '68',...
    '41', '99', '2d', '0f', 'b0', '54', 'bb', '16'
};

% All possible 8-bit key
ke = 0:255;

% Compute the V matrix (all plain text & all keys)
V = zeros(num_trace,length(ke));

for i=1:num_trace
    for j=1:length(ke)
        V(i,j) = bitxor(pt(i),ke(j));        % XorKey()
        V(i,j) = hex2dec(Scell(V(i,j)+1));  % SubBytes()
    end
end

% Compute the H matrix (hamming weight of V)
H = zeros(num_trace,length(ke));
for i=1:num_trace
    for j=1:length(ke)
        H(i,j) = sum(dec2bin(V(i,j))=='1'); % Hamming weight
%           H(i,j) = bitget(V(i,j),1);
    end
end

%% Measurement
% Compute the T matrix (power vs. time data traces)
T = zeros(num_trace,num_time);
for i=1:num_trace
    fid = fopen(['1_' num2str(i-1)]);
    temp = fscanf(fid, '%g %g', [2 inf]);    % read the traces
    T(i,:) = temp(2,:);                        % get the power data only
    fclose(fid);
end

% Compute the R matrix (correlation of H and T)
R = zeros(length(ke),num_time);
for i=1:length(ke)
    for j=1:num_time
        R(i,j) = corr(H(:,i),T(:,j));    % correlation
        if(isnan(R(i,j)))
            R(i,j) = 0;                  % remove NaN
        end
    end
```

```
end
%%
[val,pos] = max(R');
[val2, pos2] = max(val);

dec2hex(pos2)

hold all;
plot(R');
xlabel('Sample');
ylabel('Correlation');
%%
figure;
plot(R','k');
hold on;
plot(R(pos2,:),'r');
hold on;
plot(R(1,:),'g');
```