Highly secure cryptographic computations against side-channel attacks

Yiruo He

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
He, Yiruo, "Highly secure cryptographic computations against side-channel attacks" (2013). Electronic Theses and Dissertations. 4717.
https://scholar.uwindsor.ca/etd/4717
HIGHLY SECURE CRYPTOGRAPHIC COMPUTATIONS AGAINST SIDE-CHANNEL ATTACKS

by

Yiruo He

A Thesis
Submitted to the Faculty of Graduate Studies
Through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

2012

© 2012 Yiruo He
HIGHLY SECURE CRYPTOGRAPHIC COMPUTATIONS AGAINST SIDE-CHANNEL ATTACKS

by

Yiruo He

APPROVED BY:

__________________________
Dr. Huiming Zhang
Department of Biological Sciences

__________________________
Dr. Mitra Mirhassani
Department of Electrical and Computer Engineering

__________________________
Dr. Huapeng Wu, Advisor
Department of Electrical and Computer Engineering

__________________________
Dr. Roberto Muscedere, Chair of Defense
Department of Electrical and Computer Engineering

19 October, 2012
DECLARATION OF CO-AUTHORSHIP/PREVIOUS PUBLICATION

I. Co-Authorship Declaration

I hereby declare that this thesis incorporates material that is result of joint research. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the author and Dr. H. Wu as advisor.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Declaration of Previous Publication

This thesis includes 1 original paper that has been previously published/submitted for publication in peer reviewed journals, as follows:

<table>
<thead>
<tr>
<th>Thesis Chapter</th>
<th>Publication title/full citation</th>
<th>Publication status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 4</td>
<td>Efficient Architectures for Modular Exponentiation Using Montgomery Powering Ladder, IEEE Canadian Conference on Electrical and Computer Engineering 2011 (CCECE 2011), May 8-11, 2011 Niagara Falls, Canada</td>
<td>Published</td>
</tr>
</tbody>
</table>

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Windsor.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone’s copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis.
I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.
ABSTRACT

Side channel attacks (SCAs) have been considered as great threats to modern cryptosystems, including RSA and elliptic curve public key cryptosystems. This is because the main computations involved in these systems, as the Modular Exponentiation (ME) in RSA and scalar multiplication (SM) in elliptic curve system, are potentially vulnerable to SCAs. Montgomery Powering Ladder (MPL) has been shown to be a good choice for ME and SM with counter-measures against certain side-channel attacks. However, recent research shows that MPL is still vulnerable to some advanced attacks [21, 30 and 34]. In this thesis, an improved sequence masking technique is proposed to enhance the MPL’s resistance towards Differential Power Analysis (DPA). Based on the new technique, a modified MPL with countermeasure in both data and computation sequence is developed and presented. Two efficient hardware architectures for original MPL algorithm are also presented by using binary and radix-4 representations, respectively.
ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Huapeng Wu for introducing me to Montgomery Powering Ladder and giving me all that advice, help and support throughout this research work. I also want to thank Dr. Mitra Mirhassani for the invaluable feedbacks she given me on this thesis, and Dr. Huiming Zhang for extensive point of view.

I would also grateful to my colleagues and friends, Wangcheng Dai, Xiaojing Yu, Rodney Mclean and Karl Leboeuf, for their time and support.
# TABLE OF CONTENTS

DECLARATION OF CO-AUTHORSHIP/PREVIOUS PUBLICATION ............................ iii

ABSTRACT ............................................................................................................ v

ACKNOWLEDGEMENTS ....................................................................................... vi

LIST OF TABLES ................................................................................................... x

LIST OF FIGURES ................................................................................................ xi

LIST OF ALGORITHMS ........................................................................................ xii

LIST OF ACRONYMS ........................................................................................... xiii

CHAPTER

I. INTRODUCTION .............................................................................................. 1

II. PUBLIC KEY CRYPTOGRAPHY AND SIDE CHANNEL ATTACK

2.1 Asymmetric Cryptography ................................................................. 4
2.2 RSA and Elliptic Curve Cryptosystem ............................................ 5
2.3 Modular Exponentiation and Montgomery Powering Ladder ....... 8
2.4 Side Channel Attacks .......................................................................... 11
   2.4.1 Power Analysis Attack ............................................................... 12
   2.4.2 Timing Attack ........................................................................... 14
   2.4.3 Fault Attack .............................................................................. 14
   2.4.4 EM Attack ................................................................................ 14
2.5 Giving Protections in Algorithm Level .............................................. 15
III. EXISTING WORK REVIEW AND PROBLEM STATEMENT

3.1 DPA against MPL ................................................................. 18
3.2 Coron’s three countermeasures in ECC and RSA ................. 21
   3.2.1 First countermeasure: Randomization of the Private Exponent ...................................................... 22
   3.2.2 Second countermeasure: Blind the Point P ...................... 22
   3.2.3 Third countermeasure: Randomization of Projective Coordinates ...................................................... 22
   3.2.4 First countermeasure in RSA: Randomization of the Private Exponent ...................................................... 23
   3.2.5 Second countermeasure in RSA: Randomization of the Message ...................................................... 24
   3.2.6 Comparative Power Anlaysis against Coron’s Work ...... 25
3.3 Follow up Countermeasures on Exponent and Message ........... 27
   3.3.1 Exponent Splitting .......................................................... 27
   3.3.2 High Orders Attack against Exponent Splitting ............. 28
   3.3.3 Blinded Fault Resistant Exponentiation .......................... 30
   3.3.4 Template Attack against Masked MPL .......................... 32
3.4 Sequence Masking .............................................................. 33

IV. PROPOSED PARALLEL COMPUTATION MONTGOMERY POWERING LADDER ARCHITECTURE

4.1 Proposed Architecture for Montgomery Power Ladder ............. 35
4.2 Proposed Modified Montgomery Power ladder ...................... 38
4.3 Proposed Architecture for the Radix-4 MPL ......................... 39

V. PROPOSED NOVEL SEQUENCE MASKING TECHNIQUE

5.1 Applying on MPL ................................................................. 42
5.2 Security Analysis .............................................................. 43
VI. PROPOSED COUNTERMEASURES FOR MONTGOMERY POWER LADDER

6.1 Algorithm Explanation .................................................................45
6.2 Efficiency Analysis........................................................................48
6.3 Security Analysis...........................................................................48
  6.3.1 Against Simple Side Channel Attacks .........................48
6.3.2 Against Relative Doubling Attack and Comparative Power Analysis .................................................................49
6.3.3 Against Template Attack......................................................51
6.3.4 Against High Order Attack and Complex Attacks ..........52
6.4 Summary.......................................................................................54

VII. HARDWARE IMPLEMENTATION FOR PROPOSED COUNTERMEASURE

7.1 SASEBO-GII..................................................................................57
7.2 HDL Simulation..............................................................................59
7.3 Synthesis Results ......................................................................59
7.4 Summary.......................................................................................60

VIII. CONCLUSION

8.1 A Summary of Contributions .....................................................65
8.2 Conclusion....................................................................................65
8.3 Possible Future Work .................................................................66

REFERENCES....................................................................................67

APPENDICES......................................................................................72

VITA AUCTORIS................................................................................85
LIST OF TABLES

2.1 Side Channels and Corresponding Side Channel Attacks ........................................ 11

2.2 Comparison between MPL and Square-and-Multiply Method on simple SCA resistance. ......................................................................................................................... 16

3.1 Probability transition for different exponent bits ............................................................ 28

3.2 Imbalance probability for Exponent Splitting .................................................................. 29

6.1 Efficiency analysis for proposed Algorithm 6.1 ............................................................. 48

6.2 Countermeasures versus SCAs .......................................................................................... 55

7.1 Hardware usage of FPGA implementation of proposed modified MPL (Algorithm 6.1) .................................................................................................................................... 59

7.2 Hardware usage of different algorithms implemented on SASEBO-GII .................... 61

7.3 Exponentiation circuit performance for 1024 bit ............................................................ 61
LIST OF FIGURES

2.1 Message delivery in Public Key Cryptography ........................................... 4
2.2 SPA reveals secret exponent in binary method ........................................... 13
2.3 Comparison of Square-and-Multiply and MPL in power traces ................. 16
3.1 Example of Relative Doubling Attack ....................................................... 20
3.2 A. Shamir’s pattern for Exponent Masking [22] ....................................... 23
3.3 Comparative Power Analysis example[21] ................................................ 26
4.1 Architecture for Montgomery Powering Ladder ....................................... 36
4.2 An Implementation of the 2-by-2 Cross-point Switch in Figure 7 ............... 37
4.3 Proposed Architecture for the Modified MPL (Algorithm 4.1) ................. 39
4.4 Two Shift Register Holding Exponent Bits .............................................. 40
6.1 Algorithm 6.1 against Relative Doubling Attack ....................................... 50
7.1 Top-level block diagram of SASEBO-GII .............................................. 58
7.2 Architecture of proposed modified MPL with countermeasures ............... 62
7.3 Block diagram of Cryptographic FPGA for realizing proposed work ....... 63
7.4 On board waveform of data outputting .................................................. 64
## LIST OF ALGORITHMS

1. Left to right version of Square-and-Multiply Method ........................................... 8
2. Montgomery Powering Ladder .................................................................................. 10
3. Masked Montgomery Powering Ladder .................................................................... 30
4. Square-and-Multiply Always method ....................................................................... 33
5. Proposed Modified Montgomery Powering Ladder ..................................................... 38
6. Proposed sequence masking applying on MPL ............................................................. 42
7. Proposed Modified MPL with countermeasures ......................................................... 46
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIST</td>
<td>National Institute of Advanced Industrial Science and Technology of Japan</td>
</tr>
<tr>
<td>BUFG</td>
<td>Global Buffer</td>
</tr>
<tr>
<td>DPA</td>
<td>Differential Power Analysis</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic Curve Cryptosystem</td>
</tr>
<tr>
<td>ECDLP</td>
<td>Elliptic Curve Discrete Logarithm Problem</td>
</tr>
<tr>
<td>EM</td>
<td>Electromagnetic</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-programmable Gate Array</td>
</tr>
<tr>
<td>FTDI</td>
<td>Future Technology Devices International</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IP cores</td>
<td>Intellectual Property cores</td>
</tr>
<tr>
<td>LUT</td>
<td>Look Up Table</td>
</tr>
<tr>
<td>ME</td>
<td>Modular Exponentiation</td>
</tr>
<tr>
<td>MPL</td>
<td>Montgomery Powering Ladder</td>
</tr>
<tr>
<td>MVN</td>
<td>Multivariate Normal Distribution</td>
</tr>
<tr>
<td>RNS</td>
<td>Residue Number System</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest, Shamir, Adleman</td>
</tr>
<tr>
<td>SASEBO</td>
<td>Side-channel Attack Standard Evaluation Board</td>
</tr>
<tr>
<td>SCA</td>
<td>Side Channel Attack</td>
</tr>
<tr>
<td>SPA</td>
<td>Simple Power Analysis</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

The Internet has grown rapidly and it becomes a ubiquitous part of our modern society. One of the cornerstones for its success is users’ trust on secure data transaction over the Internet. Cryptography provides many core services for the network security to ensure such trust. For instance, public key cryptography is famous for its strong security strength and is frequently used as initial key exchange between two parties over insecure communication channel. However, in many practical scenarios, attackers are able to access the cryptographic device and gain information about internal data by monitoring the physical information released from the device. Such attacking methodology is introduced in [24], [25] and named Side Channel Attacks (SCA).

Modular exponentiation (ME), the most demanded computation in RSA public key cryptosystem, is extensively targeted by SCAs. Unprotected ME algorithm offers various possibilities for SCAs because the “leakage” physical information released by cryptographic device is greatly associated with ME algorithm. Carefully designed ME algorithms with proper countermeasures could result in more regular side channel signals which may not be taken advantage of by the attackers easily. For instance, Montgomery Powering Ladder (MPL) can provide resistance to one of the most popular SCAs, Simple Power Analysis (SPA). Because MPL always maintains regular operations throughout its process and has no redundant computations, the corresponding power consumption signals release little information making SPA no longer applicable.

It has been shown that MPL is not immune to all types of SCAs. For example, MPL remains sensitive to Differential Power Analysis (DPA) which is a powerful SCA
that extracts the leakage of information related to power consumption. Differential Power Analysis (DPA) is first described by Kocher et al. in [10]. Many follow ups can be found in [17, 20, 21, 25, and 30] and among which, [21, 30] are specially designed attacks for breaking MPL.

Coron pointed out [9] that DPA may be prevented by randomizing the group, the exponent or the base element. The following research work was focusing on masking techniques targeting the exponent and the base element which can be shown as examples in [22, 26, and 28]. However, they have all been proven ineffective toward later proposed attacks [21, 27, 29] respectively. Other than protecting the exponent and the base element, another algorithmic countermeasure is proposed by changing the procedure of ME algorithm and is referred to as Sequence Masking technique in [21]. One example is Square-and-multiply-always method which is effective in hiding computation sequences but vulnerable to safe-error attack.

In this thesis an improved sequence masking technique is proposed. Based on the proposed technique, a modified MPL algorithm with countermeasures of randomization on exponent and the base element is developed. It has been shown that the new modified MPL algorithm could provide protection to more SCAs than any other existing MPL-like algorithms.

The thesis is organized as the following chapters. Chapter II gives an overview of asymmetric cryptography system where MPL is put into application and introduces SCA and explains why it is a serious threat. MPL as well as its natural resistance toward SCAs is also depicted. Then, Chapter III explains the philosophy that existing works use to stop SCA. Then it states the existing works have very little power before certain advanced
SCAs, thus new countermeasures are in need.

Chapters IV to VII depict proposed works. In Chapter IV, two efficient architectures for modular exponentiation are proposed respectively using MPL algorithm and radix-4 MPL algorithm. It follows a novel sequence masking technique, which is described in Chapter V. In Chapter VI, a new modified MPL algorithm with countermeasures is proposed and analyzed. Its hardware implementation is described in the following chapter VII. Chapter VIII concludes the contributions of this thesis and describes some possible future work.
2.1 Asymmetric Cryptography

In key generation point of view, there are two types of cryptographic techniques, namely, symmetric cryptography and asymmetric cryptography. Symmetric cryptography uses the same key for both encryption and decryption, while asymmetric cryptography differentiates decryption key from encryption key. Asymmetric cryptography is also popularly known as public key cryptography.

Assume that Alice and Bob are parties engaging a secure communication using cryptographic technique. In asymmetric cryptography, each of them has her/his own pair of public key and private key. The public key is placed in a public register accessible to the public while the private key is kept private and known to its owner only.

![Diagram of message delivery in Public Key Cryptography]

Figure 2.1 Message deliveries in Public Key Cryptography

In a scenario that Alice would like to send a confidential message to Bob, she looks up Bob’s public key and uses it as encryption key during encryption process. Upon
receiving the encrypted message from Alice, Bob interprets this message only by using his private key. In this example, two different keys are evolved in encryption and decryption process. The encryption key is Bob’s public key which is revealed to public. The decryption key is a private key that is only known by Bob. Thus, as the scheme illustrated in Figure 2.1, the scheme is able to allow Alice sending messages to Bob privately. Since Bob holds the only key that can be used to decrypt the encrypted message.

It is well known that the feature of making a distinction between encryption and decryption key for public key cryptography can facilitate many unique cryptographic/secure services such like digital signature and key exchange. However, public key cryptography systems usually require significant higher computation cost than symmetric key systems as a trade off. Specifically, longer computation time and more memory room requirement are usually expected in public key cryptography systems. Therefore, it is very important to develop efficient algorithms for public key cryptosystems. For the popular public key cryptosystems such like RSA, the main computation cost is spent in performing modular exponentiation. This is why the research on efficient modular exponentiation algorithms has becomes a focus in this area.

In real world practice of network security, symmetric and asymmetric cryptography are co-operated. Asymmetric cryptography realizes the initial key exchange with strong security strength, while encryption/decryption process is achieved by the low cost symmetric cryptography process. In conclusion, both cryptography systems play critical role. Their cooperation balances security strength and efficiency in network security.
2.2 RSA and Elliptic Curve Cryptosystem

Two popular cryptosystems, RSA and ECC are explained in this section. RSA is a wildly used asymmetric cryptosystem and digital signature scheme. The invention of this scheme was in later 1970s at MIT, by Ron Rivest, Adi Shamir and Len Adleman in [1]. And such cryptosystem is named after the first digit of their last name. The strength of RSA is its mathematic difficulty in factorize $\varphi(n)$ where $\varphi(n) = (p - 1)(q - 1)$. The description of such system is given as follows.

The cryptosystem holds the public key defined as $(n, e)$ and the private key defined as $(n, d)$. Where, the integer $n$ is obtained by multiply two prime number $p$ and $q$. $e$ and $d$ are exponents and they satisfy such requirements.

$$ ed \equiv 1 \mod \varphi(n) \text{ where } \varphi(n) = (p - 1)(q - 1) $$

The encryption and decryption process of RSA can be described as follows. Alice wants to send encrypted message to Bob. Thus, she use Bob’s public key $(n,e)$ to compute $c \equiv m^e \mod n$ where $c$ is the encrypted message. The legitimate receiver, Bob for this case, is able to decrypted $c$ through his own private key $(n,d)$ by computing $m \equiv c^d \mod n$. The underlying equation can prove such encryption and decryption is valid and original message are assured to be successfully delivered.

$$ c^d \equiv (m^e)^d \equiv m^{(ed)} \equiv m^1 \equiv m \mod n $$

RSA system can also be used as digital signature scheme. Bob wants to identify Alice. Therefore, Alice use her own private key $(n,d)$ to “sign” a message to tell Bob “I am Alice”. Such purpose is fulfilled by computing $c \equiv m^d \mod n$. Since the private key $(n,d)$ is unique, she is the only one in this world who can create such signature. And
Alice’s signature can be easily verified by decrypting using Alice’s public key \((n,e)\) by computing \(m \equiv c^e \mod n\). The verification of such scheme is simple and straightforward.

\[
c^e \equiv (m^d)^e \equiv m^{(ed)} \equiv m^1 \equiv m \mod n
\]

According to [2], “the most notable features about RSA are its apparent simplicity and considerable elegance”. This sentence perfectly concludes RSA. Because of its simplicity and elegance, RSA is abundantly applied in the world of cryptography.

Different than RSA, Elliptic Curve Cryptosystem (ECC) has not been patented. The suggestion of using elliptic curve in cryptography is first published in [33] in 1985. This system uses the elliptic curve discrete logarithm problem (ECDLP) which can be defined as follows:

Let \(E(\mathbb{F}_q)\) to be an elliptic curve over \(\mathbb{F}_q\) and let \(P\) be a point in such curve. For any point \(R \in E(\mathbb{F}_q)\) find the integer \(k\), where \(0 \leq k \leq \#P - 1\) (\(\#P\) is the order of \(P\)) such that \(kP = R\) is an ECDLP. The basic operation in ECC is scalar multiplication \(kP = \{P + P + \cdots + P\}\), there are \(k\) number of times additions.

A quick example will illustrate as follows. Alice tries to send a signed message to Bob. They both share a point \(P\) on the same elliptic curve \(E\). Bob has a private key \(s_B\) and private key \(p_B\) satisfy \(p_B = s_BP\). Because of the difficulty on solving ECDLP, making known to public of \(p_B\) and point \(P\) will not reveal private key \(s_B\).

Alice randomly generates an integer \(r\) and compute \(rP\). And then she encrypts the message \(m\) by computing \(c = m + rp_B\). where \(p_B\) is the public key of Bob. At last, Alice sends \((rP, c)\) to Bob.
Bob decrypts the message from Alice by multiplying his secret key $s_B$ with value of $rP$. And then he subtracts the product from $c$. Since $p_B = s_BP$, thus $rs_BP = rp_B$. Then the message is successfully decrypted as follows:

$$c - s_BrP = m + rp_B - s_BrP = m$$

It is obviously shown that modular exponentiation is exhaustively used and appears nearly in nearly every derivation equations in RSA. On other hand, ECC has huge amount of scalar multiplication which shares many commons with modular exponentiation. The significance of modular exponentiation is self-evident. In summary, the research work on modular exponentiation has its meaning reflected in widely usage in popular public key cryptography like RSA and ECC etc.

### 2.3 Modular Exponentiations and Montgomery Powering Ladder

<table>
<thead>
<tr>
<th>Algorithm 2.1. Left to right version of Square-and-Multiply method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $M$, $e=(e_{n-1} \ldots e_1e_0)_2$</td>
</tr>
<tr>
<td><strong>Output:</strong> $C = M^e$</td>
</tr>
<tr>
<td><strong>Step 1:</strong> Set $R \leftarrow M$;</td>
</tr>
<tr>
<td><strong>Step 2:</strong> For $i = n-1$ to 0 Step -1</td>
</tr>
<tr>
<td><strong>Step 2a:</strong> $R \leftarrow R^2$;</td>
</tr>
<tr>
<td><strong>Step 2b:</strong> if $(e_i=1)$</td>
</tr>
<tr>
<td>Then $R \leftarrow R \times M$;</td>
</tr>
<tr>
<td><strong>Step 3:</strong> Return $(C = R)$</td>
</tr>
</tbody>
</table>

One of the most naive fast modular exponentiation methods is Binary Method which represents the exponent in binary form. The basic idea is to take advantage of
binary expression of the exponent to do faster computation rather than direct multiply the base by some number of times.

Binary Method, also known as Square-and-multiply is a very classical method and it is over 2000 years old. The left to right version starts at the exponent’s most significant non-zero bit and work downward to least significant bit. Pseudo code illustration is shown as follows. It shows in Algorithm 2.1 that at the beginning of each loop, the value in register R is squared. And whenever exponent bit $e_i = 1$, R is multiply by M. An equation is able to prove this algorithm’s correctness.

$$e = e_{n-1} \times 2^{n-1} + e_{n-2} \times 2^{n-2} + \ldots + e_1 \times 2^1 + e_0 \times 2^0 = \ldots (e_{n-1} \times 2 + e_{n-2}) \times 2 + \ldots + e_0$$

Module exponentiation is the most critical part because it is the most demanded computation in public key cryptography and have greatly influence to the computation overhead. Unfortunately it’s also the weakest and most vulnerable point in front of SCAs. In most case, since the secret information is part of the exponentiations parameters and the surrender of modular exponentiation will direct result in the compromise of secret information. Having resistance to Side Channel Attacks in Modular Exponentiation is very important. With such background, the application of Montgomery Powering Ladder [3] in cryptography causes great excitement for its nature resistance to Simple Power Analysis (SPA). A comparison between Binary Method and MPL is disclosed in section 2.5 of this chapter to show MPL’s advantages towards SPA. And in this section, we still focus on what is MPL.

MPL was originally invented as an improvement of left to right binary algorithm towards SPA. And it is based on the following observation. [16]

Let $L_i = \sum_{i=j}^{t-1} e_i 2^{i-j}$, $H_i = L_i + 1$
It is easy to get:

\[
\begin{align*}
L_i &= 2L_{i+1} + k_i = H_{i+1} + L_{i+1} + k_i - 1 \\
H_i &= H_{i+1} + L_{i+1} + k_i
\end{align*}
\]

That shows the relationship between \(L_i\) and previous cycle value \(L_{i+1}\) and also the relationship between \(H_i\) and previous cycle value \(H_{i+1}\). Moreover, we also found that to obtain \(L_i\) and \(H_i\), there are always exist iteration where:

When \(e_i = 0\),

\[
\begin{align*}
L_i &= 2L_{i+1} \\
H_i &= H_{i+1} + L_{i+1}
\end{align*}
\]

And when \(e_i = 1\),

\[
\begin{align*}
L_i &= H_{i+1} + L_{i+1} \\
H_i &= 2H_{i+1}
\end{align*}
\]

Notice that \(L_i\) and \(H_i\) have very similar expression structure. And \(L_i\) doubles itself when \(e_i = 0\) and \(H_i\) doubles itself when \(e_i = 1\). Moreover, the summation of \(H_{i+1} + L_{i+1}\) is assigned to \(H_i\) in case of \(e_i = 0\) and to \(L_i\) in case of \(e_i = 1\).

---

**Algorithm 2.2. Montgomery Powering Ladder**

**Input:** \(M, e=(e_{n-1} \ldots e_1 e_0)_2\)  
**Output:** \(C = M^e\)  

Step 1: Set \(R0 \leftarrow 1, R1 \leftarrow M;\)  
Step 2: For \(i = n-1\) to \(0\) Step -1  
Step 2a: if \((e_i=0)\)  
   Then \{Set \(R1 \leftarrow R0 \times R1, R0 \leftarrow R0^2;\}\}  
Step 2b: if \((e_i=1)\)  
   Then \{Set \(R0 \leftarrow R0 \times R1, R1 \leftarrow R1^2;\}\}  
Step 3: Return \((C = R)\)
As shown in Step2a and 2b of Algorithm 2.2, register R0 and R1 have corresponding iteration process as H_i and L_i in the exponent. The summation of H_{i+1} and L_{i+1} in exponent is carried out by a multiplication between R0 and R1. And the doubles in H_{i+1} and L_{i+1} is realized by a squaring operation on R1 and R0 respectively. These operations are valid since the base value is always M throughout the exponentiation.

Consider the computation overhead, MPL takes 2\log n multiplications on average. This may not as good as the performance in Square-and-multiply which takes around 3/2\log n on average. Nevertheless, the capability for parallel computing makes this method more efficient than basic binary algorithms. In [16], Marc Joye and Sung-Ming Yen exhibit that as:

$$R_{1-e_j} = R_0 \times R_1 \quad \text{and} \quad R_{e_j} = R_{e_j}^2$$

where \(R_{e_j}\) could be either \(R_1, R_0,\) and \(R_{1-e_j}\) is the negation of \(R_{e_j}\)

It is obviously that calculations relate to \(R_{1-e_j}\) and the ones relate to \(R_{e_j}\) are independent. So, on a bi-processor, multiplication and squaring can compute at the same time. That results parallel version of the MPL nearly attains the optimal 200% speed-up factor over the standard one [16]. According to MPL’s capability of parallel computing, two efficient architectures are proposed in Chapter IV in this thesis.

2.4 Side Channel Attacks

Side channel attacks exploit the information leaked by the physical characteristics of the cryptographic modules during execution of the algorithm. The term “side channel” is used to describe the leakage of system information. Depends on what kind of leakage of system information SCA relies on, we can categorized it into several types as shown in Table 2.1.
<table>
<thead>
<tr>
<th>Side Channels</th>
<th>Side-channel Attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Consumptions</td>
<td>Simple Power Analysis, Differential Power Analysis, Comparative Power Analysis etc</td>
</tr>
<tr>
<td>Timing information</td>
<td>Timing Attacks</td>
</tr>
<tr>
<td>Faults response</td>
<td>Safe-Error Attacks</td>
</tr>
<tr>
<td>Electromagnetic Radiation</td>
<td>EM Attacks</td>
</tr>
</tbody>
</table>

-2.4.1 Power analysis attack

Different operation in cryptographic algorithms consumes different powers. And these power variations can leak useful information about secret parameters. In worse case, the secret parameters can be fully recovered by careful statically analysis on these leakage information. Power analysis attacks proved to be very effective in attacking smart cards and other embedded systems. And it can be categorized into Simple and Differential Power Analysis (SPA and DPA respectively). In SPA, measured power traces are used for analyzing which particular instruction is being carried out at specific time. And this knowledge can lead to expose of secret parameters. DPA exploits more statistical method in analysis process. And it is considered as one of the most powerful SCAs for it requires relatively little resources [25]. Some of the DPAs will be detailed explained in chapter III section 3.4 in this thesis. Currently, we focus on describe the mechanism of SPA.
In order to illustrate the idea of SPA, consider an RSA encryption involves the computation of $C = M^e \mod n$, where $n$ is modulus and $M$ is the message need to be encrypted. Adversary’s goal is to know the secret key $e$.

![Power Trace Diagram]

Figure 2.2 SPA reveals secret exponent in binary method

In Square-and-Multiply algorithm (Algorithm 2.1), different instructions are carried out according to the value of secret exponent. In Step2b, the multiplication is conditional and only occurs at the case of $e_i = 1$. In the other case of $e_i = 1$, only squaring operation is performed. In another word, if adversary knows how to identify this conditional multiplication, he knows the value of secret exponent. Unfortunately, the multiplication is distinguishable in power consumption signals. As illustrate in Figure 2.2, each wave pulse represents the power consumption of running an operation. Operations could only be squaring or multiplication. Compare to multiplication, squaring operation usually consumes less power and therefore has lower amplitude in power traces. As a result, power traces can fully disclose which operation the cryptosystem was running by identifying the amplitude difference. Operation types are record at the top of corresponding wave pulses in Figure 2.2 where S represent squaring and M represents...
multiplication. In the case of an S is followed by an M, which is highlighted as red wave pulses in Figure 2.2, the corresponding secret exponent must be 1 since the conditional multiplication is carried out and the triggering condition must be satisfied. Otherwise, as shown in green, only squaring is performed which indicates the secret exponent is 0.

2.4.2 Timing attack

Cryptographic algorithms in majority of implementation execute the computations in a non-constant time. And these time variations sometimes related to secret exponent. Moreover, careful statically analysis on this leakage information may fully recover the secret exponent. First timing attack is proposed by Kocher et al. on 1996 [24]. He shows it is possible to use such timing attack to against RSA. More works can be found in [6, 10].

2.4.3 Fault Attack

Fault Attacks try to introduce errors into cryptographic computation, and to identify the key by analyzing the mathematical and statistical properties of the erroneously computed results. [22]

As illustration of the attack scheme, one of fault based attack mentioned by Sung-Ming Yen and Marc Joye is explained as follows. In [12, 13], they descript the attack like this: By timely inducing a fault during the execution of an instruction, an attacker may deduce whether the targeted instruction is redundant: if the final result is correct then the instruction is indeed redundant (or dummy operation [13]); if not, the instruction is effective. This knowledge may then be used to obtain one or more bits of exponent. Such attacks are referred to as safe-error attack. Since safe-error attack is able to check the
effectiveness of each operation, it is dangerous to have dummy operations in cryptographic algorithms. More fault attacks can be found in [11, 34].

2.4.4 EM Attack

Electromagnetic (EM) radiation is considered as an extension of the power consumption leakage and the attacks/countermeasures are applied without change [17]. Instead of measuring power consumptions, Electromagnetic radiation can be an alternative leakage source used by adversary. More EM works can be found in [35].

2.5 Giving Protections in Algorithm Level

One of the practical approaches to stop SCAs is to provide protections in ME algorithm level as refers to countermeasures. In this section, comparison between Square and Multiply Algorithm and MPL are given to show how improvements in algorithm level enhance its SCA resistances.

Square-and-Multiply is vulnerable to SPA because it has a conditional statement that makes system operating differently and thus results in different power consumption. This has already been discussed in section 2.4.1 as a demonstration of SPA.

MPL has shown more reliable resistance to SPA. First of all, as shown in Algorithm 2.2, it always performs a multiplication followed with a squaring. Consequently, there is no difference in power consumption regarding to computation on different exponents. Secondly, there is no dummy operation in the algorithm. The faults induced by safe-error attack always results in an incorrect exponentiation result. Thus, no leakage information will release in erroneously computed results.
Table 2.2 Comparisons between MPL and Square-and-Multiply on simple SCA resistance

<table>
<thead>
<tr>
<th></th>
<th>Square-and-Multiply</th>
<th>MPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication Numbers</td>
<td>2log n in worst case</td>
<td>2log n constantly</td>
</tr>
<tr>
<td></td>
<td>1.5log n on average</td>
<td></td>
</tr>
<tr>
<td>Resistance to SPA types attack</td>
<td>Vulnerable</td>
<td>Resistive</td>
</tr>
<tr>
<td>Resistance to Safe-error attack</td>
<td>Vulnerable</td>
<td>Resistive</td>
</tr>
</tbody>
</table>

As illustrate in Figure 2.3, a comparison is performed in ME algorithms regarding to power traces. Since Multiplication and Squaring is distinguishable by the amplitude, the operation types are disclosed by the power traces and record as M and S in the figure. Moreover, the secret exponents are listed below the exact wave pulse which is generated by computing such exponent. It is obvious that Square-and-Multiply and MPL have different power traces with inputting same secret exponents. Square-and-Multiply has recognizable power traces for conditional multiplications. On the other hand, MPL consistently performs multiplication with a squaring. In [14], “Highly Regular” is used to evaluate an exponentiation algorithm. First, the algorithm is regular; which means will always repeat the same instructions in the same order for any inputs; second, it has no dummy operation which refers to the non-function operation padded within algorithm. Dummy operation doesn’t effective but in some cases it has to be executed. MPL is a “Highly Regular” exponentiation algorithm because the operation of MPL satisfies both
requirements. Meanwhile, Square-and-Multiply fails on the first requirement of “Highly Regular”.

In conclusion, when taking into account simple Side channel attacks resistance, MPL has better performance than Square-and-multiply algorithm. The essential of such improvement is more regular instructions in iteration process. And the comparison also demonstrates that improvements in algorithm level can regulate the power consumptions and further enhance the SCA resistance.
CHAPTER III
EXISTING WORK REVIEWS

According to recent research, MPL is able to resist SPA but still vulnerable to Differential Power Analysis (DPA). Therefore, many research works are proposing on MPL’s DPA countermeasures. Several existing works are mentioned in this chapter for their great inspiration and influence. Respect has to be given to those pioneers in this research field. In addition, their weaknesses are also concluded to show the potential of further improvements.

3.1 DPA against MPL

In order to illustrate MPL’s vulnerability to DPA, Relative Doubling Attack is explained as an example in this section. Relative Doubling Attack is proposed in [30]. Because the original doubling attack [20] does not apply to MPL, S.M Yan and etc noticed another doubling-like attack is applicable. It is based on the following observation:

Recall the invention of MPL algorithm, Low and High Registers were defined as:

When \( k_i = 0 \), \( \begin{align*}
    L_i &= 2L_{i+1} \\
    H_i &= H_{i+1} + L_{i+1}
\end{align*} \)

, and when \( k_i = 1 \), \( \begin{align*}
    L_i &= H_{i+1} + L_{i+1} \\
    H_i &= 2H_{i+1}
\end{align*} \).

It’s easy to notice two facts.

Fact1. Given \( e_i = 0 \), then we have \( L_i = 2L_{i+1} \).

Fact2. Given \( e_i = 1 \), then we have \( H_i = 2H_{i+1} \).
Assume two exponentiations are computing using MPL. They have specific
inputs M and M^2. Therefore, the whole process is computing M^e mod N and (M^2)^e mod
N, where k is exponent and should be keep secret all the time. And N is modular number.

It’s easy to obtain that if e_i = e_{i-1} = 0 then, two squaring are performed as
follows:

\[
\begin{aligned}
R_0 &\leftarrow (M^{L_i})^2: \text{Step 2a of iteration } i - 1 \text{ when evaluating } M^e \\
R_0 &\leftarrow ((M^2)^{L_{i+1}})^2: \text{Step 2a of iteration } i \text{ when evaluating } (M^2)^e
\end{aligned}
\]

These two squaring are computing the same values because of L_i = 2L_{i+1}. Due to
this observation of “Collisions” on computation, a new doubling-like attack can be
mounted to derive the knowledge of e_i = e_{i-1} = 0. Once two computations are found not
coincidentally identical, the triggering condition would be known if such computations are
spotted. Such computations are called “Collisions”.

For the same reason, it’s also clear that if k_i = k_{i-1} = 1 then,

\[
\begin{aligned}
R_0 &\leftarrow (M^{H_i})^2: \text{Step 5 iof iteration } i - 1 \text{ when evaluating } M^e \\
R_0 &\leftarrow ((M^2)^{H_{i+1}})^2: \text{Step 5 iof iteration } i \text{ when evaluating } (M^2)^e
\end{aligned}
\]

These two squaring are also performing same computation because H_i = 2H_{i+1}.
And such leads to the knowledge of e_i = e_{i-1} = 1. And for all the other case of k_i \neq
k_{i-1}, there are not any collisions in the computation process.

Figure 3.1 demonstrates an example of spotting collisions in power traces may
harm the cryptosystem. Assume two separate messages are input to the system. These
two inputs are carefully chosen as M and M^2. The corresponding power traces for
exponentiation of M^e and (M^2)^e are indicated in Figure 3.1. Meanwhile, the
corresponding values in registers R0 and R1 are recorded under the exact power pulses.
Notice, these values and secret bits are used to provide better understanding of what the
internal data is in real time. They are no shown to the public. Only the power traces can be obtained from public.

For case 
\[ e_i = e_{i-1} = 0 \text{ or case } e_i = e_{i-1} = 1, \]
or in another word, two adjacent zeros or ones in secret keys, a pair of squaring in adjacent iterations is processing the same data. As seen in highlighted blocks. Two collision was generated since there is two “1” in a row and two “0” in a row in exponent bits. The first collision results in two identical operations \( M^2 \rightarrow M^4 \) in target and reference power traces. The second collision can be spotted in two \( M^6 \rightarrow M^{12} \) computations.

![Figure 3.1 Example of Relative Doubling Attack](image)

In [30], collision of two same squaring within \( M^e \) and \( (M^2)^e \) at adjacent iteration lead to the knowledge of equivalence between two neighboring key bits. And since two
collisions are not distinguishable, the detection of collision will not reveal the value of the operand directly. However, the attacker is still able to conclude that once collision is detected, two adjacent key bits are the same. Otherwise, they are different. As a result, for given any bit in exponent, it is not difficult to figure out the rest. In addition, the most significant bit of exponent is often to be chosen as one. With such awareness, the private exponent is no longer secret.

Relative Doubling Attack is very effective against MPL. It proves that MPL is considerable unsecure in front of DPAs. More other attacks can be found in [21, 34].

3.2 Coron’s Three DPA Countermeasures in ECC and RSA

As illustrated in previous section, MPL cannot resist DPAs. As a result, many researchers are working on MPL’s DPA countermeasures. In [19], J.S. Coron inclusively concludes three types of DPA countermeasures in Elliptic Curve System. These countermeasures are based on randomizing different parameters of scalar multiplication $Q = dP$. And inducing randomization in scalar multiplication and modular exponentiation are well accepted method to against DPAs. Coron’s idea can be further extended into RSA cryptosystem. The third countermeasure of Coron refers to randomize the Group in RSA. This idea is not included in thesis. Thus, just first two countermeasures are detailed mentioned in this section. In the last, the weakness of Coron’s work is explained in a specific example: Comparative Power Analysis [21].

3.2.1 First countermeasure: Randomization of the Private Exponent

Let $#\mathcal{E}$ be the total number of points in Elliptic Curve. The scalar multiplication $Q = dP$ can be realized by two steps.
1. Compute \( d' = d + k \cdot \#e \), where \( k \) is a random number and its size is suggested to be 20 bits in practice.

2. Compute the point \( Q = d'P \).

Exponent \( d \) can be replaced by \( d' \) in realization, because \( \#eP = 0 \). And

\[
Q = d'P = (d + k \cdot \#e)P = dP + k \cdot \#e \cdot P = dP + 0 = dP
\]

This countermeasure transfer scalar multiplication \( Q = dP \) to a new computation \( Q = d'P \). Since exponent \( d \) and \( d' \) are related, the computation result \( Q \) will still be same.

-\( 3.2.2 \) Second countermeasure: Blind the point \( P \)

The point \( P \) is masked by adding a random point \( R \) which also belongs to the same curve. And also \( S = dR \) is known. Then scalar multiplication can be computed by \( d(R + P) \). To recover \( Q = dP \) is just subtract \( d(R + P) \) with \( S \). The mathematical proof is as follows.

\[
Q = d(R + P) - S = dR + dP - s = dP
\]

This countermeasure transfer scalar multiplication \( Q = dP \) to a new computation \( Q = d(R + P) - S \). According to the above equation, the computation result is correct.

-\( 3.2.3 \) Third countermeasure: Randomization of Projective Coordinates

The projective coordinates of a point are not unique thus the projective coordinates of \( P=(X, Y, Z) \) can be randomized by inducing a random number \( \lambda \). \( P \) is represented in a new projective coordinates of \( (\lambda X, \lambda Y, \lambda Z) \) where \( \lambda \neq 0 \) in the finite field. This countermeasure protects the binary representation of \( P \) in projective coordinates.
-3.2.4 First countermeasure in RSA: Randomization of the Private Exponent

This countermeasure is also known as Exponent Masking. As the name indicates, it masks the exponent in order to protect the cryptosystem. The exponent masking technique for RSA is firstly disclosed in [22] invented by Adi Shamir.

For computing $C = m^e \pmod{n}$, instead of set the exponent as $e$, we choose alternative exponent $e'$, where $e' = e + r \phi(n)$ and $\phi(n)$ is the totient of modular $n$, and $r$ is a random number. Since:

$$C = m^{e'} \pmod{n} = m^{e+r\phi(n)} \pmod{n} = (m^e \times (m^{\phi(n)})^r) \pmod{n}$$

And because $m^{\phi(n)} \pmod{n} \equiv 1$, we could easily verify that

$$C = (m^e \times (m^{\phi(n)})^r) \pmod{n} = (m^e \times (1)^r) \pmod{n} = m^e \pmod{n}$$

In Shamir’s Patten, the computation of $C = m^{e'} \pmod{n}$ has the same result as the expected $C = m^e \pmod{n}$. However, it actually computes different operands. The physical performance of alternative exponentiation is totally different, including power
consumptions and EM radiation features. Therefore, the private exponent is protected even if the whole computation is compromised. The adversary only knows $e'$ is computed but still have no idea about original exponent $e$. Moreover, if the $\mathcal{O}(n)$ is a relative small number, this technique can be very efficient. For instance, if $n$ and $d$ are 1024 bit numbers, and $r$ is a 32 bit random number as it recommended, $d + r \times t$ is a 1056 bit number consequently and it only need to take extra 32 multiplications or squaring. As a result, this technique only cost $32/1024 \approx 3\%$ extra computation.

-3.2.5 Second countermeasure in RSA: Randomization of the Message

This countermeasure is also known as Message Masking. Also as the name indicates, it masks the message to be encrypted to prevent the potential attacks. For computing $C = m^e \pmod{n}$, in order to confuse the adversary, message $m$ is transformed into other format. Following Coron’s idea, message $m$ is randomized by multiplying with a random number $R$. Thus, the computation is transformed into $C' = (mr)^e \pmod{n}$. We say it is masked by random number $r$. It’s obviously that $C'$doesn’t match the ordinary $C$. In order to recover the ordinary $C$, $C'$ need to be unmasked by multiply an anti-mask $(r^{-1})^e$.

$$C = C'(r^{-1})^e \pmod{n} = (mr)^e \times (r^{-1})^e \pmod{n} = m^e \pmod{n}$$

In order to evaluate a message masking technique, the complexity of “Mask Updating” is always the most important criterion. For Coron’s second countermeasure, the mask $r$ will be updated as the following equation specify.

$$r' = r^e \pmod{n} \text{where } e \text{ is the number of iteration}$$
If the mask won’t change as stays always as \( r \), its update pattern is considered as a relatively weak masking technique. But Coron’s second countermeasure update the mask in a stable pattern and it is stronger than fixed masks.

3.2.6 Comparative Power Analysis against Coron’s Work

Unfortunately, Coron’s two masking countermeasures are vulnerable to the proposed attack as suggested in [21]. Comparative power analysis attack is proposed by N. Homma et.al in [21]. It can be applied to many standard implementations of the exponentiation, for instance, the binary Method, M-ary Methods and MPL. Similar to Relative Doubling Attack [30] mentioned in section 3.1, the basic idea of this attack is to input a pair of chosen messages to generate collisions. The two chosen inputs \( Y \) and \( Z \) have to be able to find the solution of \( Y^\alpha \equiv Z^\beta \) so that it can generate collisions. Computing \( Y \) as exponent gives a power trace including the target operation. The other input \( Z \) gives another power trace used as reference for it has a particular operation which is identical with target operation. In contrary to Relative Doubling Attack [30], the Collision was generated at two arbitrary time frames. And it’s claimed in [21] that, the two inputs have more flexible relationship.

With the intention to find \( Y \) and \( Z \) to satisfy \( Y^\alpha \equiv Z^\beta \), the attacker can choose an arbitrary value \( r \) and can compute \( Y = r^\beta \mod N \) and \( Z = r^\alpha \mod N \), where \( \alpha \) and \( \beta \) can be user customized.

Let’s have an example for better understanding. In figure 5.2, the input condition was chosen as \( Y^\alpha \equiv Z^\beta \), where \( \alpha = 13 \) and \( \beta = 2 \). The attacker was assuming to know the first four bits are \( 1100_2 \). If the fifth bit is one, the attacker can detect that collision was generated in two squaring operations at highlighted time frame. Otherwise, the fifth
bit is considered as zero. It’s simple to notice that the binary representation of decimal 13 is $11001_2$, and first four bits are $1100_2$ which is treated as attacker’s knowledge in the first place. Thus if the fifth key bits is one, the squaring at that time frame is computing $Y^{13}$.

![Figure 3.2 Comparative Power Analysis Examples](image)

In the beginning, the attacker would choose the input condition $Y^{13} \equiv Z^2$ according to his knowledge of revealed key bits. And then he can figure out the other input $Z$ to create reference operation $Z^2$. If fifth bit is one and because $Y^{13} \equiv Z^2$; we are expecting similarity for target and reference operation. Otherwise, fifth bit more likely to be zero. And after repeated attacks, the secret key bits will be exposed one by one.

And according to the analysis in [21], the Comparative Power Analysis is capable for cracking algorithm that carries both Coron’s second countermeasure [19] and Shamir’s exponent masking technique [22]. A valid example was shown in [21], assume input $X$ is randomized by Coron’s second countermeasure as $rX \mod N$ and $r$, where $r$ is
a random number. Meanwhile, exponent E is randomized with a multiple of \( \phi(n) \). The attacker will simple choose input \( X = -1 \). Thus, the exponentiation of \( X^E \) will turns into \((-r)^E\). At the same time, the updating process for mask is essentially \( r^E \). Notice that they are taking the same exponent. And with simple comparison in power traces, the randomized exponent \( E + i \phi(n) \) will be uncovered. Although the real exponent does not yield, \( E + i \phi(n) \) is equally useful. If the attacker repeats such attack, he would get another randomized exponent \( E + i' \times \phi(n) \) with same E but different i. the subtraction for \( E + i \phi(n) \)and \( E + i' \times \phi(n) \) will gives a multiple of \( \phi(n) \), which is sufficient to factorize N.

3.3 Follow up Countermeasures on Exponent and Message Masking

Two follow up countermeasures on exponent and message masking are brought up in this section, Exponent Splitting [26] and Blinded Fault Resistant Exponentiation [28]. These countermeasures are very effective and inspiring. Reviewing such countermeasures helps understanding the recent research results for masking technique. Their weaknesses are also included to show how they fail towards later proposed attacks. More specifically, High Order attack [27] is able to break Exponent Splitting Technique. And Masked MPL is vulnerable to Template Attack [29]. It also helps understanding the proposed algorithm presented later in this thesis.

-3.3.1 Exponent Splitting

The idea of data splitting was first abstracted in [26]. And in [23], the idea was used specific on exponent. Based on the simple observation of:

\[
m^e = m^{r+(e-r)} = m^r \times m^{e-r}
\]
C. Clavier and M. Joye states in [23] that values of both \( r \) and \((e-r)\) are required to recover the value of \( e \). In other word, only one of the two exponentiations requires protection. Even though this statement was proved to be wrong in [27], it still gives an idea on we could split the exponent to thwart side channel attacks.

The main idea of the splitting technique is to pick a random \( r \) (smaller than \( e \)) and to compute the value \( r' = e - r \). After that the recovery process is completed fairly easy by computing

\[
S = S_r \times S_{r'} = M^{(r+r')} \mod N = M^e \mod N
\]

Exponent splitting technique has very high security strength but the cost is severe. Naturally it doubles the computation load. Thus it is considered less efficient than other alternative algorithms. Unfortunately, such technique is compromised to attack proposed in [27] which is explained the following section. Further enchantment for such technique is necessary.

### 3.3.2 High Orders Attack against Exponent Splitting

Table 3.1 Probability transition for different exponent bits

<table>
<thead>
<tr>
<th>( \Pr(r_i r_{i-1}) )</th>
<th>( E_i = 0 )</th>
<th>( E_i = 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Pr(0,0) )</td>
<td>( 0.5 \times P_i )</td>
<td>( 0.5 \times (1 - P_i) )</td>
</tr>
<tr>
<td>( \Pr(0,1) )</td>
<td>( 0.5 \times (1 - P_i) )</td>
<td>( 0.5 \times P_i )</td>
</tr>
<tr>
<td>( \Pr(1,0) )</td>
<td>( 0.5 \times (1 - P_i) )</td>
<td>( 0.5 \times P_i )</td>
</tr>
<tr>
<td>( \Pr(1,1) )</td>
<td>( 0.5 \times P_i )</td>
<td>( 0.5 \times (1 - P_i) )</td>
</tr>
<tr>
<td>( P_i+1 )</td>
<td>( 0.5 \times P_i )</td>
<td>( 0.5 \times (1 - P_i) )</td>
</tr>
</tbody>
</table>

High Orders Attack is proposed in [27] by Frederic Muller and Frederic Valette. They discovered a hidden weakness of Exponent Splitting technique. That weakness was initiated with a very tricky statistic property. Such property stays in the probability transitions of carry bits for different exponent bits.
Table 3.2 Imbalance probability for Exponent Splitting [27]

<table>
<thead>
<tr>
<th>(r, r')</th>
<th>E_0</th>
<th>E_1</th>
<th>E_2</th>
<th>E_3</th>
<th>E_4</th>
<th>E_5</th>
<th>E_6</th>
<th>E_7</th>
<th>E_8</th>
<th>E_9</th>
<th>E_{19}</th>
<th>E_{20}</th>
<th>E_{21}</th>
<th>E_{22}</th>
<th>E_{23}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pr(0,0)</td>
<td>0</td>
<td>25</td>
<td>38</td>
<td>31</td>
<td>35</td>
<td>33</td>
<td>34</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>.......</td>
<td>47</td>
<td>23</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>Pr(0,1)</td>
<td>50</td>
<td>25</td>
<td>12</td>
<td>19</td>
<td>15</td>
<td>17</td>
<td>16</td>
<td>34</td>
<td>41</td>
<td>46</td>
<td>.......</td>
<td>3</td>
<td>27</td>
<td>39</td>
<td>45</td>
</tr>
<tr>
<td>Pr(1,0)</td>
<td>0</td>
<td>25</td>
<td>13</td>
<td>18</td>
<td>15</td>
<td>17</td>
<td>16</td>
<td>33</td>
<td>42</td>
<td>46</td>
<td>.......</td>
<td>4</td>
<td>28</td>
<td>40</td>
<td>46</td>
</tr>
<tr>
<td>Pr(1,1)</td>
<td>50</td>
<td>25</td>
<td>37</td>
<td>32</td>
<td>35</td>
<td>33</td>
<td>34</td>
<td>17</td>
<td>9</td>
<td>4</td>
<td>.......</td>
<td>46</td>
<td>22</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

The following equation is always satisfied:

\[ C_i \oplus r_i \oplus r_i' = E_i \]

where \( C_i \) is the carry bit in i-th iteration and \( r_i \) and \( r_i' \) refer to the two random numbers that construct the real exponent E.

If we define \( P_i \) as the probability for the case of \( C_i = 0 \), and \( \Pr(r_i r_i') \) as the probability for bracket case, we could have the probability transaction as summarized in Table 3.1. The probability for bracket cases belongs to a Markov chain, where next step’s probability can be derived from two previous probability transaction expressions. An example will further explain.

Two random numbers \( r_i, r_i' \) are generated to construct real exponent E for \( r + r' = E \). The Table 3.2 records the probabilities of all pair of \( r_i \) and \( r_i' \) associated with the real exponent.

For \( E_0 \) is 0, then \( r_0 \) and \( r_0' \) can either be 00 or 11, each has 50% chance. If \( E_0 \) is 1, then \( r_0 \) and \( r_0' \) will be either 10 or 01, each has 50% chance. We notice that after a long run of 0s in exponent bits, the \( R \) is approaching to zero indicating no carry bits generated. And after a long run of 1s, \((1-P_i)\) is very close to zero showing a carry bit is propagating along with the computation. In fact, the statistical probability for \( r \) and \( r' \) infers the value.
of actual exponent. If adversary launch any attack methods suggested in [27], exponent splitting is not long safe.

### 3.3.3 Blinded Fault Resistant Exponentiation

**Algorithm 3.1. Masked Montgomery Powering Ladder**

**Input:** \( M, e=(e_{n-1} \ldots e_1e_0)_2 \); 

**Output:** \( C = M^e \in \mathbb{G} \)

1. Pick Random Number \( r \)
2. Set \( R_0 \leftarrow r, R_1 \leftarrow r \cdot M, R_2 \leftarrow r^{-1} \in \mathbb{G} \)
3. For \( i = n-1 \) to 0 Step -1
   4. if \( e_i = 0 \) Then \( \{ R_1 \leftarrow R_0 \cdot R_1, R_0 \leftarrow R_0^2, R_2 \leftarrow R_2^2, \text{ update } (\text{CKS}, e_j) \} \)
   5. if \( e_i = 1 \) Then \( \{ R_0 \leftarrow R_0 \cdot R_1, R_1 \leftarrow R_1^2, R_2 \leftarrow R_2^2, \text{ update } (\text{CKS}, e_j) \} \)
6. \( R_2 \leftarrow R_2 \oplus \text{CKS} \oplus \text{CKS}_{\text{ref}} \)
7. Return \( C = R_0 \cdot R_2 \)

Blinded Fault Resistant Exponentiation is also known as Masked Montgomery Powering Ladder (AKA Masked MPL). It is first proposed in [28] by G. Fumaroli and D. Vigilant in 2009. It is a message masking technique based on Montgomery Powering Ladder algorithm. At the very beginning, two register \( R_0 \) and \( R_1 \) is multiplicatively blinded by random picked number \( r \) in the same Group. All the intermediate values of \( R_0 \) and \( R_1 \) are masked by the element \( r^{2^{n-1}} \in \mathbb{G} \).
The register R2 is initialized with the anti-mask \( r^{-1} \in \mathbb{G} \), and such anti-mask is also updating during each iteration process. As a result, after \( n \) number of times iterations, the register R2 would hold \( r^{-2^n-1} \in \mathbb{G} \). And multiply R0 and R2 give the precise exponentiation results. In addition, in order to thwart potential fault attack and exponent or loop counter disturbance, an on-the-fly checksum function was used to fulfill such purpose.

The updating pattern for the mask is \( r^{2^n-1} \in \mathbb{G} \). Compare to Coron’s second Countermeasure’s mask updating as \( r^n \), Masked Montgomery Powering Ladder has better randomness in mathematic point of view. It’s obvious that taking \( n \) as parameter, \( r^{2^n-1} \) has high order. And we are expecting more variation on the change for higher orders.

Since Masked MPL keeps the same structure as the regular MPL, it inherits Montgomery Powering Ladder’s feature of Highly Regular, it’s intrinsically resistive to simple side-channel attacks as well as Safe-Error Attack.

Other than resistance to simple side channel attack, Masked MPL contains improved resistance towards Differential Power Analysis and Fault Attacks. By means of masking all the computation intermediate value, the input is believed to be “statistically independent” [28] from output. Unless the random number \( r \) is revealed, or it is a weak mask, differential side-channel attacks can not apply in practice. And thanks to the Checksum function’s participation, most fault attack cannot pass the very last sum checking. Failure in such checking will cause the calculated results wiped.
3.3.4 Template attack against Masked MPL

Masked MPL is considered as a very strong countermeasure. Nevertheless, a template attack [29] is claimed to be a great threat to Masked MPL.

C. Herbst and M. Medwed proposed a crypto-analysis in [29] that building a template to guess the operand of given operation by maximum-likelihood decision rule. It has been proved that it is effective to attack Masked Montgomery Ladder via guessing the value of random mask. Such template represents statistical properties of the power consumption for a given operation. It states in [29] that the power consumption of a device follows a multivariate normal distribution (MVN). Similarly like MVN, the power consumptions can be described by template consisted by a mean vector $\mathbf{m}$ and a covariance matrix $\mathbf{C}$. And it also assumes that the adversary can model every possible occurring operation. Therefore, the adversary is able to fully characterize all possible operations and know the corresponding hamming weights.

Since the adversary also knows the moments of time when the mask is operating, he can extract those points and apply the previously built templates to them. Therefore, adversary has the knowledge of the Hamming height of the mask as well as those of the partial products of the multiplication.

So far, the adversary successfully extracts the Hamming weights of the processes data out of a given trace. In this case, the attack focuses on the masking operation $r \times x$. Here starts the stage of so called Sieving Step [29] which can determine the mask $r$. The first part of sieving is to narrow down the mask candidate by knowing the exact Hamming weight. The second part is checking the hamming weight of partial products lead by left mask candidate.
The effectiveness for such attack is the same in 8bits and 16 bits system. However, for 32-bit platforms the sieving step becomes computationally infeasible. Although such attack is limited in low bit platforms so far, the solid standing for masked MPL has been challenged. Further enhancement is under demands.

3.4 Sequence Masking

Algorithm 3.2 Square-and-Multiply Always method

<table>
<thead>
<tr>
<th>Input: M, e=(e_{n-1} \ldots e_1 e_0)_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: C = M^e</td>
</tr>
<tr>
<td>Step 1: Set R← M;</td>
</tr>
<tr>
<td>Step 2: For i = n-1 to 0 Step -1</td>
</tr>
<tr>
<td>Step 2a: R ← R^2;</td>
</tr>
<tr>
<td>Step 2b: if (e_i=1)</td>
</tr>
<tr>
<td>Then R ← R×M;</td>
</tr>
<tr>
<td>Step 2C: if (e_i=0)</td>
</tr>
<tr>
<td>Dummy Operation R×M;</td>
</tr>
<tr>
<td>Step 3: Return (C = R)</td>
</tr>
</tbody>
</table>

There is another method to against DPA other than Coron’s countermeasures. Sequence masking technique usually changes the procedure of exponentiation methods. One unsuccessful example is Square-and-Multiply-always method [19]. This method adds a dummy multiplication to standard Square-and-Multiply method to make it more balanced. As illustrated in Algorithm 3.2. When the exponent bit is equal to zero, the multiplication that is not necessary but used as a cover which can be referred as dummy
operation. However, such dummy operation is vulnerable to safe-error attack and didn’t improve the strength of original algorithm. It becomes very easy to locate when it computes the multiplication \( R = R \times M \) after the squaring. And when exponent bit is equal to zero at this iteration, \( R = R \times M \)’s result won’t affect the final result. This can be abused conversely. If some computational fault is induced to system when it computes \( R = R \times M \), it’s easy to know the exponent bit at that time frame by verifying whether the final result is correct or not.

Sequence masking receives less attention and there is very few existing work. First, for given ME algorithm, it is difficult to changing the computation sequence. Non careful adjusting may ruins the computation correctness. Second, adding redundant operation is dangerous; it can be seen in example of Square-and-Multiply Always method. Nevertheless, sequence masking is a possible solution for DPA protection. And if it’s well adopted, such kind of technique is additive to exponent masking and message masking techniques.
CHAPTER IV

PROPOSED ARCHITECTURES FOR MPL AND RADIX-4 MPL

In this chapter, two new architectures for exponentiation are proposed. The first one is an efficient implementation of Montgomery power ladder algorithm (Algorithm 3.2) by using its parallel computing feature. The second proposed architecture is based on a modified Montgomery powering ladder method (Algorithm 4). We firstly extend Montgomery ladder algorithm by applying loop unrolling technique to it. The resultant algorithm takes only half number of the loops to complete the exponentiation. A new architecture for this modified Montgomery ladder algorithm is then proposed. The hardware complexity and time delay of the proposed architectures are analyzed and compared.

4.1 Proposed Architecture for Montgomery Power Ladder

An efficient architecture for realization of MPL (Algorithm 2.2) is shown in Figure 4.1. Two registers R0 and R1 store the variables R0 and R1 in Algorithm 2.2, and they are initialized as 1 and M respectively. Registers R0 and R1 should be larger enough to hold the power $M^k$. The exponent $k$ is stored in the binary shift register $k$ which shifts to the left by one bit every clock cycle. (In Figure 4.1, it is shown as a circular shift register.) Other hardware components include one modular multiplier, one modular squaring unit, one multiplexer, and one 2-by-2 cross-point switch.

Assume that the modulus is $M$ and has $m$ bits. Then each of registers R0 and R1 should be large enough to hold an $m$-bit number. Modular multiplier and modular squaring unit take input operand(s) of $m$-bit and generate output of $m$-bit. The
The multiplexer takes two inputs of m-bit number and selects one of them as the output depending on the select bit $k_i$.

Figure 4.2 shows one design of the 2-by-2 cross-point switch shown at the bottom of Figure 4.1. The implementation of the switch utilizes two multiplexers and it realizes the following function:

If $E=0$, then $C=A$, $D=B$;
If $E=1$, then $D=A$, $C=B$.

The architecture works as follows. Registers $R_0$ and $R_1$ are initially loaded as 1 and $M$, respectively. At cycle $j$, $j=0, 1, \ldots, n-1$, exponent bit $k_{n-1-j}$ is the leftmost bit in Register $k$ and controls both the multiplexer and the 2-by-2 cross-point switch. If $k_{n-1-j}=0$, the output $R_0$ of register $R_0$ is selected by the multiplexer and upon which the
squaring operation is performed. Otherwise if \( k_{n-1,j} = 1 \), the output of register R1 is selected (R1) by the multiplexer and squaring operation is performed to generate R12.

![Diagram of 2x2 cross-point switch](#)

Figure 4.2 Implementation of the 2-by-2 cross-point switch in Figure 4.1

The 2-by-2 cross-point switch works as follows. At cycle \( j \), if \( k_{n-1,j} = 0 \), or the control input to the switch \( E = 1 \), the switch is configured as two cross paths where the output of the multiplier is connected to the input to R1 and the output of the squarer is connected to the input to R0. If \( k_{n-1,j} = 1 \), or \( E = 0 \), the 2-by-2 switch is configured into two parallel paths. The output of the multiplier is then written into R0 while the output of the squaring unit is written into R1.

During clock cycle \( j \) the architecture completes the computation in loop \( i=j \) in Algorithm 2.2. After \( n \) clock cycles, Register R0 contains the final result \( C = M^k \).

The complexity of the architecture includes one multiplier, one squarer, two multiplexers, and two registers. The critical path delay \( T \) is given by

\[
T = \max \{ T_{\text{Multiplier}}, T_{\text{Squarer}} + T_{\text{Mux}} \} + T_{2\times2}
\]

\[
= \max \{ T_{\text{Multiplier}} + T_{\text{Mux}}, T_{\text{Squarer}} + 2T_{\text{Mux}} \}
\]

It can be seen from Figure 4.2 that the time delay of the 2-by-2 cross-point switch is equivalent to that of one multiplexer. If we assume \( T_{\text{Multiplier}} >> T_{\text{Squarer}} \) for very
large operand, then the critical path delay is \( T = T_{\text{Multiplier}} + T_{\text{Mux}} \). The time delay
taken to complete one exponentiation is \( nT = n \left( T_{\text{Multiplier}} + T_{\text{Mux}} \right) \).

4.2 Proposed Modified Montgomery Power Ladder

We apply the loop unrolling technique to the existing Montgomery power ladder
algorithm by unrolling two loops into one. The resultant algorithm is shown in Algorithm 4.1 as follows.

<table>
<thead>
<tr>
<th>Algorithm 4.1. Modified Montgomery Powering Ladder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: ( M, e=(e_{n-1} \ldots e_1 e_0)_{2} )</td>
</tr>
<tr>
<td>Output: ( C = M^e )</td>
</tr>
<tr>
<td>Step 1: Set ( m \leftarrow (n-2)/2 ) if ( n ) is even;</td>
</tr>
<tr>
<td>otherwise set ( m \leftarrow (n-1)/2 ) and ( kn \leftarrow 0 ).</td>
</tr>
<tr>
<td>Step 2: Set ( R_0 \leftarrow 1, R_1 \leftarrow M );</td>
</tr>
<tr>
<td>Step 3: For ( i = n-1 ) to 0 Step -1</td>
</tr>
<tr>
<td>Step 3a: if ( k_{2i+1}k_{2i} = 00 )</td>
</tr>
<tr>
<td>Then { Set ( R_1 \leftarrow R_0 \times R_1, R_0 \leftarrow R_0^2 ),</td>
</tr>
<tr>
<td>( R_1 \leftarrow R_0 \times R_1, R_0 \leftarrow R_0^2 ) };</td>
</tr>
<tr>
<td>Step 3b: if ( k_{2i+1}k_{2i} = 01 )</td>
</tr>
<tr>
<td>Then { Set ( R_1 \leftarrow R_0 \times R_1, R_0 \leftarrow R_0^2 ),</td>
</tr>
<tr>
<td>( R_0 \leftarrow R_0 \times R_1, R_1 \leftarrow R_1^2 ) };</td>
</tr>
<tr>
<td>Step 3c: if ( k_{2i+1}k_{2i} = 10 )</td>
</tr>
<tr>
<td>Then { Set ( R_0 \leftarrow R_0 \times R_1, R_1 \leftarrow R_1^2 ),</td>
</tr>
<tr>
<td>( R_1 \leftarrow R_0 \times R_1, R_0 \leftarrow R_0^2 ) };</td>
</tr>
<tr>
<td>Step 3b: if ( k_{2i+1}k_{2i} = 11 )</td>
</tr>
<tr>
<td>Then { Then { Set ( R_0 \leftarrow R_0 \times R_1, R_1 \leftarrow R_1^2 ),</td>
</tr>
<tr>
<td>( R_0 \leftarrow R_0 \times R_1, R_1 \leftarrow R_1^2 ) } };</td>
</tr>
<tr>
<td>Step 4: Return ( C = R_0 )</td>
</tr>
</tbody>
</table>
4.3 Proposed Architecture for the Modified Montgomery Power Ladder Algorithm

The proposed architecture to implement Algorithm 3 is shown in Figure 4.3. Two register R0 and R1, initialized as 1 and M stores the variables R0 and R1 in Algorithm 4.1, respectively.

Figure 4.3 Proposed Architecture for the modified MPL (Algorithm 4.1)

The two register should be large enough to hold the power $M^k$. The binary exponent k is split into two parts and they are stored in two shift registers, as shown in Figure 4.4.
As shown in Figure 4.4, Register K₀ stores all the odd bits of the exponent k, ..., k<sub>2i+1</sub>, ..., k₃, k₁, whose output bit is used to control the top multiplexer and the top 2-by-2 cross-point switch (Figure 4.2). Register K₁ stores all the even bits of k, k<sub>2i</sub>, ..., k₂, k₀, and its output bit controls the bottom multiplexer and the bottom 2-by-2 switch as shown in Figure 4.3. Other units include two multipliers, two squaring units, two multiplexers, and two 2-by-2 cross-point switches. The architecture can be roughly divided into two parts: the upper part works similar to that in Figure 4.1, except that the outputs of the 2-by-2 switch become the inputs to the multiplier and squaring unit at the lower part, rather than are written back into R₀ and R₁ in Figure 4.1. The lower part of the architecture works also similar to that in Figure 4.1 except that the input to the multiplier and squaring units are the 2-by-2 switch in the upper part, rather than from the registers as in Figure 4.1.

The complexity of the architecture includes two multipliers, two squaring units, two multiplexers, two 2-by-2 switches, and two registers. The critical path delay T is given by

\[ T = \max \{2T_{\text{multiplier}} + 2T_{2x2}, T_{\text{multiplier}} + T_{\text{squarer}} + 2T_{2x2} + T_{\text{Mux}}, 2T_{\text{squarer}} + 2T_{2x2} + 2T_{\text{Mux}} \} \]
\[ = \max\{2T_{\text{multiplier}} + 2T_{2x2}, T_{\text{multiplier}} + T_{\text{squarer}} + 3T_{\text{Mux}}, 2T_{\text{squarer}} + 4T_{\text{Mux}}\} \]

Note that the delay of the 2-by-2 cross-point switch is equivalent to that of one multiplexer. The number of clock cycles required to complete one exponentiation is \([(n + 1)/2]\).
CHAPTER V

PROPOSED NOVEL SEQUENCE MASKING TECHNIQUE

In this chapter, a novel sequence masking technique is proposed. This technique is explained through the case of MPL. Then security analysis of propose technique is followed. Even the proposed technique can only resist the

5.1 Applying on MPL

Algorithm 5.1 Proposed sequence masking applying on MPL

Input X, N, E₀ = (eₙ₋₁eₙ₋₂...e₃e₀); E₁ = (eₙ₋₁eₙ₋₂...e₁e₀)
Output: X^{E₀} \text{mod N} and X^{E₁} \text{mod N}
Generating 2n bits random number Seq;
Step 1: Set R₀ = 1 ; R₁ = X ; R₂ = 1 ; R₃ = X.
Step 2: For i=2n-1 down to 0 do
Step 2.1: if Seqᵢ = 0 , then
Step 2.1a: if MSB of E₀ = 0, then
R₀ = R₀ × R₁ ; R₁ = R₁²
Step 2.1b: if MSB of E₀ = 1, then
{R₁ = R₀ × R₁ ; R₀ = R₀²
Step 2.1c: E₀ shift to left;
Step 2.2: if Seqᵢ = 1 , then
Step 2.2a: if MSB of E₁ = 0, then
{R₂ = R₂ × R₃ ; R₃ = R₃²
Step 2.2b: if MSB of E₁ = 1, then
{R₃ = R₂ × R₃ ; R₂ = R₂²
Step 2.2c: E₁ shift to left;
Step 3: End for
Step 4: Return R₀ and R₂
Proposed sequence masking technique can be applied to standard exponentiation methods. As a demonstration, the application on MPL is illustrated in Algorithm 5.1. The intention of this technique is to do two the modular exponentiations by a single computation core but in irregular sequences. Therefore, the computation sequence is randomized. In Algorithm 5.1, proposed masking technique creates a longer iteration sequence contributed by two exponentiations. At the beginning of iteration parts, the algorithm computes one of the two exponentiations depends on the value of Seq\_i where \( i \) represent the number of iteration. Corresponding pairs of registers are also chosen to participate in the computation. The core operations match MPL algorithm with extra bit shift of exponent in the last step. After iteration part finished, two exponentiation results are stored in \( R_0 \) and \( R_2 \) separately.

Two exponentiations \( X^{E_0} \) and \( X^{E_1} \) have exponent \( E_0 \) and \( E_1 \) respectively.

\[
E_0 = \{e_{n-1}, e_{n-2}, \ldots, e_1, e_0\}_2 \ n \text{ bits long}
\]

\[
E_1 = \{e_{n-1}, e_{n-2} \ldots, e_1, e_0\}_2 \ n \text{ bits long}
\]

Instead of computing them one after another, the iteration processes of the two exponentiations are merged into one. Since they are computed by a single computation core, if one is under computation, the other one is paused and all the intermediate values are stored in the memory. The switch point is decided by a 2n bits random number Seq. Each bit in Seq represents which exponentiation would be computed during this iteration.

5.2 Security Analysis

The complexity of randomness induced can be represented by number of different computation sequences generated by proposed technique. For two practical 1024 bits
exponents, there could have \( C_{1024}^{512} = \frac{1024!}{(512! \times 512!)} \) which is an incredible big number. Therefore, the proposed technique is able to prevent Brute Force Attack.

And for many existing DPAs mentioned in chapter III, they are based on the assumption that target operation is always easy to locate and for sure occurs at same time frame over and over again. The intention of this technique is to randomize the computation sequence and therefore operations are hard to locate. The iteration process is doubled in proposed technique; the updating of iteration number is not longer fixed. Operations are no longer predictable and therefore randomness is induced in iteration process of exponentiation. An operation appears at same time frame in different attempts can belong to two different operations. Consequently, even the collision actually happens, it is difficult for the attackers to locate where it is. Thus, the collision itself can’t reveal any useful information.

However, since these two exponentiations are independent. Attacker can induce fault to one of them to reveal the other exponentiation. Thus, the use of this technique must be careful. It also requires two exponentiations put together, sometimes this condition might not be applied. Nevertheless, the proposed technique can be further develop into a more complex countermeasure by combining with two exiting ideas which is fully described in next chapter.
CHAPTER VI

PROPOSED MODIFIED MPL WITH COUNTERMEASURES

In this chapter, based on previous proposed sequence masking technique, a modified MPL with countermeasures (Algorithm 6.1) is developed. It has similar structure as Algorithm 5.1. And Algorithm 6.1 does not only involve proposed sequence masking technique but also borrows two existing ideas. Exponent is randomized with the idea in [23], which is discussed in section 3.3.1. At same time, G. Fumaroli’s idea [28] mentioned in section 3.3.2 randomized the message. Borrowed ideas are also improved in proposed modified MPL algorithm with countermeasure. With adequate arrangement, the extra updating operation of message anti-mask is removed. More importantly, the vulnerability of [23] and [28] which are mentioned in section 3.3.2 and 3.3.4 are either eliminated or appended with proper protection.

6.1 Algorithm Explanation

The pre-computation is constructed several steps. The first step is random number generation. The original exponent E can be divided into two equal size randomized exponent $r_1$ and $(E - r_1)$. Another n bits random number $r_2$ is utilized as random mask. Moreover, $r_2^{-1}$ is also brought into play with role of random mask for the second exponentiation and the anti-mask for the first exponentiation. After then, $r_2^{-1}$ is naturally updated along with the second exponentiation as the same pattern as the mask $r_2$ in the first exponentiation. The update pattern would be $r_2^{2n-i_0} \in \mathbb{G}$ and $r_2^{-2n-i_1} \in \mathbb{G}$. 

45
Algorithm 6.1. Proposed Modified MPL with countermeasures

Input X, N, e=(en−1 ... e1 e0)2 ;

Output: C =X^e

Pre-computation:

Step 1. Generating n bits random number r1, r2 ,
and 2n bits random number Seq

Step 2. Assign: E_0 = r_1; E_1 = (E − r_1); R_0 = r_2 ;
R_1 = r_2 × X; R_2 = r_2^{-1}; R_3 = r_2^{-1} × X.

Step 3. Initial CKS_0, CKS_1

Computation:

Step 4: For i=2n-1 down to 0 do

Step 4.1: if Seq_i = 0 , then

Step 4.1a: if MSB of E_0 = 0, then
{R_0 = R_0 × R_1 ; R_1 = R_1 ^ 2}

Step 4.1b: if MSB of E_0 = 1, then
{R_1 = R_0 × R_1 ; R_0 = R_0 ^ 2}

Step 4.1c: E_0 shift to left;
Update (CKS_0,E_0);

Step 4.2: if Seq_i = 1 , then

Step 4.2a: if MSB of E_1 = 0, then
{R_2 = R_2 × R_3 ; R_3 = R_3 ^ 2}

Step 4.2b: if MSB of E_0 = 1, then
{R_3 = R_2 × R_3 ; R_2 = R_2 ^ 2}

Step 4.2c: E_1 shift to left;
Update (CKS_1,E_1);

Step 5: End for

Step 6:

R_0 = R_0 ⊕ CKS_0 ⊕ CKS_{ref#0}
R_2 = R_2 ⊕ CKS_1 ⊕ CKS_{ref#1}

Step 7: Return R_0 = R_0 × R_2
The last step for pre-computation is initialization of a similar fault detection method proposed in [28]. It uses Checksum function to prevent possible fault attacks. Notice that all the random number generated in the pre-computation will refresh at the beginning of new round of exponentiation.

In the iteration process, two sets of exponentiation are taking turns to compute according the value of Seq at that iteration. Each set exponentiation has its own pair of registers for storing the intermediate values. Register R_0 and R_1 are reserved for first exponentiation and R_2 and R_3 are used only by second exponentiation. Accordingly, values between two exponentiations won’t cross over each other. Whenever Seq equals to zero, R_0 and R_1 are computed. Otherwise, R_2 and R_3 are involved instead. After the iteration part, the fault detection is implemented by the XOR computation between final results of each exponentiation and Checksums. And final adjustment is followed as the product between R_0 and R_2.

The correctness proof is demonstrated as follows.

The first exponentiation is \( r_2^{2^{n-l_0}} \times X^{r_1} \) and second exponentiation is \( r_2^{2^{n-l_1}} \times X^{E-r_1} \), where \( i_0, i_1 \) represent the individual iteration number for first and second exponentiation respectively. They are not counting in the algorithm since they always satisfy \( i_0 + i_1 = i \).

It could conclude as follows.

\[
1\text{st Exp} \times 2\text{nd Exp} = \left( r_2^{2^{n-l_0}} X^{r_1} \right) \times \left( r_2^{2^{n-l_1}} X^{E-r_1} \right)
\]

Because \( i_0 = i_1 = 0 \) at the end of exponentiation.

\[
\left( r_2^{2^{n-l_0}} X^{r_1} \right) \times \left( r_2^{2^{n-l_1}} X^{E-r_1} \right) = X^{r_1+E-r_1} = X^E
\]

Table 6.1 Efficiency analysis for proposed Algorithm 6.1
## 6.2 Efficiency Analysis

Compare to Masked MPL and Exponent Splitting, the proposed countermeasure has relative lower speed and higher memory requirement as shown in Table 6.1. However, as mentioned in section 3.3.2 and 3.3.4, existing works have weakness towards attacks proposed in [27 and 29]. While proposed modified MPL with countermeasures is more resistive towards these attacks. More detailed security analysis is included in next section.

### 6.3 Security Analysis

Proposed modified MPL with countermeasures can prevent the launching of a series attacks. The security analyses toward these attacks are listed in this section.

- **6.3.1 Against Simple Side Channel Attacks**

  Simple Side Channel Attack has no effect on MPL because of its feature of *Highly Regular*. Proposed modified MPL with countermeasures (Algorithm 6.1) does not change this feature. It still always has the same operation regardless to the inputs. And it does not have any dummy operations. Consequently, Simple Power Analysis as well as Safe-error attack has no effect on proposed countermeasure.

- **6.3.2 Against Relative Doubling Attack and Comparative Power Analysis**

  These two attack share the same principle of choose specific inputs with the purpose of generating *Collisions*. And these have been discussed in section 3.1 and 3.4.6.

<table>
<thead>
<tr>
<th>Countermeasures</th>
<th>Multiplication</th>
<th>Squaring</th>
<th>Register Needed</th>
<th>Iteration Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masked MPL</td>
<td>Log N</td>
<td>1.5 Log N</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>Exponent Splitting</td>
<td>2 log N</td>
<td>2 log N</td>
<td>6</td>
<td>2N</td>
</tr>
<tr>
<td>Proposed Algorithm 6.1</td>
<td>2 log N</td>
<td>2 log N</td>
<td>7</td>
<td>2N</td>
</tr>
</tbody>
</table>
However, in proposed Algorithm 6.1, such type of attack will have little use. First of all, the reference power traces is generated at second attempt. And different attempts will have different secret exponent because exponent is randomly split into two parts and each part is computed separately. Second, the sequence procedure is randomized. The target operation is more likely shift to other slots and therefore the comparison between target and reference are meaningless. Third, at each step, all the intermediate values are masked by multiplying with random mask $r_2$ or $r_2^{-1}$, even the same operation is coincidently generated in target and reference, the corresponding power traces will look differently. And as long as the mask $r_2$ remains secret, all the in-between computation appears like random squaring and multiplication. Moreover, all random masks will regenerate at the beginning of new input. For a new round of exponentiations, a different pair of mask and anti-mask will generate correspondingly. As a result, repeating attack can be effective prevented.

Figure 6.1 is an illustration for proposed countermeasure against Relative Doubling Attack. All the intermediate values for the corresponding power traces are shown under the power traces. Two split exponents Secret Bits_0 and Secret Bits_1 are record in the table right above the corresponding power traces which are released during the computation of these secret bits. Notice when Secret Bits_0 is under computation, Secret Bits_1 is not displayed, since only one of them is computed. Random number Seq is also listed. Whenever it equal to zero, Secret Bits_0 is computed. Otherwise, Secret Bits_1 is involved. Relative Doubling Attack checks the adjacent iteration between reference and target power traces to see if there any collision generated. The first comparison would be as the red blocks highlighted in Figure 6.1. The comparison results
must be different since it compares \( MR^{-1} \to M^2 R^{-2} \) and \( M^2 R^{-1} \to M^4 R^{-2} \). However, it can’t conclude that adjacent secret bits are not same. Since in the case of Figure 6.1, two adjacent exponents belong to two different exponentiations and the similarity of these two bits makes no sense.

<table>
<thead>
<tr>
<th>Seq</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>.....</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secret bits_0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>.....</td>
</tr>
<tr>
<td>Secret bits_1</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>.....</td>
</tr>
</tbody>
</table>

**Figure 6.1 Algorithm 6.1 against Relative Doubling Attack**

And the second comparison is highlighted by the green blocks. Two operations are indicated as \( M^2 R^2 \to M^4 R^4 \) and \( M^2 R^{-2} \to M^4 R^{-4} \). Even the unmasked computation is coincidently to be same as \( M^2 \to M^4 \), but since the masks \( R^2 \) and \( R^{-2} \) are different,
they still look differently in power traces. From two above example, it's clear that comparison between two adjacent iterations has no use. The hidden relationship used by Relative Doubling Attack is destroyed in proposed modified MPL with countermeasures.

Comparative power traces shares the similar attacking scheme. Even two operations under comparison have more arbitrary relationship; such relationship is also destroyed in proposed algorithm.

-6.3.3 Against Template attack

Template attack mentioned in section 3.3.4 is still very effective on violent the masking process \( r \times X \). The following analysis shows why even the mask root \( r \) is compromised; the whole Algorithm 6.1 is still not cracked. Assume the mask root \( r \) is revealed to attacker, the next following step would be calculating the masks for all iterations based on the updating pattern \( r^{2^{n-i}} \in \mathbb{G} \). The challenge comes with figuring out iteration number \( i \). Since the iteration process is modified and randomized, the mask update as \( r^{2^{n-i}} \in \mathbb{G} \), and iteration number \( i \) is different in first and second exponentiations. Two exponentiations have their own iteration number and these numbers are highly affected by the random number \( Seq \) which is safely store in register in most of time. In another word, template attack is able to successfully reveal the mask root \( r \). But, with sequence randomized, the mask updating pattern is still secure. And consequently, the algorithm as one is not compromised.

-6.3.4 Against High Order Attack and Combined attacks

High Order Attack mentioned in section 3.3.2 is difficult to launch on propose modified MPL with countermeasures (Algorithm 6.1). Since, even though, the imbalance statistic property in two split exponents still exists, the adversary has difficulty to collect
enough samples to analysis such probability. For example, in order to analysis the percentage for case \( \Pr(r_1 r_1') = \Pr(0,0) \), the adversary have to detect case of \( (r_1, r_1') = (0, 0) \) first and then account the number of such case in order to statistic the percentage. In another word, high order attack requires two exponentiations are accessible in the first place. This requires it combined with other attacks to crack individual exponentiation. As mentioned in section 3.3.1, each individual exponentiation in Exponent Splitting has no addition protection. Any simple SCA is able to crack them. Therefore, high order attacks combined with simple SCA is able to break Exponent Splitting. In contrary to this situation, the split exponentiation in Algorithm 6.1 has additional protection. The intermediate values and computation sequences are masked by random number. In order to launch High Order Attack, other attack has been induced to break both split exponentiations.

Let’s see consider simple power analysis. Since Highly Regular is still apply to each split exponentiation. SPAs have no use at all in this case. Relative Doubling Attack or Comparative Power Analysis has no use either towards split exponentiations either. With computation sequence randomized, Algorithm 6.1's power traces do not follow the same time schedule. Same operation can be shifted to other time slots in different attempts. Analysis in section 5.2 shows the functionality of sequence randomization against these two attacks.

Let’s see consider the case that High Order Attack is combined with Fault attack and Template Attack. Message masks are the first barrier; even though masked MPL is crack-able in masking process using template attack. Attacker still has no idea about the
updating pattern for these masks because of randomized sequence, which is the second barrier.

Since the random number $Seq$ does not involve in any computations, but only used for branching criteria. It’s considered very secure. Though, it favors the attacker that the sequence procedures highly sensitive to fault inducing since two exponentiations are independent during the computation process. The fault induced to any computation makes the related computations faulty but have no effect on the other exponentiation. It distinguishes two exponentiations and eventually makes secrecy of computation sequence nonsense.

Such scenario is still worry free. First of all, if the fault attack is based on bit manipulating, it’s difficult for attackers to find out a non-fault reference. Since all the intermediate values are masked by random number including the faulty results. Adversary cannot distinguish the faulty and normal values.

Even we assume there is a fault attack that is able to instantly notice the fault without reference; such assumption will not harm the security of Algorithm 6.1. Because fault attacks are destructive, the data involved with fault cannot be recovered. In another word, if any fault attack is induced to one exponentiation, it actually mess the data up and attackers have no way to recover it. Although the other exponentiation can be distinguished by then, the characteristic of exponent splitting decides that knowing only one of the two exponents is not adequate to disclose the origin exponent. For High Order Attack, it requires that both split exponents ($r_1 r'_1$) has to be known to launch. Knowing only one of them is not sufficient.
Repeated collecting one of both split exponents \((r_i, r'_i)\) has no use either. Since the number collected can either belongs to \(r_i\) or \(r'_i\). It is impossible to classify the detected samples into two groups. Beside, with Checksum function in the end, many fault attack will be stopped at that point.

In conclusion, the above security analysis shows that the proposed countermeasure is able to help building a more resistive power trace. And such power trace can stop simple SCAs, relative doubling attack, comparative power analysis and template attack. In addition, high order attacks combined with above attacks can also be stopped. Any other attacks share the same philosophy with attacks list above can be prevented too.

6.3 Summary

Even though, the proposed countermeasure has relatively less overall efficiency compare to the previous works, the main contribution for proposed work emphasis on the resistance towards SCAs in algorithm level. As mentioned in previous chapters, existing works of MPL enhancement are proven to be unsafe. However, the proposed modified MPL with countermeasure is able to produce more resistive power traces towards SCAs. It randomizes not only exponent, but also the message and computation procedure in contrary to only one in previous works. Besides, mask and anti-mask for messages are more efficient updated and neutralized compare to Masked MPL. No additional updating and mask removing process are needed. The elimination of individual anti-mask updating process has great advantage. The anti-mask updating process in [28] can be easily located and be abused. In contrary, for proposed countermeasure, updating process
is naturally complete along with exponentiation. Since it can’t be separated from major operations, there are little chances being spotted by adversary.

Table 6.2 Countermeasures versus SCAs.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Coron’s [19] 1999</td>
<td>×</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Shamir’s [22] 1999</td>
<td>×</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Exponent Splitting [23] 2001</td>
<td>√</td>
<td>×</td>
<td></td>
<td>√</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Masked MPL [28] 2006</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proposed Algorithm 6.1</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

In Table 6.2, it summarized the security strength comparisons between the proposed modified MPL with existing works. SCAs are listed on the top row. At each row, the first column is name of the countermeasure. The rest columns represent countermeasures’ resistance toward corresponding SCAs in forms of different symbols. Where check “ √ ” indicates such countermeasure have resistance to this SCA. Cross “ × ” represents this countermeasure is vulnerable to such SCA. If blank is left in this column, it means such SCA is not applicable to this countermeasure. And it’s clearly indicated in
Table 6.1 that existing works all has certain vulnerability. Among which, MPL’s weakness can refer to section 3.1. The limitation of Coron’s work [19] and Shamir’s pattern [22] has been discussed in section 3.4.6. Exponent Splitting [23] has been proven unsecure in section 3.3.2. And Masked MPL’s weakness is summarised in section 3.3.4. In contrary, proposed algorithm is able to stop the listed SCAs as mentioned in detail analysis in section 6.3.
CHAPTER VII

HARDWARE IMPLEMENTATION FOR PROPOSED COUNTERMEASURE

In this chapter, the hardware implementation for proposed modified MPL with countermeasure (algorithm 6.1) is explained. The hardware programming language is chosen as Verilog for its user friendly programming style and nice popularity. And such Verilog implementation has been downloaded and tested on the Side-channel Attack Standard Evaluation Board (SASEBO)-GII [31].

The proposed hardware implementation referenced two IP cores. The credit must give to the owner of these IP cores. The first IP core is an AES implementation belongs to the developer group of (SASEBO) [32]. It also includes the windows based Host PC application and example FPGA programming code. The Second IP core is an exhaustive Verilog solution of RSA implementation using Square-and-multiply (Algorithm 2.1) which can be obtained in [38]. It is copyrighted by AIST and Tohoku University. These two IP cores greatly benefit proposed hardware implementation.

7.1 SASEBO-GII

SASEBO-GII is a newly developed FPGA board by National Institute of Advanced Industrial Science and Technology of Japan (AIST). It is suitable for experiments such as one for security evaluation for a comprehensive cryptographic system combining various elemental technologies or one for a large circuit implemented with a variety of countermeasures. The board carries the latest Xilinx Virtex-5 LX30/LX50 as the target FPGA for implementation evaluation.

The further specification follows:

Two Xilinx FPGAs
– Cryptographic FPGA: XC5VLX30 or XC5VLX50 -1FFG324 (Virtex-5 series)
– Control FPGA: XC3S400A-4FTG256 (Spartan-3A series)
– The on-board oscillator provides the control FPGA with a clock signal of 24MHz. An external clock input is also supported.
– External power source supplies the on-board power regulators and the FPGAs with 5.0 V. The power regulators convert the 5-V input into 3.3 V, 1.8 V, 1.2 V, and 1.0 V for the FPGAs. The core voltage of 1.0 V of the cryptographic FPGA can also be applied directly through the external power connector.
– Shunt resistors are provided to insert on the core VDD and/or ground lines of the cryptographic FPGA for measuring power traces.

![Figure 7.1 Top-level block diagram of SASEBO-GII](image)

As Figure 7.1 indicates, there are two FPGAs cooperate with each other. The control FPGA is mainly responsible for input and output converting, memory access and USB communication with Host PC etc. On the other hand, cryptographic FPGA carries the algorithm logic and mathematical computation. The realization for proposed work is executed in cryptographic FPGA.
Xilinx Download cable is used for downloading designed program into SASEBO-GII. It connects the host pc with a USB cable and has 14pins J-Tag on the other end for programming the on board SPI memory.

All downloaded programming file are generated in Xilinx ISE13.2 environment. The Host PC is supported by Microsoft .Net Framework 3.5 and a FTDI D2XXX driver for USB communications.

7.2 HDL Simulation

HDL simulation is carried out in Xilinx ISE 13.2 Isim® simulation environment. The programming language is Verilog. The simulation takes 18,992,218 clock cycles to compute a 1024 bits long secret key using 1024 bits modular and plaintext. And encrypted message is outputted in the end in terms of 32 bits data string.

Table 7.1 Hardware usage of FPGA implementation of proposed modified MPL

(Algorithm 6.1)

<table>
<thead>
<tr>
<th></th>
<th>Utilized</th>
<th>Available in the system</th>
<th>% of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Registers</td>
<td>4934</td>
<td>19200</td>
<td>25%</td>
</tr>
<tr>
<td>Number of Slice LUTs:</td>
<td>3662</td>
<td>19200</td>
<td>19%</td>
</tr>
<tr>
<td>Number of IOs:</td>
<td>47</td>
<td>220</td>
<td>21%</td>
</tr>
<tr>
<td>Number of BUFG/BUFGCTRLs:</td>
<td>2</td>
<td>32</td>
<td>6%</td>
</tr>
<tr>
<td>Number of DSP48Es:</td>
<td>4</td>
<td>32</td>
<td>12%</td>
</tr>
</tbody>
</table>

7.3 Synthesis Result

The HDL is synthesized for Xilinx XC5VLX30 using Xilinx ISE 13.2. Table 7.2 summarized the hardware resource usage of the processor in FPGA implementation.
According to synthesis report, the processor operates at 78.927MHz. The operation time for 1024 bits data is 240.71ms.

This hardware implementation is tested on SASEBO-GII. Figure 7.4 shows the on board waveform captured by Chipscope® at results outputting.

7.4 Summary

Table 7.3 shows hardware usage of different algorithms adopted in SASEBO-GII. Compare to Square-and-multiply and MPL, the proposed hardware implementation roughly doubles the cost. Existing work of exponentiation algorithms mentioned in this thesis focus more on algorithm level design. And there are very few existing hardware implantations for MPL with countermeasures. Some of them are summarized in table 7.4. Implementations of [37, 39 and 40] are proposed for speed concerns. Among them, modular exponentiations algorithms used in [39 and 40] are MPL without countermeasures. And implementation in [37] uses MPL with exponent blinding which is a weak countermeasure mentioned in section 3.4.2. Proposed implementation is committed to different purpose. It is used to secure the modular exponentiation rather than improve the speed. Therefore these implementations are not comparable.

Hardware implementation in [36] is proposed to secure RSA digital signature scheme using residue number system (RNS). And its area cost is slightly more comparing to proposed hardware implementation.

The goal of proposed hardware implementation on SASEBO-GII is for next stage power trace analysis. This implementation doesn’t emphasis on the speed or execution time. In contrary, time consuming architecture is used to benefit the power trace monitoring. Since there is only one 32 bits multiplier, all operations associate with
multipliers are more predictable in power traces. Future work on power trace analysis can provide more solid proof of proposed Algorithm 6.1 has better resistive feature in algorithm level.

Table 7.2 Hardware usage of different algorithms implemented on SASEBO-GII

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Slice Registers</th>
<th>Number of Slice LUTs</th>
<th>Number of IOs</th>
<th>Number of BUFG/BUFGCTRLs</th>
<th>Number of DSP48Es</th>
<th>Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square and Multiply [38]</td>
<td>1715</td>
<td>2481</td>
<td>47</td>
<td>1</td>
<td>4</td>
<td>7,002,329</td>
</tr>
<tr>
<td>MPL</td>
<td>1725</td>
<td>2481</td>
<td>47</td>
<td>1</td>
<td>4</td>
<td>9,491,606</td>
</tr>
<tr>
<td>Proposed Algorithm 6.1</td>
<td>4934</td>
<td>3662</td>
<td>47</td>
<td>2</td>
<td>4</td>
<td>18,992,218</td>
</tr>
</tbody>
</table>

Table 7.3 Exponentiation circuit performance for 1024 bit

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Slice Registers</th>
<th>Number of Slice LUTs:</th>
<th>Technology</th>
<th>Frequency</th>
<th>Max Ex. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPL without countermeasure [40] 2001</td>
<td>6633</td>
<td>-</td>
<td>xc40250xv</td>
<td>45.66M</td>
<td>11.95ms</td>
</tr>
<tr>
<td>MPL without countermeasure [39] 2007</td>
<td>3937</td>
<td>-</td>
<td>xc4vfx-10sf363</td>
<td>200/400M</td>
<td>1.71ms</td>
</tr>
<tr>
<td>MPL with Exponent Blinding [37] 2008</td>
<td>3899</td>
<td>6931</td>
<td>Xc3s5600e</td>
<td>119M</td>
<td>7.95ms</td>
</tr>
<tr>
<td>RNS SCA Protected Exponentiation Algorithm [36] 2003</td>
<td>4956</td>
<td>16370</td>
<td>Xc2v6000</td>
<td>50M</td>
<td>158ms</td>
</tr>
<tr>
<td>Proposed Implementation of Algorithm 6.1</td>
<td>4934</td>
<td>3662</td>
<td>Xc5vlx30</td>
<td>78.9M</td>
<td>240.71ms</td>
</tr>
</tbody>
</table>
Figure 7.2 Architecture of proposed modified MPL with countermeasures
Figure 7.3 Block diagram of Cryptographic FPGA for realizing Algorithm 6.1
Figure 7.4 On board waveform of data outputting
CHAPTER VIII

CONCLUSION

8.1 A summary of contributions

This thesis contributes the efficiency by proposing two efficient architectures for modular exponentiation respectively using Montgomery powering ladder algorithm and m-ary powering ladder method which is mentioned in Chapter IV. And further enhancements on the security strength of MPL are proposed via two cryptographic computation countermeasures. Firstly, a novel sequence masking technique is proposed in Chapter V for masking computation sequence for two individual modular exponentiations. And then, it is further developed into a modified MPL algorithm with countermeasures to frustrate a series of SCAs in Chapter VI. Compare to existing work, proposed modified MPL with countermeasures is less efficient but has better resistance. In addition, the hardware implementation for proposed modified MPL with countermeasures is illustrated in Chapter VII.

8.2 Conclusion

In conclusion, in this thesis, a modified MPL algorithm (Algorithm 4.1) has been proposed which reduces the number of loops by half. Two efficient hardware architectures (Figures 4.1 and 4.3) have also been presented for the Montgomery ladder algorithm and the modified Montgomery ladder algorithm, respectively. Besides, the proposed novel sequence technique (Algorithm 5.1) is able to be effectively against comparative power analysis and chosen message attacks mentioned in chapter III. In order to solve the problem of Algorithm 5.1’s sensitivity to fault attack, two existing ideas have been adapted and combined with the proposed sequence
technique to form a modified and enhanced MPL with countermeasures (Algorithm 6.1). The comparison to existing work in chapter VI shows proposed Algorithm 6.1 has relatively less overall efficiency but more emphasis on the strong resistance towards SCAs in algorithm level. As shown in Table 6.2, proposed Algorithm 6.1 is able to stop DPAs like doubling attack [20], high order attack [27], relative doubling attack [30], fault Attack [34], template attack[29], Comparative Power Analysis[21].

8.3 Possible future work

The proposed efficient architectures (Figures 4.1 and 4.3) are applied the ME in RSA. In the future, they are expected to be extended for Montgomery powering ladder for elliptic curve scalar multiplication.

In addition, power trace analysis through oscilloscope can be launched on the hardware implementation of proposed modified MPL algorithm (Algorithm 6.1). Since the hardware implementation is realized on SASEBO-GII which is specialized circuit to evaluating SCA resistance. The FPGA power consumption is able to be monitored by probing assigned port. DPAs mentioned in chapter III will launch to SASEBO-GII while proposed Algorithm 6.1 is running. The corresponding power waveforms will be analyzed to verify the theoretical analysis.

Moreover, the hardware implementation in chapter VII is build for waveform monitoring. The optimization of running time and resources usage were not carefully concerned. It can be further improved to reduce running time and system resources usage.
REFERENCES


APPENDICES
SELECTED HDL PROGRAMMING CODES

/**************************************************************************
 *** RSA1024_RAM.v ***
 *** Yiruo He ***
 *** Aug. 16 2012 ***
**************************************************************************/

/* This is a top module for RSA encryption. It will receive 1024 bits key, modular
d and plaintext from
data port Kin, Min, Din respectively. The encrypted message outputs from data port Dout.
It contains two component modules RSA_MultiplicationBlock and RSA_SequenceBlock */

module RSA ( Kin, Min, Din, Dout, Krdy, Mrdy, Drdy, RSTn, EN, CLK, BSY, Kvld, Mvld, Dvld );

input         CLK, RSTn, EN;
input [31:0]    Kin, Min, Din;
input         Kr, Mr, Dr;
output         BSY;
output [31:0]  Dout;
output         Kvld, Mvld, Dvld;

reg [1023:0]    Krg, Krg_1,RND;
reg [2047:0]    Seq;
wire [4:0]      count;
wire [1:0]      InOutMem;
wire [2:0]      state;
wire [31:0]     w_data, d_out, r_data_m, r_data_s, r_data0, r_data1, r_data2, r_data3, d_in;
wire [30:16]    MBCon;
wire [8:0]      MemCon_m, MemCon_s, MemCon0, MemCon1, MemCon2, MemCon3, MemCon_i,
                 MemCon_o;
wire [1:0]      MemSel;
wire [1:0]      DSel;
wire          v, sign,FM;
wire          EnKey;
wire          key_bit, seq_bit;

parameter      INIT   = 3'h1;
parameter      IDLE   = 3'h2;
parameter      KEY_GET = 3'h3;
parameter      MOD_GET = 3'h4;
parameter      DATA_GET = 3'h5;
parameter      DATA_OUT = 3'h6;
parameter      ENCRYPT = 3'h7;

always @(posedge CLK) begin
    if (RSTn == 1'b0) begin
        Krg <= 1024'h0;
        RND <= 1024'h0;
        RND[31:0] <= 32'h00000000;
    end
end

Krg_1 <= 1024'h0;
Seq <=2048'h0;
end
else if (state == KEY_GET) begin
    Krg <= {(Kin-RND[31:0]), Krg[1023:32]};
    Krg_1 <= {RND[31:0], Krg_1[1023:32]};
    RND<= {RND[31:0],RND[1023:32]};
end
else if (EnKey == 1'b1) begin
    Seq <= (Seq[2046:0],1'b0);
    if (MemSel[1]==0'b0)
        Krg <= {Krg[1022:0], Krg[1023]};
    else
        Krg_1 <= {Krg_1[1022:0], Krg_1[1023]};
end
end

assign Dout = r_data_m;
assign seq_bit = Seq[2047];

function mux2_1_1;
    input  a, b;
    input Sel;
    case (Sel)
        1'b0: mux2_1_1 = a;
        1'b1: mux2_1_1 = b;
    endcase
endfunction // mux2_1_1

function [31:0] mux2_1_32;
    input [31:0] a, b;
    input Sel;
    case (Sel)
        1'b0: mux2_1_32 = a;
        1'b1: mux2_1_32 = b;
    endcase
endfunction // mux2_1_32

function [8:0] mux3_1_9;
    input [8:0] a, b, c, d;
    input [2:0] Sel;
    case (Sel)
        3'b000: mux3_1_9 = a;
        3'b010: mux3_1_9 = c;
        3'b100: mux3_1_9 = b;
        3'b110: mux3_1_9 = c;
        3'b001: mux3_1_9 = d;
        3'b101: mux3_1_9 = d;
        default: mux3_1_9 = a;
    endcase
endfunction // mux3_1_9
function [8:0] mux6_1_9;
    input [8:0] a, b, c, d, e;
    input [4:0] Sel;
    case (Sel)
        5'b00000: mux6_1_9 = a;
        5'b00010: mux6_1_9 = c;
        5'b00100: mux6_1_9 = b;
        5'b00110: mux6_1_9 = d;
        5'b01000: mux6_1_9 = 9'b000000000;
        5'b01100: mux6_1_9 = 9'b000000000;
        5'b01010: mux6_1_9 = c;
        5'b01110: mux6_1_9 = c;
        5'b01001: mux6_1_9 = d;
        5'b01101: mux6_1_9 = d;
        5'b10000: mux6_1_9 = e;
        5'b10010: mux6_1_9 = e;
        5'b10100: mux6_1_9 = e;
        5'b10110: mux6_1_9 = e;
        5'b10101: mux6_1_9 = e;
        5'b11000: mux6_1_9 = e;
        5'b11100: mux6_1_9 = e;
        5'b11010: mux6_1_9 = c;
        5'b11110: mux6_1_9 = c;
        5'b11001: mux6_1_9 = d;
        5'b11101: mux6_1_9 = c;
        default: mux6_1_9 = a;
    endcase
endfunction // mux6_1_9

function [31:0] mux3_1_32;
    input [31:0] a, b, c;
    input [1:0] Sel;
    case (Sel)
        2'b00: mux3_1_32 = a;
        2'b01: mux3_1_32 = b;
        2'b10: mux3_1_32 = c;
        default: mux3_1_32 = a;
    endcase
endfunction // mux3_1_32

function [31:0] mux4_1_32;
    input [31:0] a, b, c, d, e;
    input [2:0] Sel;
    case (Sel)
        3'b000: mux4_1_32 = a;
        3'b001: mux4_1_32 = b;
3'b010: mux4_1_32 = c;
3'b011: mux4_1_32 = d;
3'b100: mux4_1_32 = e;
3'b101: mux4_1_32 = e;
3'b110: mux4_1_32 = e;
3'b111: mux4_1_32 = e;
default: mux4_1_32 = a;
endcase
endfunction // mux4_1_32
assign d_in = (state == MOD_GET) ? Min : Din;
assign w_data = mux3_1_32(d_out, r_data_s, d_in, DSel);
assign v = r_data_s[0];

RSA_MultiplicationBlock MULT_BLK (CLK, RSTn, MBCon, r_data_m, r_data_s, d_out, sign);

RSA_SequencerBlock SEQ_BLK (CLK, RSTn, EN, Krdy, Mrdy, Drdy, key_bit, seq_bit, sign, MBCon, MemCon_m, MemCon_s, EnKey, MemSel, FM, DSel, count, InOutMem, state, BSY, Kvld, Mvld, Dvld);

assign MemCon_i = (state == MOD_GET) ? {2'b11, 2'b10, count}:{2'b11, 2'b01, count};
assign MemCon_o = {2'b01, 2'b00, count};

assign MemCon0 = mux6_1_9(MemCon_m, MemCon_s, MemCon_i, MemCon_o, 9'b000000000, {FM, seq_bit, MemSel[0], InOutMem});

assign MemCon1 = mux6_1_9(MemCon_s, MemCon_m, MemCon_i, MemCon_o, MemCon_s, {FM, seq_bit, MemSel[0], InOutMem});

assign MemCon2 = mux6_1_9(MemCon_m, MemCon_s, MemCon_i, MemCon_o, MemCon_m, {FM, MemSel[1], MemSel[0], InOutMem});

assign MemCon3 = mux6_1_9(MemCon_s, MemCon_m, MemCon_i, MemCon_o, 9'b000000000, {FM, MemSel[1], MemSel[0], InOutMem});

assign r_data_m = mux4_1_32(r_data0, r_data1, r_data2, r_data3, {FM, MemSel});
assign r_data_s = mux4_1_32(r_data1, r_data0, r_data3, r_data2, r_data1, {FM, MemSel});

assign key_bit = mux2_1_1(Krg[1023], Krg_1[1023], MemSel[1]);

// memory simulation model
RSA_Memory MEM0 (r_data0, CLK, ~MemCon0[7], ~MemCon0[8], MemCon0[6:0], w_data);
RSA_Memory MEM1 (r_data1, CLK, ~MemCon1[7], ~MemCon1[8], MemCon1[6:0], w_data);
RSA_Memory MEM2 (r_data2, CLK, ~MemCon2[7], ~MemCon2[8], MemCon2[6:0], w_data);
RSA_Memory MEM3 (r_data3, CLK, ~MemCon3[7], ~MemCon3[8], MemCon3[6:0], w_data);
endmodule // top

/* RSA_ModExpSequencer is a sequencer module for modular exponentiation X^E mod N. It is a component of RSA_SequenceBlock. It’s the structure for realizing modular exponentiation algorithm. */
module RSA_ModExpSequencer (CLK, RSTn, Rst, Msb, Exp, Cy_mr, Fin, pc);
  input CLK, RSTn, Rst;
  input Msb, Exp;
  //input PreComp;
  input Cy_mr;
  input Fin;
  output [14:0] pc;

  reg [14:0] pc;
  reg IDLE;
  always @(posedge CLK) begin
    if (RSTn == 1'b0) begin
      pc <= 15'b000000000000001;
      IDLE <= 1'b0;
    end
    else if (Rst == 1'b1) begin
      pc <= 15'b000000000000001;
      IDLE <= 1'b0;
    end
    else if (pc[0])
      if (Fin == 1) pc <= {pc[13:0],1'b0}; // 0 to 1
      else pc <= pc; // 0 to 0
      if (Fin == 1) pc <= {pc[13:0],1'b0}; // 1 to 2
      else pc <= pc; // 1 to 1
    else if (pc[4])
      if (Fin == 1) pc < 15'b0000000000100000; // 1 to 2
      else pc <= pc;
      else if (IDLE)
      if (Exp == 0) begin
        pc <= 15'b001000000000000; // 5 to 12
        IDLE <= 1'b0;
      end
      else begin
        pc <= 15'b000000001000000; // 5 to 6
        IDLE <= 1'b0;
      end
    else if (pc[5])
      if (Msb == 1) begin
        pc <= 15'b000000000000000;
        IDLE <= 1'b1;
      end
      else pc <= 15'b000000000000000; // 5 to 7
    else if (pc[6])
      if (Fin == 1) pc <= 15'b0001000000000000; // 6 to 11
      else pc <= pc; // 6 to 6
      else if (pc[7]) pc <= {pc[13:0],1'b0}; // 7 to 8
    else if (pc[8])
      if (Cy_mr == 1) pc <= {pc[13:0],1'b0}; // 8 to 9
      else pc <= 15'b0000000000100000; // 8 to 5
  end
endmodule
else if (pc[11])
    if (Fin == 1) pc <= 15'b000000100000000;  // 11 to 8
else
    pc <= pc;  // 11 to 11
else if (pc[12])
    if (Fin == 1) pc <= {pc[13:0],1'b0};  // 12 to 13
else
    pc <= pc;  // 12 to 12
else if (pc[13])
    if (Fin == 1) pc <= 15'b000000100000000;  // 13 to 8
else
    pc <= pc;  // 13 to 13
else if (pc[10])
    if (Fin == 1) pc <= 15'b100000000000000;
else
    pc <= pc;  // to next state
end
endmodule // ModExpSequencer

/* This module is a sequencer module for montgomery multiplication X * Y * R^(-1) mod N. It generates the control signal to fulfill specific operation commanded by ModExpSequencer. The control signal is a 31 bits data which are used for coordinate memory module, loop control module and Multiplication Block*/
module RSA_MontMultSequencer (CLK, RSTn, Start, i, Cy_m, Sel, exp,Con, Hlt,Con_Yj);
input CLK, RSTn, Start;
input [9:0] i;
input Cy_m, Sel,exp;
output [30:0] Con;
output Hlt, Con_Yj;
reg [27:0] pc;
wire zero;
assign zero = ~{(i ^ 10'b0000000000000000)};
always @(posedge CLK) begin
  if (RSTn == 1'b0)
    pc <= 28'b0000000000000000000000000000;
  else if (Start == 1'b1)
    pc <= 28'b0000000000000000000000000001;
  else if (pc[1])
    pc <= 28'b0000000000000000000000001000; // 1 to 3
  else if (pc[5])
    if (zero == 1'b1)
      pc <= {pc[26:0],1'b0}; // 5 to 6
    else
      pc <= 28'b0000000000000000000000000010; // 5 to 8
  else if (pc[7])
    if (Cy_m == 1'b1)
      pc <= 28'b0000000000000000000000000000; // 7 to 12
    else
      pc <= 28'b0000000000000000000000000000; // 7 to 16
  else if (pc[9])
    pc <= 28'b0000000000000000000000000000; // 9 to 10
  else if (pc[11])
    if (Cy_m == 1'b1)
      pc <= {pc[26:0],1'b0}; // 11 to 12
    else
      pc <= 28'b0000000000000000000000000000; // 11 to 16
  else if (pc[17])
    if (Cy_m == 1'b1)
      pc <= {pc[26:0],1'b0}; // 17 to 18
    else
      pc <= 28'b0000000000000000000000000000; // 17 to 20
  else if (pc[21])
    if (Cy_m == 1'b1)
      pc <= {pc[26:0],1'b0}; // 21 to 22
    else
      pc <= 28'b0000000000000000000000000000; // 21 to 24
  else if (pc[23])
    pc <= 28'b0000000000000000000000000000; // 23 to 25
  else if (pc[25])
    if (Cy_m == 1'b1)
      pc <= {pc[26:0],1'b0}; // 25 to 26
    else
      pc <= 28'b0000010000000000000000000000; // 25 to 29
end

// Sel = 1 squaring     Sel = 0 multiplication

function [30:0] decoder;
  input [27:0] pc;
  input Sel, zero, exp;
case({exp, Sel, pc})

// Pc[13] = Y * Z mod N, exp=0, sel=0

30'b000000000000000000000000000001: decoder = {15'b0101000000000000, 4'b0100, 6'b010100, 6'b001000}; // 0
30'b0000000000000000000000000000010: decoder = {15'b0000101000000000, 4'b0111, 6'b000010, 6'b000011}; // 1
30'b00000000000000000000000000000100: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 2
30'b0000000000000000000000000000001000: if (zero == 1'b1) decoder =
{15'b0101100000000000, 4'b0100, 6'b000000, 6'b011000}; // 3
else decoder = {15'b0110100000000000, 4'b0100, 6'b000000, 6'b011000}; // 3
30'b0000000000000000000000000000000000: decoder = {15'b0001101001000000, 4'b0100, 6'b000000, 6'b000000}; // 4
30'b0000000000000000000000000000000000: decoder = {15'b0000100000000000, 4'b0111, 6'b000000, 6'b000101}; // 5
30'b0000000000000000000000000000000000: decoder = {15'b0000001000000000, 4'b0101, 6'b000000, 6'b000101}; // 6
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 7
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 8
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 9
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 10
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 11
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 12
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 13
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 14
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 15
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 16
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 17
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 18
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 19
30'b0000000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000101}; // 20
30'b0000000000000000000000000000000000: decoder = {15'b0101000000000000, 4'b0100, 6'b000000, 6'b000101}; // 21
30'b00000000100000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b010000}; // 22
30'b00000001000000000000000000000000: decoder = {15'b001010101000001, 4'b0000, 6'b000000, 6'b000000}; // 23
30'b00000010000000000000000000000000: decoder = {15'b001010001000000, 4'b0100, 6'b000000, 6'b111000}; // 24
30'b00000100000000000000000000000000: decoder = {15'b010100000000000, 4'b0100, 6'b000000, 6'b000000}; // 25
30'b00010000000000000000000000000000: decoder = {15'b000000000000000, 4'b0001, 6'b010100, 6'b000101}; // 26
30'b00010000000000000000000000000000: decoder = {15'b000000000000000, 4'b0000, 6'b000000, 6'b000000}; // 27

#pragma P(12) P(2)/ Z=Z^Z mod N, exp=0, sel=1
30'b01000000000000000000000000000001: decoder = {15'b0101000000000000, 4'b0100, 6'b101000, 6'b000000, 6'b011000}; // 3
30'b01000000000000000000000000000000: decoder = {15'b0000101000010000, 4'b0111, 6'b000000, 6'b000101}; // 1
30'b01000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000000}; // 2
30'b01000000000000000000000000000000: if (zero == 1'b1) decoder =
{15'b0101100000000000, 4'b0100, 6'b000000, 6'b011000}; // 3
else decoder = {15'b0110100000000000, 4'b0100, 6'b000000, 6'b011000}; // 3
30'b01000000000000000000000000000000: decoder = {15'b0001101001000000, 4'b0000, 6'b000000, 6'b000000}; // 4
30'b01000000000000000000000000000000: decoder = {15'b0000100000000000, 4'b0111, 6'b001000, 6'b100100}; // 5
30'b01000000000000000000000000000000: decoder = {15'b0001001000000000, 4'b0000, 6'b000000, 6'b000000}; // 6
30'b01000000000000000000000000000000: decoder = {15'b0000001000000000, 4'b0001, 6'b000000, 6'b000110}; // 7
30'b01000000000000000000000000000000: decoder = {15'b0100100011000000, 4'b0101, 6'b000000, 6'b010000}; // 13
30'b01000000000000000000000000000000: decoder = {15'b0110001000000000, 4'b0001, 6'b000000, 6'b000110}; // 14
30'b01000000000000000000000000000000: decoder = {15'b1010000000000000, 4'b0111, 6'b001000, 6'b101000}; // 11
30'b01000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 12
30'b01000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000000}; // 8
30'b01000000000000000000000000000000: decoder = {15'b0010001000000000, 4'b0000, 6'b000000, 6'b000110}; // 9
30'b01000000000000000000000000000000: decoder = {15'b0000000000000000, 4'b0000, 6'b000000, 6'b000000}; // 10
30'b01000000000000000000000000000000: decoder = {15'b0100000000000000, 4'b0100, 6'b010100, 6'b000000, 6'b010000}; // 11
30'b01000000000000000000000000000000: decoder = {15'b0100000000000000, 4'b0100, 6'b010100, 6'b000000, 6'b010000}; // 12
30'b01000000000000000000000000000000: decoder = {15'b0101000000000000, 4'b0100, 6'b010100, 6'b000000, 6'b010000}; // 13
30'b01000000000000000000000000000000: decoder = {15'b0110000000000000, 4'b0100, 6'b010100, 6'b000000, 6'b010000}; // 14
30'b01000000000000000000000000000000: decoder = {15'b1011000000000000, 4'b0100, 6'b010100, 6'b000000, 6'b011000}; // 15

80
<table>
<thead>
<tr>
<th>Address</th>
<th>Decoder</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>30'b010000000000000000000000000000000: decoder = {15'b0000001000000000, 4'b0001, 6'b000000, 6'b000000};</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000001: decoder = {15'b1010000000000000, 4'b0111, 6'b100110, 6'b001101};</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000002: decoder = {15'b0000000000000000, 4'b0101, 6'b000100, 6'b110101};</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000003: decoder = {15'b0010101010000000, 4'b0000, 6'b000000, 6'b000000};</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000004: decoder = {15'b0010100010000000, 4'b0100, 6'b000000, 6'b111000};</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000005: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000110};</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000006: decoder = {15'b0000000000000000, 4'b0000, 6'b000000, 6'b000000};</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000007: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000101};</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000008: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b001000};</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000009: decoder = {15'b0010101010000010, 4'b0000, 6'b000000, 6'b000000};</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000010: decoder = {15'b0000000000000000, 4'b0101, 6'b000000, 6'b000000};</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>30'b010000000000000000000000000000011: decoder = {15'b0000000000000000, 4'b0000, 6'b000000, 6'b000000};</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

// Pc[11] Z = Z*Y modN , exp=1, sel=0

30'b01000000000000000000000000000001: decoder = {15'b0101000000000000, 4'b0100, 6'b010100, 6'b001000}; if (zero == 1'b1) decoder = {15'b0101100000000000, 4'b0100, 6'b000000, 6'b011000}; else decoder = {15'b0110100000000000, 4'b0100, 6'b000000, 6'b000000}; // 3
30'b100000000000000000000000100000000000: decoder = {15'b0100000000000000, 4'b0111, 6'b001000, 6'b001000}; // 11
30'b100000000000000000000000100000000000: decoder = {15'b0000000100000000, 4'b0b101, 6'b000000, 6'b000000}; // 12
30'b100000000000000000000000100000000000: decoder = {15'b0100100010000000, 4'b0101, 6'b000000, 6'b000000}; // 13
30'b100000000000000000000000100000000000: decoder = {15'b0110000000000000, 4'b0001, 6'b010000, 6'b000000}; // 14
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 15
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 16
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 17
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 18
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 19
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 20
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 21
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 22
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 23
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 24
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 25
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 26
30'b100000000000000000000000100000000000: decoder = {15'b0000000000000000, 4'b0001, 6'b000000, 6'b000000}; // 27

// Pc[6], Y=Y^Y mod N, exp=1, sel=1
30'b110000000000000000000000000001: decoder = {15'b0101000000000000, 4'b0111, 6'b010100, 6'b001000}; // 0 m
30'b110000000000000000000000000001: decoder = {15'b0000100000000000, 4'b1111, 6'b000000, 6'b011000}; // 1
30'b110000000000000000000000000001: decoder = {15'b0000100000000000, 4'b0000, 6'b000000, 6'b000000}; // 2
30'b110000000000000000000000000001: decoder = {15'b0000100000000000, 4'b0000, 6'b000000, 6'b000000}; // 3
else decoder = {15'b0101101000000000, 4'b0100, 6'b000000, 6'b011000}; // 5
30'b110000000000000000000100000000: decoder = {15'b000100100000000, 4'b0000, 6'b000000, 6'b000000}; // 6
30'b110000000000000000000100000000: decoder = {15'b100000000000000, 4'b0111, 6'b000100, 6'b100101}; // 7 m s
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0101, 6'b000000, 6'b010011}; // 8 m
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 9
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 10
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 11
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 12
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 13
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 14
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 15
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 16
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 17
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 18
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 19
30'b110000000000000000000100000000: decoder = {15'b000000000000000, 4'b0001, 6'b000000, 6'b000110}; // 20 m
30'b110000000000000000000100000000: decoder = {15'b100000000000000, 4'b0111, 6'b000100, 6'b100111}; // 21
30'b110000000000000000000100000000: decoder = {15'b100000000000000, 4'b0111, 6'b000100, 6'b100111}; // 22
30'b110000000000000000000100000000: decoder = {15'b100000000000000, 4'b0111, 6'b000100, 6'b100111}; // 23
30'b110000000000000000000100000000: decoder = {15'b100000000000000, 4'b0111, 6'b000100, 6'b100111}; // 24
30'b110000000000000000000100000000: decoder = {15'b100000000000000, 4'b0111, 6'b000100, 6'b100111}; // 25
30'b110000000000000000000100000000: decoder = {15'b100000000000000, 4'b0111, 6'b000100, 6'b100111}; // 26
30'b110000000000000000000100000000: decoder = {15'b100000000000000, 4'b0111, 6'b000100, 6'b100111}; // 27

default: decoder = 31'b0000000000000000000000000000000000;
endcase
endfunction

assign Con = decoder(pc, Sel, zero, exp );
assign Hlt = pc[27];
assign Con_Yj = exp ^~ Sel;

endmodule  // MontMult_Sequencer
VITA AUCTORIS

Yiruo He was born in 1987 in P.R.China. He received his Bachelor’s Degree from Faculty of Engineering and Applied Science in University of Regina 2010. He is currently a candidate for the Master of Applied Science Degree in the Department of Electrical and Computer Engineering at University of Windsor and hopes to graduate in Winter 2012.