2017

Private and Public-Key Side-Channel Threats Against Hardware Accelerated Cryptosystems

Dylan Roderick Lalonde
University of Windsor

Private and Public-Key Side-Channel Threats Against Hardware Accelerated Cryptosystems

by

Dylan Roderick Lalonde

A Thesis
Submitted to the Faculty of Graduate Studies through the
Department of 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
2017

Private and Public-Key Side-Channel Threats Against Hardware Accelerated Cryptosystems

by

Dylan R. Lalonde

APPROVED BY:

A. Jaekel, External Reader
School of Computer Science

K. Tepe, Departmental Reader
Electrical and Computer Engineering

H. Wu, Co-Advisor
Electrical and Computer Engineering

M. Mirhassani, Advisor
Electrical and Computer Engineering

April 19, 2017

Declaration of Originality

I hereby certify that I am sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any propriety 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 institution.

Abstract

Modern side-channel attacks (SCA) have the ability to reveal sensitive data from non-protected hardware implementations of cryptographic accelerators whether they be private or public-key systems. These protocols include but are not limited to symmetric, private-key encryption using AES-128, 192, 256, or public-key cryptosystems using elliptic curve cryptography (ECC). Traditionally, scalar point (SP) operations are compelled to be high-speed at any cost to reduce point multiplication latency. The majority of high-speed architectures of contemporary elliptic curve protocols rely on non-secure SP algorithms.

This thesis delivers a novel design, analysis, and successful results from a custom differential power analysis attack on AES-128. The resulting SCA can break any 16-byte master key the sophisticated cipher uses and it's direct applications towards public-key cryptosystems will become clear. Further, the architecture of a SCA resistant scalar point algorithm accompanied by an implementation of an optimized serial multiplier will be constructed.

The optimized hardware design of the multiplier is highly modular and can use either NIST approved 233 & 283-bit Kobliz curves utilizing a polynomial basis. The proposed architecture will be implemented on Kintex-7 FPGA to later be integrated with the ARM Cortex-A9 processor on the Zynq-7000 AP SoC (XC7Z045) for seamless data transfer and analysis of the vulnerabilities SCAs can exploit.

In loving memory of Brianne & Dad

Acknowledgments

I wish to express my most sincere gratitude my advisor Dr. Mitra Mirhassani for her compassion and motivational spirits that truly inspired me to complete my thesis. Her knowledge, time, and support made this work a great success. Mitra's work ethics and dedication in our joint research over the last 2 years shaped the engineer I am today.

A genuine thank you goes to my co-advisor Dr. Huepang Wu for his great knowledge and critical feedback for the mathematics of this project. I would also like to give a special thanks to Dr. Roberto Muscedere for his judgement and insight on the hardware design within the project.

I'd like to also thank my committee members Dr. Arunita Jaekel and Dr. Kemel Tepe for their feedback and advice on my work.

In addition to my loving family, I would also like to thank my good friends Philip Korta and George Kyrtsakas along with my colleagues in the ECE department at the University of Windsor for their continual support, advice, and great times during my entire post-secondary education.

Finally, to my mother, I am perpetually grateful. With your unconditional love and everlasting life lessons, you taught me that knowledge is surely a beautiful thing.

Table of Contents

List of Tables

List of Figures

Nomenclature

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
$ASIC$	Application Specific Integrated Circuit
CPA	Correlation Power Analysis
CPU	Central Processing Unit
DoD	Department of Defense
DPA	Differential Power Analysis
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
$ECDH$	Elliptic Curve Diffie-Hellman
$ECDLP$	Elliptic Curve Discrete Logarithm Problem
$ECDSA$	Elliptic Curve Digital Signature Algorithm
EEA	Extended Euclidean Algorithm
$EEPROM$	Electrically Erasable Programmable Read-only Memory
EM	Electromagnetism
$FIFO$	First in First out
$FPGA$	Field Programmable Gate Array
FSM	Finite State Machine

GCD	Greatest Common Divisor
GF	Galois Field
HDL	Hardware Descriptive Language
$HTTPS$	Hypertext Transfer Protocol Secure
IC	Integrated Circuit
ISR	Inverse Shift Row
$JTAG$	Joint Test Action Group
$L2R$	Left-to-Right
LD	López-Dehab
LUT	Look-up Table
MCC	Micro-programmable Controller
NAF	Non-Adjacent Form
$NIST$	National Institute of Standards and Technology
RSA	R. Rivest, A. Shamir, L. Adleman Cryptosystem
SCA	Side-Channel Attack
$SISO$	Serial in Serial out
SoC	System-on-Chip
SPA	Simple Power Analysis
SPM	Scalar Point Multiplication
USB	Universal Serial Bus
$V2X$	Vehicle-to-Everything
XOR	Exclusive-OR
ZVP	Zero-Value Point

Chapter 1

Introduction

Due to the ever-rising amount of private information being transmitted from one source to another over any communication network, maintaining security places the tremendous burden on internal processing capabilities. Unsatisfying performance from software results in the reign of hardware accelerators in applied cryptography. This speed comes at a large risk of modern attacks against hardware to reveal delicate intelligence.

1.1 Motivation

Currently, this technological decade is proving the pace of digital accessibility to be swift and reliable at any cost. People push the indefinitely increasing traffic of the internet with a wide range of activities. These can be businesses needing real-time updates, banks' secure transactions, and simple password protected accounts used for social media by consumers. In the late 90s, people had minimal access to internet by an expensive personal computer but today in 2017, most people who are connected to the internet out of the current 46% of the worlds population [48], have more than one device connected to the web. Also, within the last few years the automotive industry is starting to penetrate the capabilities of the internet through vehicle-to-everything (V2X) technologies with telematics systems. All of these uses of the connected world need to have a reliable secure end-to-end connection when navigating important information.

1.1.1 Transition of Software to Hardware Cryptography

Traditionally computers would run dedicated software algorithms to authenticate, maintain confidentiality, and or hold integrity of data. Due to the pressure of throughput constraints placed on the computers central processing unit (CPU), physical capabilities of the CPU would further halt speed requests. To obtain these desirable yet forced objectives, the large bit size cryptographic algorithms needed to be implemented in hardware to reach benchmarks that could never be obtained by software. Presently, the most efficient security measures will compute the encryption/decryption or processing in hardware while dedicating all the input output data transmissions and analysis in software which are the modern integrated platforms engineers see today. Hardware computes the public-key cryptosystems with ease enabling users to provide private key exchanges and establishment, authentication, and more importantly, preserving their privacy.

As public-key cryptography offers many benefits, the algorithms used till 1985 were inefficient in hardware. The new concept of Elliptic Curve Cryptography (ECC) offered the same level of symmetric security [34], while maintaining smaller key sizes, memory usage, and power consumption. Naturally, the industry standard then directed its interest to ECC for most large high-speed, high security measures.

Hardware accelerators are in high demand due to the custom, high-speed throughput they can provide. As these sophisticated circuits have an unreachable performance merit compared to software, they possess a characteristic that fingerprints the adept algorithms. Fraudulent acts on these circuits can result in an extensive amount of valuables compromised.

1.1.2 Side-Channel Attacks

Side-Channel Attacks (SCAs) are attacks that gain delicate information acquired from hardware implementations of cryptosystems. This important data leaked is from any side-channel of the circuit as it encrypts/decrypts plaintext-ciphertext or as the system alters keys states [17]. The results of a successful SCA can reveal the

architecture of the integrated circuit (IC), intermediate keys within cryptosystems, and more frightening, can compromise the master key to recover all sensitive input data. Opposed to traditional brute force, SCAs exploit the hardware's nature through timing, fault injection, power, and or electromagnetism (EM) radiation analysis to acquire secret information in merely a fraction of the time.

To protect an IC properly against SCAs there is a broad background required from different domains of embedded security. The central knowledge required includes areas from hardware design for feasibility, functionality, and constraints, a cryptographic algorithm aptitude, and the ability to perform a successful SCA.

1.2 Objective

The main objective of this thesis is to create an entire platform to develop and test side-channel attacks against a wide range of cryptosystems available in hopes to better protect the hardware at the highest level of operations within or outside of the scale of the algorithms.

Specifically, the other objective of this thesis is to create a complete base architecture of a secure scalar point multiplication (SPM) to open the development of an integrated hardware accelerator to be applied in ECC protocols for SCA investigations. The auxiliary support of a comprehensive SCA is also needed to accurately design the custom hardware.

1.2.1 Solution

This thesis will include an in-depth explanation with experimental results, of a successful side-channel attack to show the susceptibilities of cryptographic hardware from multiple aspects. The weaknesses will be discovered to transfer the applied knowledge to the architectural design on a public-key system.

The thesis will additionally include an examination of the entire architecture of the targeted SPM on a field-programmable gate array (FPGA). This will also

include an implementation and analyses of two types of multipliers to be utilized in the design.

1.3 Organization of Thesis

The progression of this thesis will focus on the process of implementing the algorithms for a novel SPM architecture with an insight and practise of modern SCAs for maximum security. The rest of the thesis is as follows.

Chapter 2 is the primitive mathematics required in elliptic curve (EC) operations that encompass the base for ECC. The algebraic basics of group laws and finite fields are be addressed in this chapter. Provable security will be discussed in respect to choosing a correct finite field and parameters. A discussion of the problem that makes ECC strong will be explained along with the fundamental operations of the point operations. A brief introduction of the statistics needed for a SCA will also be explained.

Chapter 3 discusses, investigates, and performs a pertinent side-channel attacks on an industrial practised cipher. Initially the chapter will brief the most opportune SCAs to leverage the targeted hardware auspiciously. The main attacks include timing, safe-error, and differential power attacks. Lastly, the chapter will explain in great depth the exact process to break AES-128 providing a simple method to break power dependant states within a cryptosystem.

Chapter 4 provides a literature survey of the most opportune designs to fight SCAs. It outlines the proposed secure high-level architecture and why it will be secure against the previously mentioned side-channel attacks. This level of the design is most susceptible to SCAs as it deciphers how the master key manipulates the base point of the EC during the SPM to produce the product of a public-private key system. Joye's SP algorithm is designed in hardware and is broken into the two optimized point operations. The datapath of the point doubling and addition is displayed. Lastly, the newly high-level inversion algorithm is selected

and explained to translate to a hardware design.

Chapter 5 offers the low-level implementation of the proposed design. The finite field multiplier is the most important design to be made as it needs to be optimized to the correct application of the overall hardware accelerator. This chapter discusses two multipliers with a hardware throughput solution in detail and gives an analysis of the cost, speed, and feasibly through simulations and synthesis. Finally, an overview of the complete scalar point multiplication algorithm with a hierarchy of operations that build this design.

Chapter 6 covers the overall contributions of this work as well as the future work needed to progress the full development of the secure hardware implementation. The future works include the remainder of the designed hardware in Verilog hardware descriptive language (HDL) to ultimately be attacked to diagnose threats, an implementation of a custom cryptographic library for hardware functionality testing, and vulnerabilities solutions towards symmetrical key hardware ciphers.

Chapter 2

Mathematical Preliminaries

ECC naturally revolves around number theory, group laws, and finite fields arithmetic. Accompanied by the elliptic curve discrete logarithm problem (ECDLP) over a NIST approved, efficient elliptic curve is the formulae for a tenacious mathematical backbone when designing custom hardware.

Sections 2.1-2.3 are in reference to the books [39,40]. These textbooks provide a descriptive yet concise way to understand the ECC algebra fundamentals with ease. The last section will brief the small amount of formulae to grasp the numerical concept of a particular SCA.

2.1 Number Theory

Given two integers x, y, and a positive integer n:

Definition 2.1.1: Congruence

x is congruent to $y \, mod \, n$ if the difference of $x - y$ is integrally divisible by n:

$$x \equiv y \, mod \, n$$

Property: x is congruent to y *if and only if* $y \, mod \, n = x \, mod \, n$.

Definition 2.1.2: Multiplicative Group

The set of elements x of Z_n relatively prime with n, is the *multiplicative group* $\mathbb{Z}_n^*$:

$$\mathbb{Z}_n^* = \{x \in Z_n \mid gcd(x, n) = 1\}, \text{ where } Z_n = \{0, 1, 2, ..., n-1\}$$

Property: The Euler totient function $\Phi(n)$ is the *number of elements* in $\mathbb{Z}_n^*$. Also, if $\mathbb{Z}_n^*$ has a generator, then $\mathbb{Z}_n^*$ is said to be *cyclic*.

Definition 2.1.3: Multiplicative Inverse

In a multiplicative group where the operation is a product, if $xy \, mod \, n = 1$, then y is the the *multiplicative inverse* of x:

$$y = x^{-1} \, mod \, n$$

Property: x has a multiplicative inverse *if and only if* $gcd(x, n) = 1$. If inverse exist, it is unique.

Definition 2.1.4: Order of an Element

The *order* of element $x \in \mathbb{Z}_n^*$ is the least positive integer r such that:

$$x^r = 1 \, mod \, n$$

Property: If the order of x is equal to the number $\Phi(n)$ of elements in $\mathbb{Z}_n^*$, then x is said to be a generator or *primitive element* of $\mathbb{Z}_n^*$.

2.2 Algebra

The following definitions are shown below defined over set $\mathbb{G}$.

2.2.1 Group Law

Using the binary operator $*$, the group is $\mathbb{G}^*$:

Definition 2.2.1.1: Associativity

$$x * (y * z) = (x * y) * z, \forall\, x, y, z \in \mathbb{G}$$

Definition 2.2.1.2: Commutativity

$$x * y = y * x, \forall\, x, y \in \mathbb{G}$$

Property: If group $\mathbb{G}^*$ has Commutativity, then group $\mathbb{G}^*$ is an *Albanian Group*.

Definition 2.2.1.3: Identity Element

There exists an element $0 \in \mathbb{G}$ such:

$$a * 0 = 0 * a = a, \forall\, a \in G$$

Definition 2.2.1.4: Inverse Element

For $\forall\, a \in G$, $a \neq 0$, there exists a single element $a^{-1} \in \mathbb{G}$ such:

$$a * a^{-1} = a^{-1} * a = 0, \forall\, a \in G$$

2.2.2 Finite Fields

Defined over field $\mathbb{F}$ with the binary operator $*$, finite fields possess the same group definitions and properties previously mentioned in Section 2.2.1 [18] with the addition of the following:

Definition 2.2.2.1: Associativity of Closure under Multiplication

Given $a * (b * c) = c * (a * b) \in \mathbb{G}$:

$$a, b, c \in \mathbb{F}$$

Definition 2.2.2.2: Distributivity

$$a * (b * c) = c * (a * b) = a * bc = c * ab, \forall\, a, b, c \in \mathbb{F}$$

Definition 2.2.2.3: Multiplicative Identity

There exists an element $1 \in \mathbb{F}$ such:

$$a * 1 = 1 * a = a, \forall\, a \in \mathbb{F}$$

Definition 2.2.2.4: Multiplicative Inverse

For $\forall\, a \in \mathbb{F}$, $a \neq 0$, there exists a single element $a^{-1} \in \mathbb{F}$ such:

$$a * a^{-1} = a^{-1} * a = 1, \forall\, a \in \mathbb{F}$$

Finite fields are defined as $\mathbb{F} = \mathbb{Z}_n^*/f(x)$, where $f(x) \in \mathbb{F}$. A finite field is a field of finite length [41]. The field selection now rises as a design decision. Whether to implement a prime or binary field over various ECs with different key sizes is crucial. Changing any detail in the base preliminary design alters the entire architecture dramatically.

2.3 Levels of Security Within Public & Private Key Systems

All aspects of ECC applications are important to understand the capabilities of specifics design to be integrated into realizable cryptosystems. For example, the below table visually shows the sizable keys needed to provide 80, 112, 128, 192, & 256-bit levels of security.

Table 2.1: Achieving Standard Security - Keys

Symmetric	Example Algorithm	Prime Field	Binary Field	Usage
2^{80}	RSA-1024	$\|p\| = 2^{192}$	$m = 2^{163}$	Authentication
2^{112}	3DES	$\|p\| = 2^{224}$	$m = 2^{233}$	Authentication
2^{128}	AES-128	$\|p\| = 2^{256}$	$m = 2^{283}$	Confidentiality
2^{192}	AES-192	$\|p\| = 2^{384}$	$m = 2^{409}$	Confidentiality
2^{256}	SHA-256	$\|p\| = 2^{521}$	$m = 2^{571}$	Integrity

The algorithms above need to establish their targeted security level which is defined by the *key length*. Public-Key authentication can be developed by a digital signature algorithms such as the R. Rivest, A. Shamir, L. Adleman cipher (RSA)-1024 used by certificate authorities (CA) or a key-establishment can be implemented with EC Diffie Hellman (ECDH) key-exchange. Confidentiality is acquired by a symmetrical block cipher such as the Advanced Encryption Standard (AES) which is the leading method in preventing man-in-the-middle attacks by using a secret private-key. To gain integrity, or uniqueness, one needs to apply a hashing function with large complexity [1]. All of these systems need specific symmetrical cipher key lengths to ensure brute-forced attacks are negligible. Below, Equation (2.1) that displays the number of possibilities to be growing exponentially.

$$y(x) = 2^{m-1} \tag{2.1}$$

Clearly as m increases, computationally this calculation becomes impossible past 128-bits [37]. Modern ECC applications can work with notorious protocols like HyperText Transfer Protocol Secure (HTTPS) that readily use AES-128 implementations [19] to provide key-exchanges.

Practising ECC begins by choosing a key length, field, basis, and an elliptic curve. In the following subsections, those qualities will be considered.

2.3.1 Binary Field

A binary field can be defined as a field of which all 2^m elements are of radix-2 within a specified finite field and in this case, a Galois field ($GF(2^m)$). If $f(x)$ is an irreducible/primitive binary polynomial of size m-bit, $\mathbb{F}_{2^m} = GF(2^m)$ - the field is of *degree* m [39]. All elements within the field exhibit binary strings of length m-bit.

$$GF(2^m) = \{a(x) \mid a(x) = a_{m-1}x^{m-1} + ... + a_1x + a_0, x_i \in GF(2)\} \tag{2.2}$$

Equation (2.2) shows the Galois binary field $GF(2)$, to explicitly depict that the field and basis will be modulo 2. Normally this is implicit quality. All operations will be completed under the binary polynomial basis within this field. For the

scope of this project, binary fields of 233 and 283-bit will be tested and compared.

Another type of field is a *prime field*. Prime field encompass a set of integers of any prime p-radix, $[0, ..., p-1]$. All field calculations are computed over modulo p similarly to 2^m. These fields are typically implemented in software as they are computationally faster compared to binary fields while using multiple CPU cores.

2.3.2 Polynomial Basis

Polynomial or standard bases, are specified by a primitive polynomial of highest degree m. This polynomial acts as the irreducible string $(a_m...a_1a_0)$ in hardware of which all other element strings defined as $(a_{m-1}...a_1a_0)$ are concealed. Hence, all elements shown as a polynomial sum under the binary field's standard basis are shown below in Equation (2.3).

$$X = \sum_{i=0}^{m-1} a_i x^i, a_i \in GF(2^m) \tag{2.3}$$

Irreducible polynomials are chosen to be either trinomials or pentanomials depending on the m-bit size of the key being used. An example is using 233 and 283-bit keys; respectively, they need a trinomial and pentanomial to encompass the Galois field.

A primitive trinomial is defined as $t^m + t^n + 1$, where n is the lowest-degree middle term. If the trinomial basis is not available, the pentanomial defined by $t^m + t^x + t^y + t^z + 1$ has to be applied. Similarly x, y, z are the lowest-degree successive terms. Using the pentanomial forces sacrifices of larger memory usage (look-up table (LUT) on a FPGA), register complexion, and slower reduction computations [8].

The subsequent field arithmetic includes typical polynomial multiplication and addition modulo 2. Addition/subtraction in hardware will be simply be an m-bit exclusive-OR (XOR) gate. Further operations are designed using a polynomial basis. Hardware works end-to-end calculations in binary therefore introducing another basis such as a *normal* basis is cumbersome when targeting a larger goal

such as side-channel attack analysis.

Normal bases are quite popular in hardware and software implemented ECC protocols. Due to complexities with special class *Type T*, the normal basis proves superiority in specific situations regarding fast squaring operations [20]. Due to the difficulties testing and verifying hardware results using a normal basis, it will not be further attempted.

2.3.3 Provable Security

In 2003, standards such as Brainpool, used in German passports, or the National Security Agency (NSA) Suite B (2005) presently used in United States Department of Defence (DoD) security clearance projects [49] were and still are the modern ECC standards. Within Suite B, the National Institute of Standards and Technology (NIST) selected curves of which they have approved based on three main categories of curve parameters, the elliptic curve discrete logarithm problem (ECDLP) difficulty, and complex ECC security. A fantastic reference for a more detailed analysis of the applied algebraic security is found from cyber-security experts, Safecurves' website [47].

The most widely used curves that are the state-of-the-art are Montgomery, Kobliz, and Edwards prime and binary curves [21]. These special curves are optimized to produce maximum efficiency over the elliptic curve operations. Any of these EC equations would suffice as they are used in standards worldwide. However, Kobliz curve was selected due to accessible curve *order*, basis, and coefficients that are open sourced by NIST [34]. The order and curve coefficients will be in following section Elliptic Curves over $GF(2^m)$.

2.4 Elliptic Curves over $GF(2^m)$

Secure ECC FPGA implementations are extremely valuable due to the need of high-speed, low-cost, and rapid prototyping hardware, that can maintain high security asymmetric-key cryptography. Opposed to it's predecessors, RSA and DSA,

ECC uses much smaller keys, lower power consumption, and smaller memory usage all while providing the same level of security in any public-key system. This results in fewer clock cycles and reduced hardware overhead [45].

The security of ECC is based on the elliptic curve ECDLP; this allows ECC applications to have a smaller key size compared to RSA because the ECDLP is practically infeasible to solve versus the integer factorization problem [36].

2.4.1 Marginal Note on EC Discrete Logarithm Problem

The ECDLP is defined as the this following situation. Let an elliptic curve E defined over the finite Field $\mathbb{F}$, point P of order r and $Q \in P$, find $k\,[0, 1, .., r-1]$ such that the scalar multiplication (SM) $Q = kP$.

The positive integer k is the discrete logarithm of Q base P, $k = log_P Q$. Research has been conducted to break the ECDLP and the most prominent attack is the *Pollard Rho* method [22]. This method is improves the looping iteration-based methods, but still tries to break this algebraic problem iteratively in $3\sqrt{\frac{\pi m}{2}}$ cycles. As m, the bit size of 2^m increases exponentially, this becomes exceedingly unrealistic.

2.4.2 Curves and EC Group Theory

Below, Equation (2.4) is the pseudo-random curve; the NIST Kobliz curve (2.5) is a special case of Equation (2.4) where $b = 1$. When $b = 1$, operations within the finite field are highly simplified.

$$E : y^2 + xy = x^3 + ax^2 + b,\ a, b \in GF(2^m) \tag{2.4}$$

$$E_K : y^2 + xy = x^3 + ax^2 + 1,\ a \in GF(2^m) \tag{2.5}$$

To begin computing ECC operations, the base point $P(x, y) \in E_K$ needs to be selected. Many base points are applicable as long as they provide maximum order with respect to the curve.

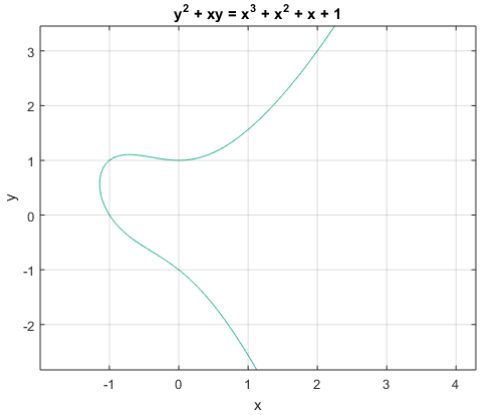


Figure 2.1: Kobliz Elliptic Curve - E_K :$y^2 + xy = x^3 + x^2 + 1$

To improve functionality of the EC, the *cofactor* should be minimized. The finite number of points on the EC is n defined by Equation (2.6) where $\mathbb{F}_q$ is the finite field.

Assuming $a = 1$, Equation (2.5), the cofactor f defined in Equation (2.7) and graphically displayed in Figure 2.1. The *order* of the base point is r, which multiplies point P to the theoretical point infinity. The order is a natural number while infinity is depicted as $\mathbb{O}$. The order is defined such that the minimum positive prime integer r such that $rP = \mathbb{O}$.

$$n = |\#E_K(\mathbb{F}_q) - (q+1)| \geq 2\sqrt{q} \tag{2.6}$$

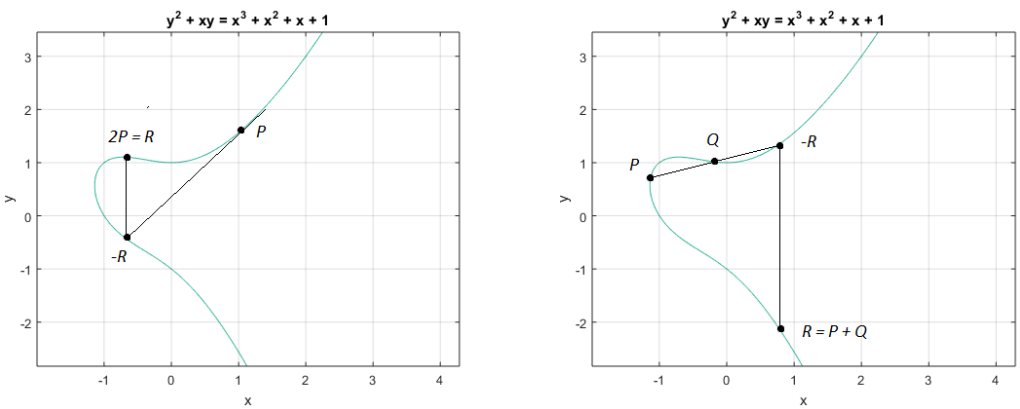
$$f = \frac{n}{r} = 2 \tag{2.7}$$

The identity element infinity implies $P + \mathbb{O} = P$. Therefore $P - P = \mathbb{O}$ at $(x, 0)$ implies $-P(x, -y)$. This is the modular compliment of base point P y-coordinate. Other group laws within ECC are as follows. If $x_1 = x_2$ & $y_1 \neq y_2$, then $y_2 = x_1 + y_1$ therefore $P_1 = -P_2$; if x_1 of P_1, then $2P_1 = \mathbb{O}$. To add two points on an elliptic curve E, one needs to check the simple condition of $Q = P$ or $Q \neq P$. If the points are equal, then point doubling follows.

2.4.3 Point Operations

If $P(x_1, y_1) = Q(x_2, y_2) \in E_K$, *point doubling* equations are needed to compute $2P(x_3, y_3) \in E_K$. Below in Equations (2.8), the set of Weierstrauss equations defining point doubling in affine coordinates[1]. In Figure 2.2(a), a tangent line is drawn from the point P that intersects the curve at point $-R$. Once reflected upon the x-axis, the point $2P$ is found.

$$\begin{aligned} \lambda &= \frac{y_1}{x_1} + x_1 \\ x_3 &= \lambda^2 + \lambda + a \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \end{aligned} \tag{2.8}$$



(a) Point Doubling

(b) Point Addition

Figure 2.2: Kobliz Elliptic Curve - Point Operations

When $P(x_1, y_1) \neq Q(x_2, y_2) \in E_K$, *point addition* equations are needed to compute $P + Q = R(x_3, y_3) \in E_K$. Again, Weierstrauss equations defining point addition in affine coordinates are Equations (2.9). In Figure 2.2(b), a tangent line is drawn connecting P and Q that intersects the curve at point $-R$. Once reflected upon the x-axis, the point $R = P + Q$ is established.

$$\begin{aligned} \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \end{aligned} \tag{2.9}$$

[1]Affine coordinates are (x, y) which span an indefinite xy-plane. They are the realizable coordinates compared to other methods like *projective coordinates* [8].

2.4.4 ECC Overview and Vulnerability Insight

Understanding the necessary background of ECs is vital to recognize potential security threats on all levels. The hierarchy of ECC protocols is shown in Figure 2.3. There are 3 sets of operations that build the echelon from the ground up. Firstly, the multiplication and inversion methods under finite field arithmetic, second the point operations, and lastly the scalar multiplication. As previously explained, addition/subtraction is the same operation under $GF(2)$ and is simply an XOR. Each step in the hierarchy will be discussed in much greater detail while designing the architecture in chapters 4 and 5.

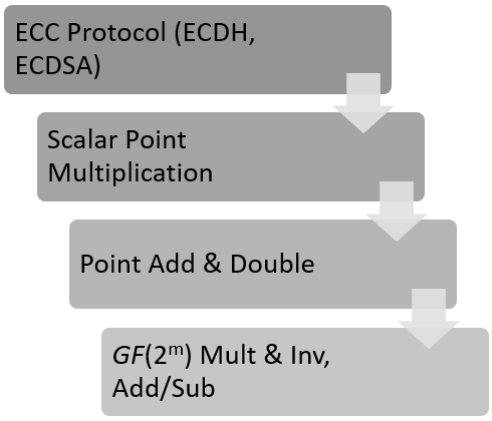


Figure 2.3: ECC $GF(2^m)$ Hierarchy of Operations

The primitive operation exercising all lower operations is the scalar point multiplication. This SM is the pronounced task of popular protocols such as ECDH or an EC digital signature algorithm (ECDSA). This makes this operation one of the biggest security risks in ECC.

2.5 Statistical Analysis for Side-Channel Attacks

In reference to [38], the models and algorithms in this section are needed in differential power analysis (DPA) in order to carry out the analysis. These two concepts are the essential basics behind DPA and are explained using the procedure of AES.

The Hamming Distance (HD) model is used to measure bus activity within the

selected device. This activity is directly related to the output power on the bus. HD is the number of bit changes or bit inversions, in a binary word. With respect to the next chapters analysis, the change in the output bit stream is the count 1's from logical XORs between two words and is calculated by the following Equation (2.10). This count is defined as the Hamming Weight (HW).

$$D_H = \sum_{i=1}^{k} |x_i - y_i|,\ x_i, y_i \in [0, 1] \tag{2.10}$$

This will give a precise digital average of any state with a system for further statistical analysis. This simple yet powerful model will be used to map the hypothetical power consumption values to the hypothetical intermediate values.

After the appropriate values are mapped, the resulting matrix must have a strong correlation with the power traces previously captured at a specific key.

$$R = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{2.11}$$

The correlation coefficient R, is calculated with the hypothetical power consumption versus the traces.

Chapter 3

Side-Channel Attacks Against Hardware

Side-Channel Attacks are invasive or non-invasive manoeuvres to exploit physical leakages of information from hardware. Timing signals, register to register dependencies, and physical power consumption are a few pieces of information that can be easily obtained from hardware implementations through SCAs. Three rising SCAs are the timing attacks, safe-error attacks, and differential power analysis. These attacks all possess the ability to extract different pieces of information from cryptographic accelerators.

"If you think technology can solve your security problems, then you don't understand the problems and you don't understand the technology" (B. Schneier, 2000).

In this chapter, the concepts of side-channel attacks will become clear and a side-channel attack is performed on a notorious 128-bit encryption standard to exploit it's unique flaws. A description of steps needed to perform the attack on a different system will be displayed.

3.1 Malicious Actions Against Hardware

As emerging techniques of attacks on hardware devices seem relentless, there are three types that remain as the most pertinent issues. These attacks will be described acknowledging physical weaknesses within the hardware and small issues related to SPM algorithms.

3.1.1 Timing & Safe-Error Attacks

Timing attacks rely on the fact that operations on different inputs have a large time variance [4]. This gives the attacker the non-invasive ability to measure the time between computations of the attacked algorithm.

As shown in recent literature, [31, 32], timing attacks are sometimes focused against software implemented cryptosystems. These attacks would rely on the inter-process times through the state of the CPU's cache as it reads and writes data. This leads to leakage memory access patterns which can be made to make data dependant look-up table and break the system at hand.

These methods are easily transferable to software-hardware SoC implementations as they rely on the CPU to transmit, receive, and store values in memory while the hardware computes the encryption.

The timing attack employed against a hardware implementation needs a CPU regularly communicating with it's cache in order to effectively complete the hack. Since the hardware implementation is not at the integration level, this attack will be a candidate for future work as explained in Conclusions.

Safe-error attacks maliciously modify bits of a specific word in a specified register [3] to determine if the registers are independent of one another. This invasively shows the direct register dependencies that distinguishes parts or an entire algorithm from another. This attack needs to physically tamper with the hardware in order to falsify words, or to introduce a fake instruction [3] in the internal

arithmetic logic unit (ALU) to trigger a fault resulting in the adversary inquiring sensitive information.

Safe-error attacks are categorized as computational safe-error (C Safe-error), focused on tampering with the ALU or memory safe-error (M Safe-error) which modifies CPU to memory address communication [6].

These attacks are primarily out of the scope of this work, but need to be mentioned as they are a prominent SCA.

3.1.2 Zero Point Attacks

The Zero-Value Point (ZVP) attacks on ECC processors were introduced in [16]. The attackers choose a specific base point on an EC to produce the zero-value coordinate in the scalar multiplication. The power consumption of the zero-value multiplication will dramatically decreases therefore, exposing secret key distinguished by single observation of a set of power traces. This requires the attacker to have physical access to the processor and or the CPU's memory to tamper with the embedded base point for the SPM. Having said that, this knowledge of the ZVP power consumption can be applied with the help of another attack to differentiate the key from intermediate scalar values.

3.1.3 Differential Power Analysis

Correlation power analysis (CPA) is widely notorious in the domains of embedded security. CPA focuses on reading the leakage power from the encryption stage of a device and relates it to the inputted data stream. This could be through electromagnetic radiation or passively sniffing output bus activity. It's first derivative was simple power analysis (SPA) which later became a shadow to it's sibling, differential power analysis (DPA) [24].

DPA was announced to the public in 1998 by researchers P. Kocher, J. Jaffe, and B. Jun. It is a type of CPA where the attacker non-invasively reads the output power consumption of the cryptographic processor to differentially compare those

results to potential state or key values. The assaulter then runs a series of statistical processes giving the ability to learn inner-mechanisms or variables within the core i.e. secret session and master keys.

This manoeuvre relies on the fact that internal switching of CMOS technology consumes different amounts of power depending on different inputs' operations. This type of CPA would be a efficient, adaptable, and more importantly, a feasible attack to a wide set of cryptosystems.

3.2 Novelty

The section titled *Executing DPA on AES-128* is novel work that expands the broader scope of the past research such as [25, 26, 33, 42] to detail exact algebraic steps with explanation in order to successfully hack the hardware implementation of AES-128. To the best of the authors knowledge, there is no research that outlines the detail of DPA to that of this thesis.

This detail is needed due to the elaborate steps and cryptographic insight of where to attack and why. Understanding why the proposed attack works at a hardware level is paramount for applying the practise for future research.

3.3 Executing DPA on AES-128

Due to the complexity of the AES and the fact that a brute-force attack is impossible in any life time, the encryption is viewed as an excellent option to handle sensitive data for high-level security of 128-256 bits in reference to Table D.1.

In order to validate the security of data being processed through AES-128 in electronic code book (ECB) configuration, the standard must be exposed to exploit it's flaws to propose solutions to issues in both software and hardware. This attack will uncover the vulnerability of the hardware implementation of the Advanced Encryption Standard to differential power analysis.

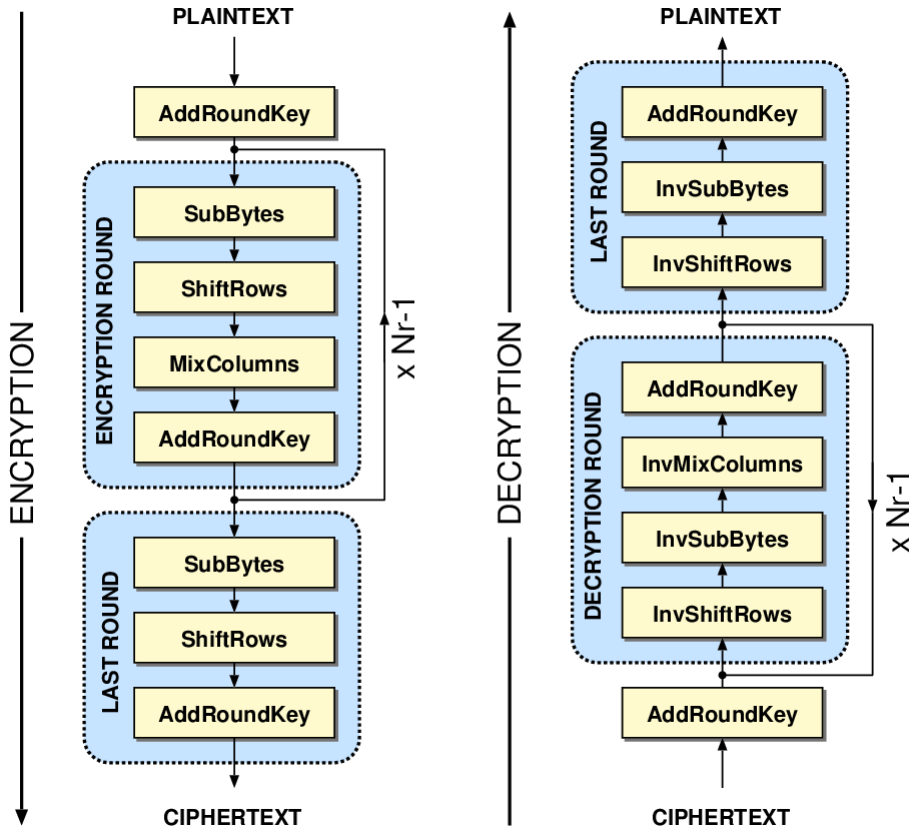


Figure 3.1: Block Diagram of AES-128 [38]

As a guideline for the process of AES, the block diagram of AES-128 is shown above in Figure 3.1.

3.3.1 Experimental Setup

The three main hardware components of the attack include a cryptographic FPGA evaluation board, an oscilloscope, and a computer. The Side-channel Attack Standard Evaluation Board (SASEBO)-GIII is the cryptographic research and development board that is used to perform two tasks on two FPGAs. The data transfer mitigation of plaintext and ciphertext are steered through Spartan-6, the controlling FPGA, while Virtex-7, the processing FPGA, symmetrically encrypts the plaintext from a master key established. The random data is manipulated in software from a C# open-sourced script [50].

1. Cryptographic FPGA Evaluation Board: SASEBO-GIII

2. Oscilloscope: Agilent Technologies DSO-X 3012A at 50 M/s samples

3. Computer: Intel Xeon 64-bit 3.0 GHz Processor with 8 GB Memory

Below, in Figure 3.2, the board is labelled 1-6 and the labels are as follows. 1 output power pin of the hardware encryption bus, 2 output power pin that triggers the oscilloscope set at 50 M/s samples to capture a power traces, 3 Virtex-7 FPGA, 4 Spartan-6 FPGA , and 5 & 6 is the Joint Test Action Group (JTAG) port to program the the corresponding FPGAs electrically erasable programmable read-only memory (EEPROM) with the combinational AES-128 implementation.

On the bottom left of the board, not labelled, is the universal standard bus (USB) 2.0 that is the bi-lateral data transfer connection the communicates with the Spartan-6.

The three main software components being used on are Xilinx ISE, Visual Studio, and Matlab.

Xilinx ISE is the design suite used to modify and compile Verilog HDL code for the both FPGAs. The open-sourced HDL scripts were used from [50] since the scope of this project is not to design a hardware implementation of AES-128 but rather exploit the standard's flaws; Visual Studio is the environment of choice. Matlab is utilized to develop the entire DPA attacking algorithm since it is tailored to analyze and import very large matrices with ease to sort them accordingly.

The provided scripts from [50] are modified to establish a connection to the evaluation board through the USB 2.0 and to display a graphical user interface (GUI) that allows the user to view the hexadecimal values of the variables being processed by the board; the GUI also allows the user to manipulate the AES-128 master key random 16-byte keys.

3.3.2 Framework Differential Power Analysis

The FPGA consumes characteristic power due to the exertion of words pushed from the output pins of the Kintex-7 to the Spartan-6 as the switching activity

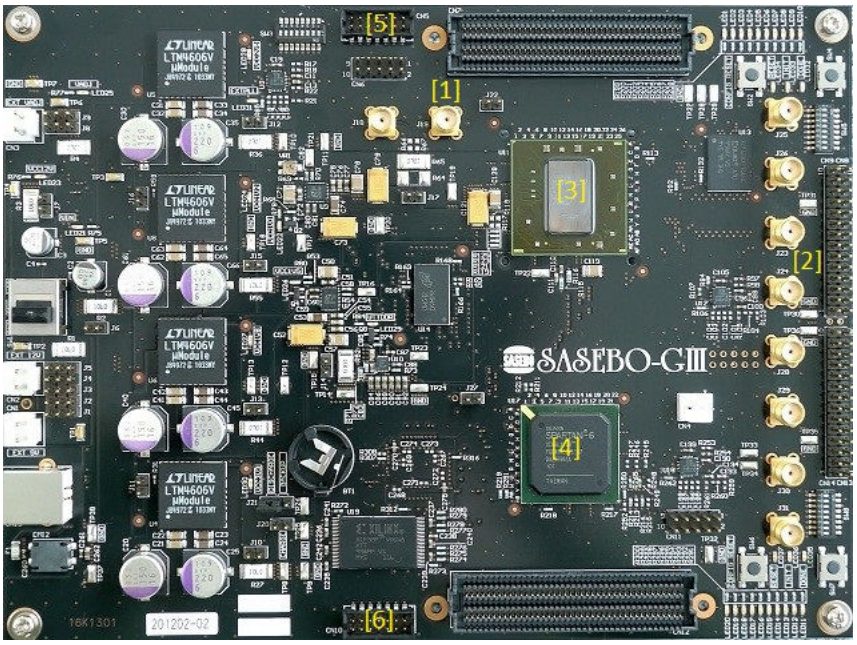


Figure 3.2: SASEBO-GIII

1 Output power encryption bus, 2 Output power trigger, 3 Virtex-7, 4 Spartan-6, and 5, 6 JTAG port to program EEPROMs

from internal signals changes.

The pin on the output of the encrypted text bus can be probed to read the power traces. Each bit on the bus requires power in order to invert itself after each clock cycle. This means that the power consumption is directly proportional to the number of bit changes. Therefore, if one has a known state, ciphertext or plaintext, and all hypothetical possibilities for a neighbouring state, they could can count the number of bit changes between each hypothetical and the known state to correlate it to the power consumption in order to find out which trace it corresponds to.

The Kintex-7 performs an entire 1/10 rounds of AES-128 on 1 byte before changing the values on the bus. This translates to investigating a whole round of AES to get all hypothetical states at the neighbouring round. Due to the absence of Mix-Columns, a $GF(2^8)$ operation [41] in the 10th round, the simple choice to use the ciphertext as the known value. Working backwards to get every hypothetical value of station 9 labelled as ST9 as shown in Figure 3.3, will be executed in next subsections. Using these values, the HD is obtained between the two refer-

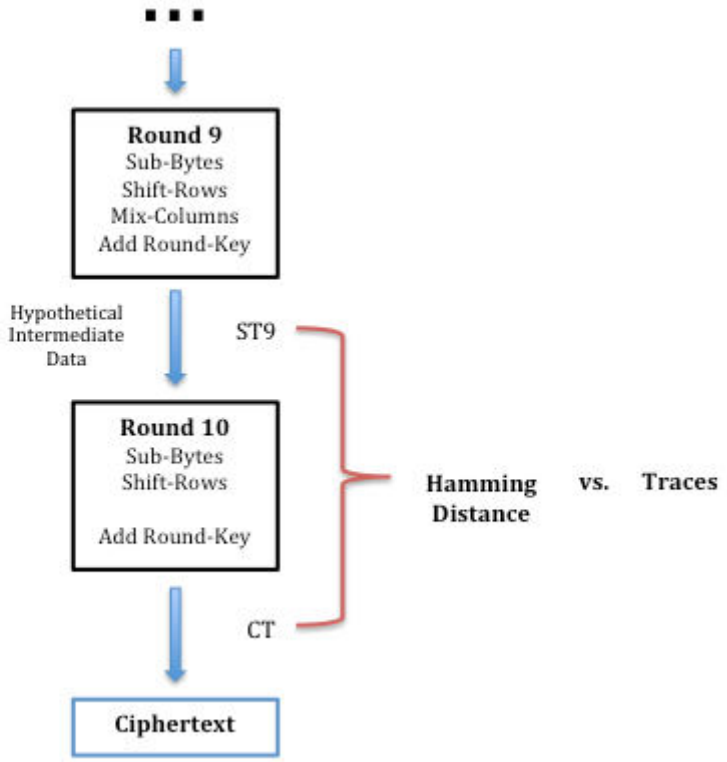


Figure 3.3: Block Diagram of DPA Against AES-128

ence states and the correlation coefficient is used to match these distances to the power consumption from their respective traces captured.

A momentous observation that enables this attack possible is that each byte of the 16-byte key are independent of each other at each n-state and all operations on them are essentially in parallel - this is true in software as well. Clearly this is a large flaw in the algorithm and due to the nature of this attack there is no byte-to-byte single state key dependencies, but rather state-to-state map below.

$$[B_{15}(n), B_{14}(n), ..., B_0(n)] \longrightarrow [B_{15}(n+1), B_{14}(n+1), ..., B_0(n+1)] \tag{3.1}$$

There are 7 operations required in the developed algorithm and 3 of which are inverse operations of the AES-128 algorithm. The other 4 are procedures to develop hypothetical 1-byte keys, correlation coefficients, and lastly the inverse sessional key.

Figure 3.3 above should be used in tangent with Figure 3.1 to understand the concepts discussed and the operations that follow. To begin, the known 128-bit

ciphertext (CT) is split into 16 bytes as seen below.

$$\mathbf{CT} = \begin{array}{c} \begin{array}{ccccc} Byte1 & Byte2 & & Byte15 & Byte16 \end{array} \\ \begin{pmatrix} CT_1 & CT_2 & ... & CT_{15} & CT_{16} \end{pmatrix} \end{array}$$

3.3.3 Generating Hypothetical Keys

Working backwards, the first operation encountered is the Add-Round Key. Since the 10th round sessional key is unknown and what is being sought, all 0-255 possibilities for each byte is generated below. All of the AES operations and the correlations are computed on bytes, not bits, which is why it is sufficient to capture every hypothetical value of each byte rather than each bit of the possible 128-bit key.

$$\mathbf{Hyp.Keys} = \begin{array}{ccccc} Byte1 & & Byte2 & & Byte16 \\ \begin{pmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{pmatrix} & & \begin{pmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{pmatrix} & \cdots & \begin{pmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{pmatrix} \end{array}$$

3.3.4 Inverse Add-Round Key & Shift Row

The add-round key operation takes the input data and XORs it with the sessional key in order to get the ciphertext output. The hypothetical keys and the ciphertext are XORd in order to get the input in Equation (3.2).

$$A = CT \oplus Key \in [0, 1] \tag{3.2}$$

Each ciphertext byte is XORd with its corresponding 256 possibilities of the key. In other words, the first byte of the ciphertext is XORd with every guess in byte one of the key and the rest of the bytes follow the same operation.

$$\mathbf{A} = \begin{array}{c} Byte1 \\ \left(\begin{array}{c} \begin{bmatrix} CT_1 \\ CT_1 \\ CT_1 \\ \vdots \\ CT_1 \end{bmatrix} \\ \oplus \\ \begin{bmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{bmatrix} \end{array}\right) \end{array} \begin{array}{c} Byte2 \\ \left(\begin{array}{c} \begin{bmatrix} CT_2 \\ CT_2 \\ CT_2 \\ \vdots \\ CT_2 \end{bmatrix} \\ \oplus \\ \begin{bmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{bmatrix} \end{array}\right) \end{array} \cdots \begin{array}{c} Byte16 \\ \left(\begin{array}{c} \begin{bmatrix} CT_{16} \\ CT_{16} \\ CT_{16} \\ \vdots \\ CT_{16} \end{bmatrix} \\ \oplus \\ \begin{bmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{bmatrix} \end{array}\right) \end{array}$$

To simplify the following steps, we name this matrix as A, which has the following configuration.

$$\mathbf{A} = \begin{array}{c} Byte1 \\ \begin{pmatrix} A_1[1] \\ A_1[2] \\ A_1[3] \\ \vdots \\ A_1[256] \end{pmatrix} \end{array} \begin{array}{c} Byte2 \\ \begin{pmatrix} A_2[1] \\ A_2[2] \\ A_2[3] \\ \vdots \\ A_2[256] \end{pmatrix} \end{array} \cdots \begin{array}{c} Byte16 \\ \begin{pmatrix} A_{16}[1] \\ A_{16}[2] \\ A_{16}[3] \\ \vdots \\ A_{16}[256] \end{pmatrix} \end{array}$$

To perform the *shift row* operation, the 16 bytes of data are rearranged in a 4x4 matrix. Each row has a shift left operation of value 0, 1, 2, and 3, respectively. In order to do the inverse shift row (ISR), Mat A is rearranged in a 4x4 formation and each row is shifted right by 0, 1, 2, and 3.

$$\begin{bmatrix} \begin{bmatrix} A_1[1] \\ \vdots \\ A_1[256] \end{bmatrix} & \begin{bmatrix} A_5[1] \\ \vdots \\ A_5[256] \end{bmatrix} & \begin{bmatrix} A_9[1] \\ \vdots \\ A_9[256] \end{bmatrix} & \begin{bmatrix} A_{13}[1] \\ \vdots \\ A_{13}[256] \end{bmatrix} \\ \begin{bmatrix} A_2[1] \\ \vdots \\ A_2[256] \end{bmatrix} & \begin{bmatrix} A_6[1] \\ \vdots \\ A_6[256] \end{bmatrix} & \begin{bmatrix} A_{10}[1] \\ \vdots \\ A_{10}[256] \end{bmatrix} & \begin{bmatrix} A_{14}[1] \\ \vdots \\ A_{14}[256] \end{bmatrix} \\ \begin{bmatrix} A_3[1] \\ \vdots \\ A_3[256] \end{bmatrix} & \begin{bmatrix} A_7[1] \\ \vdots \\ A_7[256] \end{bmatrix} & \begin{bmatrix} A_{11}[1] \\ \vdots \\ A_{11}[256] \end{bmatrix} & \begin{bmatrix} A_{15}[1] \\ \vdots \\ A_{15}[256] \end{bmatrix} \\ \begin{bmatrix} A_4[1] \\ \vdots \\ A_4[256] \end{bmatrix} & \begin{bmatrix} A_8[1] \\ \vdots \\ A_8[256] \end{bmatrix} & \begin{bmatrix} A_{12}[1] \\ \vdots \\ A_{12}[256] \end{bmatrix} & \begin{bmatrix} A_{16}[1] \\ \vdots \\ A_{16}[256] \end{bmatrix} \end{bmatrix}$$

$$\textbf{ISR} \downarrow$$

$$\mathbf{B} = \begin{bmatrix} \begin{bmatrix} A_1[1] \\ \vdots \\ A_1[256] \end{bmatrix} & \begin{bmatrix} A_5[1] \\ \vdots \\ A_5[256] \end{bmatrix} & \begin{bmatrix} A_9[1] \\ \vdots \\ A_9[256] \end{bmatrix} & \begin{bmatrix} A_{13}[1] \\ \vdots \\ A_{13}[256] \end{bmatrix} \\ \begin{bmatrix} A_{14}[1] \\ \vdots \\ A_{14}[256] \end{bmatrix} & \begin{bmatrix} A_2[1] \\ \vdots \\ A_2[256] \end{bmatrix} & \begin{bmatrix} A_6[1] \\ \vdots \\ A_6[256] \end{bmatrix} & \begin{bmatrix} A_{10}[1] \\ \vdots \\ A_{10}[256] \end{bmatrix} \\ \begin{bmatrix} A_{11}[1] \\ \vdots \\ A_{11}[256] \end{bmatrix} & \begin{bmatrix} A_{15}[1] \\ \vdots \\ A_{15}[256] \end{bmatrix} & \begin{bmatrix} A_3[1] \\ \vdots \\ A_3[256] \end{bmatrix} & \begin{bmatrix} A_7[1] \\ \vdots \\ A_7[256] \end{bmatrix} \\ \begin{bmatrix} A_8[1] \\ \vdots \\ A_8[256] \end{bmatrix} & \begin{bmatrix} A_{12}[1] \\ \vdots \\ A_{12}[256] \end{bmatrix} & \begin{bmatrix} A_{16}[1] \\ \vdots \\ A_{16}[256] \end{bmatrix} & \begin{bmatrix} A_4[1] \\ \vdots \\ A_4[256] \end{bmatrix} \end{bmatrix}$$

To simplify the following steps, we name this matrix as B, which has the following configuration.

$$\mathbf{B} = \begin{bmatrix} \begin{bmatrix} B_1[1] \\ \vdots \\ B_1[256] \end{bmatrix} & \begin{bmatrix} B_5[1] \\ \vdots \\ B_5[256] \end{bmatrix} & \begin{bmatrix} B_9[1] \\ \vdots \\ B_9[256] \end{bmatrix} & \begin{bmatrix} B_{13}[1] \\ \vdots \\ B_{13}[256] \end{bmatrix} \\ \begin{bmatrix} B_2[1] \\ \vdots \\ B_2[256] \end{bmatrix} & \begin{bmatrix} B_6[1] \\ \vdots \\ B_6[256] \end{bmatrix} & \begin{bmatrix} B_{10}[1] \\ \vdots \\ B_{10}[256] \end{bmatrix} & \begin{bmatrix} B_{14}[1] \\ \vdots \\ B_{14}[256] \end{bmatrix} \\ \begin{bmatrix} B_3[1] \\ \vdots \\ B_3[256] \end{bmatrix} & \begin{bmatrix} B_7[1] \\ \vdots \\ B_7[256] \end{bmatrix} & \begin{bmatrix} B_{11}[1] \\ \vdots \\ B_{11}[256] \end{bmatrix} & \begin{bmatrix} B_{15}[1] \\ \vdots \\ B_{15}[256] \end{bmatrix} \\ \begin{bmatrix} B_4[1] \\ \vdots \\ B_4[256] \end{bmatrix} & \begin{bmatrix} B_8[1] \\ \vdots \\ B_8[256] \end{bmatrix} & \begin{bmatrix} B_{12}[1] \\ \vdots \\ B_{12}[256] \end{bmatrix} & \begin{bmatrix} B_{16}[1] \\ \vdots \\ B_{16}[256] \end{bmatrix} \end{bmatrix}$$

3.3.5 Inverse S-Box

The S-box takes each byte of data and maps them to a given well-established value. The inverse S-box is a standard 16x16 array that simply maps the inverse output of AES's Substitute Box operation. It takes the hex or decimal value of each byte and exchanges it with a new value. It accomplishes this by selecting the most significant 4 bits of the code word of each byte as the row of the standard array and the least significant 4 bits as the column. Every byte in the Mat B is remapped through the developed inverse S-box in order to get the new Mat C.

$$\mathbf{C} = \begin{bmatrix} \begin{bmatrix} sbox^{-1}[B_1[1]] \\ \vdots \\ sbox^{-1}[B_1[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_5[1]] \\ \vdots \\ sbox^{-1}[B_5[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_9[1]] \\ \vdots \\ sbox^{-1}[B_9[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_{13}[1]] \\ \vdots \\ sbox^{-1}[B_{13}[256]] \end{bmatrix} \\ \begin{bmatrix} sbox^{-1}[B_2[1]] \\ \vdots \\ sbox^{-1}[B_2[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_6[1]] \\ \vdots \\ sbox^{-1}[B_6[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_{10}[1]] \\ \vdots \\ sbox^{-1}[B_{10}[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_{14}[1]] \\ \vdots \\ sbox^{-1}[B_{14}[256]] \end{bmatrix} \\ \begin{bmatrix} sbox^{-1}[B_3[1]] \\ \vdots \\ sbox^{-1}[B_3[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_7[1]] \\ \vdots \\ sbox^{-1}[B_7[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_{11}[1]] \\ \vdots \\ sbox^{-1}[B_{11}[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_{15}[1]] \\ \vdots \\ sbox^{-1}[B_{15}[256]] \end{bmatrix} \\ \begin{bmatrix} sbox^{-1}[B_4[1]] \\ \vdots \\ sbox^{-1}[B_4[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_8[1]] \\ \vdots \\ sbox^{-1}[B_8[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_{12}[1]] \\ \vdots \\ sbox^{-1}[B_{12}[256]] \end{bmatrix} & \begin{bmatrix} sbox^{-1}[B_{16}[1]] \\ \vdots \\ sbox^{-1}[B_{16}[256]] \end{bmatrix} \end{bmatrix}$$

To simplify the following steps, we name this new matrix as C, which has the following configuration.

$$\mathbf{C} = \begin{bmatrix} \begin{bmatrix} C_1[1] \\ \vdots \\ C_1[256] \end{bmatrix} & \begin{bmatrix} C_5[1] \\ \vdots \\ C_5[256] \end{bmatrix} & \begin{bmatrix} C_9[1] \\ \vdots \\ C_9[256] \end{bmatrix} & \begin{bmatrix} C_{13}[1] \\ \vdots \\ C_{13}[256] \end{bmatrix} \\ \begin{bmatrix} C_2[1] \\ \vdots \\ C_2[256] \end{bmatrix} & \begin{bmatrix} C_6[1] \\ \vdots \\ C_6[256] \end{bmatrix} & \begin{bmatrix} C_{10}[1] \\ \vdots \\ C_{10}[256] \end{bmatrix} & \begin{bmatrix} C_{14}[1] \\ \vdots \\ C_{14}[256] \end{bmatrix} \\ \begin{bmatrix} C_3[1] \\ \vdots \\ C_3[256] \end{bmatrix} & \begin{bmatrix} C_7[1] \\ \vdots \\ C_7[256] \end{bmatrix} & \begin{bmatrix} C_{11}[1] \\ \vdots \\ C_{11}[256] \end{bmatrix} & \begin{bmatrix} C_{15}[1] \\ \vdots \\ C_{15}[256] \end{bmatrix} \\ \begin{bmatrix} C_4[1] \\ \vdots \\ C_4[256] \end{bmatrix} & \begin{bmatrix} C_8[1] \\ \vdots \\ C_8[256] \end{bmatrix} & \begin{bmatrix} C_{12}[1] \\ \vdots \\ C_{12}[256] \end{bmatrix} & \begin{bmatrix} C_{16}[1] \\ \vdots \\ C_{16}[256] \end{bmatrix} \end{bmatrix}$$

3.3.6 Hamming Distance

At this point, Mat C corresponds to every possible value/state for each byte of data at ST9, reference Figure 3.3 for the known ciphertext. Now the hamming distance between each of these values and their corresponding ciphertext must be calculated in order to get the number of bit changes on the bus between ST9 and the ciphertext. Equation (2.10) is applied to calculate the HD. In order to get the HD, all 256 values of byte 1 in Mat C are XORd with the first byte of ciphertext. This will be repeated for all 16 bytes.

$$\mathbf{I} = \begin{array}{c} Byte1 \\ \left(\begin{array}{c} \begin{bmatrix} C_1[1] \\ C_1[2] \\ C_1[3] \\ \vdots \\ C_1[256] \end{bmatrix} \\ \oplus \\ \begin{bmatrix} CT[1] \\ CT[1] \\ CT[1] \\ \vdots \\ CT[1] \end{bmatrix} \end{array}\right) \end{array} \begin{array}{c} Byte2 \\ \left(\begin{array}{c} \begin{bmatrix} C_2[1] \\ C_2[2] \\ C_2[3] \\ \vdots \\ C_2[256] \end{bmatrix} \\ \oplus \\ \begin{bmatrix} CT[2] \\ CT[2] \\ CT[2] \\ \vdots \\ CT[2] \end{bmatrix} \end{array}\right) \end{array} \cdots \begin{array}{c} Byte16 \\ \left(\begin{array}{c} \begin{bmatrix} C_{16}[1] \\ C_{16}[2] \\ C_{16}[3] \\ \vdots \\ C_{16}[256] \end{bmatrix} \\ \oplus \\ \begin{bmatrix} CT[16] \\ CT[16] \\ CT[16] \\ \vdots \\ CT[16] \end{bmatrix} \end{array}\right) \end{array}$$

To simplify the following steps, we name this matrix as I, which has the following configuration.

$$\mathbf{I} = \begin{array}{c} Byte1 \\ \begin{pmatrix} I_1[1] \\ I_1[2] \\ I_1[3] \\ \vdots \\ I_1[256] \end{pmatrix} \end{array} \begin{array}{c} Byte2 \\ \begin{pmatrix} I_2[1] \\ I_2[2] \\ I_2[3] \\ \vdots \\ I_2[256] \end{pmatrix} \end{array} \cdots \begin{array}{c} Byte16 \\ \begin{pmatrix} I_{16}[1] \\ I_{16}[2] \\ I_{16}[3] \\ \vdots \\ I_{16}[256] \end{pmatrix} \end{array}$$

Using HW model described in the previous chapter, the count of the number of bit changes is shown.

$$\mathbf{F} = \begin{array}{ccc} Byte1 & Byte2 & Byte16 \\ \begin{pmatrix} HW(I_1[1]) \\ HW(I_1[2]) \\ HW(I_1[3]) \\ \vdots \\ HW(I_1[256]) \end{pmatrix} & \begin{pmatrix} HW(I_2[1]) \\ HW(I_2[2]) \\ HW(I_2[3]) \\ \vdots \\ HW(I_2[256]) \end{pmatrix} \cdots & \begin{pmatrix} HW(I_{16}[1]) \\ HW(I_{16}[2]) \\ HW(I_{16}[3]) \\ \vdots \\ HW(I_{16}[256]) \end{pmatrix} \end{array}$$

In conclusion, the 256 hypothetical power consumption values for each byte is left as seen in Mat F below. This entire process is repeated for every trace that is captured.

$$\mathbf{F} = \begin{array}{ccc} Byte1 & Byte2 & Byte16 \\ \begin{pmatrix} F_1[1] \\ F_1[2] \\ F_1[3] \\ \vdots \\ F_1[256] \end{pmatrix} & \begin{pmatrix} F_2[1] \\ F_2[2] \\ F_2[3] \\ \vdots \\ F_2[256] \end{pmatrix} \cdots & \begin{pmatrix} F_{16}[1] \\ F_{16}[2] \\ F_{16}[3] \\ \vdots \\ F_{16}[256] \end{pmatrix} \end{array}$$

3.3.7 Correlation Coefficient

The columns of Mat F below are correlated against each sample point value's columns of the power traces. This is the reason why having a precise trigger on the oscilloscope that occurs on same sample of the trace is important. The waveforms must overlap over each sample to get the true bus change in power consumption for the highest correlation. This correlation is repeated for all 256 hypothetical ST9 values.

The notation following is $F_{Trace,ColumnF}$[Row of F] and $T_{TraceNumber}$[Sample Number]. These matrices are built for an example byte 1 of 16.

$$\mathbf{F_1[1]} = \begin{bmatrix} F_{1,1}[1] \\ F_{2,1}[1] \\ F_{3,1}[1] \\ \vdots \\ F_{n,1}[1] \end{bmatrix} \quad \begin{bmatrix} T_1[1] & T_1[2] & T_1[3] & \dots & T_1[k] \\ T_2[1] & T_2[2] & T_2[3] & \dots & T_2[k] \\ T_3[1] & T_3[2] & T_3[3] & \dots & T_3[k] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ T_n[1] & T_n[2] & T_n[3] & \dots & T_n[k] \end{bmatrix}$$

$$\mathbf{F_1[2]} = \begin{bmatrix} F_{1,1}[2] \\ F_{2,1}[2] \\ F_{3,1}[2] \\ \vdots \\ F_{n,1}[2] \end{bmatrix} \quad \begin{bmatrix} T_1[1] & T_1[2] & T_1[3] & \dots & T_1[k] \\ T_2[1] & T_2[2] & T_2[3] & \dots & T_2[k] \\ T_3[1] & T_3[2] & T_3[3] & \dots & T_3[k] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ T_n[1] & T_n[2] & T_n[3] & \dots & T_n[k] \end{bmatrix}$$

$$\vdots$$

$$\mathbf{F_1[256]} = \begin{bmatrix} F_{1,1}[256] \\ F_{2,1}[256] \\ F_{3,1}[256] \\ \vdots \\ F_{n,1}[256] \end{bmatrix} \quad \begin{bmatrix} T_1[1] & T_1[2] & T_1[3] & \dots & T_1[k] \\ T_2[1] & T_2[2] & T_2[3] & \dots & T_2[k] \\ T_3[1] & T_3[2] & T_3[3] & \dots & T_3[k] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ T_n[1] & T_n[2] & T_n[3] & \dots & T_n[k] \end{bmatrix}$$

The highest correlation among these hypothetical ST9 values will be the sessional key result for the byte under analysis. Therefore, if the highest correlation occurs using the $F[41]$ values, that means that the sessional key has a value of $40-1$, due to the index of Matlab. This is because the key guesses were initially XORd into the ciphertext with the values of 0-255 so in the end, the index of the successful byte actually corresponds to the key value. Again, it is stressed that this algorithm done for all 16 bytes of the key as they are independent of each other.

$$\text{Exp.Keys} = \begin{array}{ccc} Byte1 & Byte2 & Byte16 \\ \begin{pmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{pmatrix} & \begin{pmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{pmatrix} \cdots & \begin{pmatrix} 00000000 \\ 00000001 \\ 00000010 \\ \vdots \\ 11111111 \end{pmatrix} \end{array}$$

3.3.8 Inverse Sessional Round Key

The result from the correlation above is the 10th round sessional key since we are attacking the 10th round. This means that the sessional key needs to be an inverse of 10 rounds in order to get the master key. The Python inverse sessional key open-sourced script [51] for a given DPA result is used to provide the master key. This can be done as the sessional key generator is predictable and easily calculated with a given input string. The figure below shows the reverse operation on a 16-byte string assuming that the sessional key DPA result is: '$00\,01\,02\ldots0F$'. The master key is highlighted along with the expected DPA result both in hexadecimal.

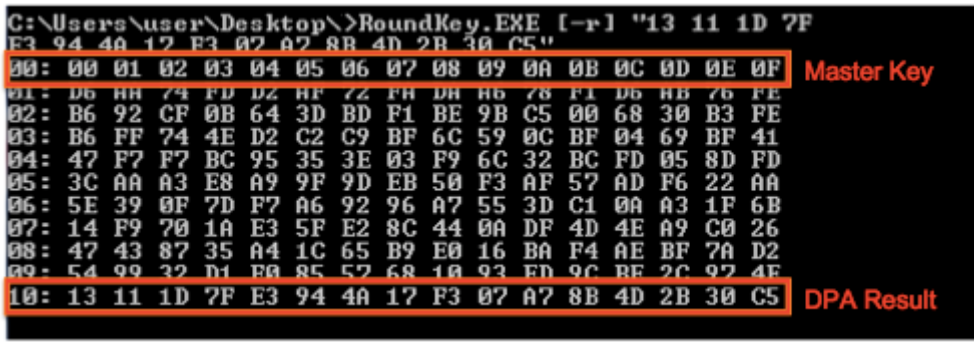


Figure 3.4: 10 Rounds of Inverse Session Key

3.3.9 Results & Analysis of Vulnerabilities

The results of the DPA algorithm were previously discussed as the 10th sessional key within AES-128. The last matrix of the algorithm gives a matrix of correlation that shows the maximum of correlation for each guess from 0-255. When this max correlation vector for each byte is illustrated, a graph is obtained in resemblance

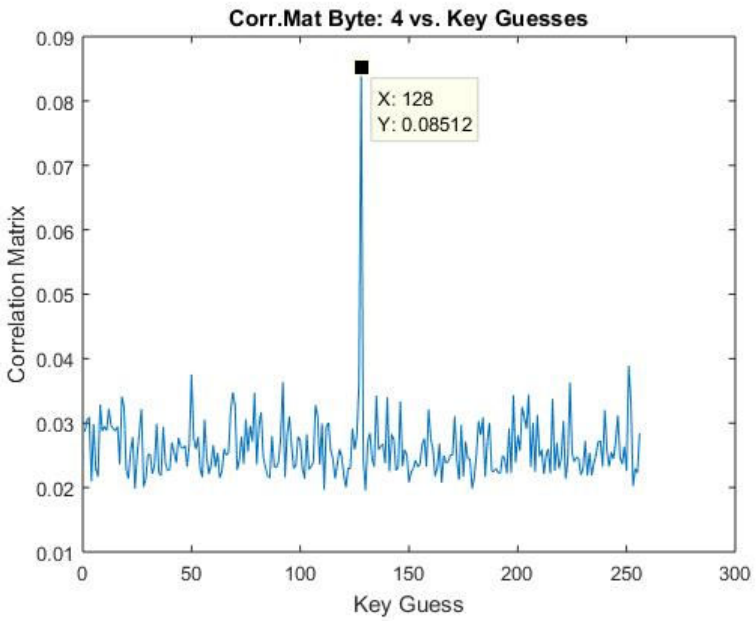


Figure 3.5: 15,000 Traces Max Correlation Vector for Byte 4

to Figure 3.5. As an example, byte 4 will be analyzed.

Clearly there is a spike in correlation of the normalized vector to the traces that is graphed in Figure 3.5. The spike has an index of 128, but since Matlab indexes from 1 instead of 0, the proper first byte of the 10th round sessional key is $(127)_{10}$ or $(7F)_{16}$. This matches the same key needed in Figure 3.4 to recover the master key's first byte '03'.

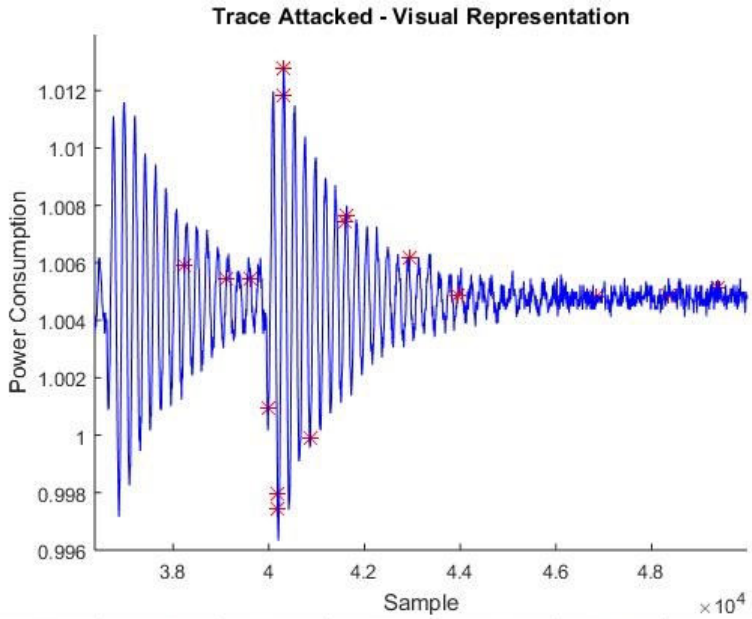
All bytes were broken using the same correlation matrices for their respective bytes and are all visually shown for 15,000 traces in *Appendix - 16-Byte Key DPA Results*. Of the 50,000 samples acquired, only the last 15,000 are used to free at least a quarter of memory in the computer during the DPA calculations - this greatly accelerates the attack.

The threshold of the amount of data needed to break the cipher was tested and it was determined that approximately 8,000 traces are required. For clarity, the data given is at 15,000 traces.

On a 3 GHz processor, it takes 50 minutes to obtain every set of 2,000 traces,

ciphertext data, and execute the DPA algorithm with the inverse sessional script. After all of the data is imported to the computer, the process takes approximately 7 hours to run. These are extremely noble results compared to the only other method to break AES-128 that takes billions of years.

The vulnerability analysis of where the power traces are susceptible to the DPA attack is visually shown below in Figure 3.6. This is done by referencing back to any power trace's sample with an offset with respect to the key found in the DPA algorithm.



1	2	3	4	5	6	7	8
40187	48339	41604	49376	40184	39992	42928	41617
9	10	11	12	13	14	15	16
40315	46840	39621	39113	40299	40864	38245	43943

Figure 3.6: Vulnerable Samples within a Power Trace

From the graph displayed above, the 9th and 10th round of the encryption shows the last 15,000 samples that were the samples being attacked. The 1-16 index at the bottom of the figure show the 16-byte key found and where exactly the trace had a very high correlation with respect to the output data. The red stars on the graph correspond to the sample values from 35,000-50,000, the samples of AES-128 in it's final round.

3.3.10 Hardware Solutions Against DPA

Public and private-key systems in modern day have large risks to mitigate with very few hardware implemented cost effective solutions. Examples of these solutions can be categorized in two types. They are creating new logic families or implementing an external circuit to work as a voltage-current buffer for the encrypting core eliminating the sensitive side channel on the system-level entirely.

Creating new logic families [27–29] such as MOS Current Mode Logic, Sense Amplifier Based Logic, or Wave Dynamic Differential Logic, for encryption cores is unrealistic and non-efficient since every logical component of the chip would need to be re-designed and calibrated accordingly. The high silicon area and power overhead required for these methods do not justify the implementation cost of replacing all gates in the hardware realization.

External circuits are a sensible solution but they carry the burden of a large power consumption and have heavily bottlenecked throughput restraints in modern systems that need to be achieved.

In literature one of the most recognized and cited circuit is [30], a three-stage switched capacitor current equalizer. Overbearing drawbacks of this circuit is that it has a +44% power overhead and −100% degradation throughput efficiency. This popular circuit does protect against a DPA over 10×10^6 power traces, but it compromises strict performance standards that need to be met in any hardware accelerator. Even an application specific integrated circuit (ASIC) on-board solution on the same encryption die struggle give the reliable results.

The risks against application specific hardware solutions seem to be endless while more threats arise and hardware solutions generally cannot deliver results. In the last chapter, other potential hardware-software solutions are proposed. The scope of this work will enable future works in realizable solutions against present and future threats. Understanding the inner workings of the specific algorithm, in this case AES-128, along with the way the ASIC or FPGA execute operations

are the most important preliminaries to patch this vulnerabilities to prevent any associated threats.

3.4 Related Applications

After understanding this attack on a complex symmetric cipher, it seems daunting to yield transferable skills however, it is not since the same rules apply. Whatever part of an algorithm that can release sensitive data in which any of the operations at the desired state consume unique power can be broken using the same general method. The method is as follows.

Generalized DPA Attack

1. Determine the closest exterior state, data in or out and byte-to-byte independence.
2. Determine the desired interior state to be attacked.
3. Generate hypothetical values and or keys.
4. Calculate hamming distance from exterior state to interior state.
5. Calculate the correlation between hamming distance matrix versus the output power consumption.
6. Acquire interior state information.

In the case of a ECC SPM multiplication, the process requires knowledge of the present SPM algorithm being executed. As an example to analyze potential threats, the Double-and-Add method, the founding SPM is shown in Algorithm 1.

When looking at the main operations of this loop, which is completely key dependant, there are only two operations which dictate the final result on register R_2. Line 4 leaks a large amount of power since there is a point addition operation stating that the current state of the binary key string is a 1. While if the key's index bit is a 0, point doubling, occurring on Line 6, will always consumes less power

Algorithm 1 Double-and-Add Scalar Multiplication

Input: Point $P \in E, k = (k_{h-1}k_{h-2}...k_1k_0)_2,\ k_i \in [0,1]$,

Output: $kP \in E$

```
R_1 = P;   R_2 = 0;
for i = 0 to h − 1 do
    if k_i = 1 then
        R_2 = R_1 + R_2;
    else
        R_1 = 2R_1;
    end if
end for
The final value is R_2 = kP
```

compared with its counterpart. This may seem trivial, but after understanding that point operations implemented in hardware consume distinctive amounts of power, the volatile results become highly evident [25].

Chapter 4

Secure High-Level Architecture

The high-level design of any ECC processor determines whether or not it is vulnerable to various SCAs. Though they all have risks, using proper techniques to lower power consumption differences such as different coordinates over a finite plane and a protected, highly regular SPM can ensure the safety of the unsuspecting hardware.

In this chapter the point operations, proposed inverse operation, and scalar point algorithms will be examined. A literature review is also completed to establish the most efficient designs that attempts to secure their respective architectures.

4.1 Previous Research & Review

There are numerous architectures of accelerated cryptographic processors for many different applications. The usual top figures of merit include clock speed (MHz), area, speed of SPM (s), small countermeasures against SCAs, and optimized low-level multiplication and inversion operations. Typically the combination of efficient algorithms and a well organized architecture present the best solutions for their individual objectives.

The finite field layer of the hardware is the most influential decision in the entire design [5]. This is due to the fact that the squaring and the inversion operations within the layer require both major aspect of the multiplier, the multiplication

and reduction stages. Table 4.1 shows the recent, most prominent processors.

These processors from [5] are controlled by a finite state machine (FSM) or a micro-programmable controller (MCC). Implementing a state driven design on all levels is necessary. Next, FPGAs are the hardware platform of choice due to the reconfigurability, modularity, and testing purposes to increase or decrease key lengths. Clearly the binary polynomial basis fields are the most popular from effortless transition to hardware. Key sizes range from 163-571 bits - the most popular is 233-bit. The product of choosing this key is that the primitive trinomial simplifies multiplication based operations at the finite field layer of the architecture and will be further discussed in the next chapter, Low-Level Multiplier Implementation.

An inversion in the finite field layer is not performance hindering arithmetic if the optimized coordinates for hardware are used. The coordinate system that is the most popular is the projective coordinate system as seen in the above review, specifically López-Dehab coordinates [5, 8, 11, 12]. When employing projective coordinates, it replaces all inversions within the point operations with added multiplications over the new three dimensional plane. Therefore, if projective coordinates are employed, the field inversion is only computed once after the SPM is completed - this is so that the calculation can be realized in the original two dimensional plane.

If Affine coordinates are used, the inversion operation is an extremely costly low-level operation; the inversion when $m \geq 128$ requires approximately 7 multipliers [39]. Below in Table 4.2, the amount of hardware operations needed to compute the large point operation in $GF(2^m)$ [39].

Table 4.1: Keynote ECC Processors in Literature

Ref.	Platform	Control	Basis	Bit	Clk (MHz)	Area (Slices)	SM(s)	SM	Coordinate	Mult./Inv.	Protected [5]
[8]	XCX2V6000	FSM	Bin. Poly.	163	93.3	16188	34.11	López-Dehab	LD-Proj.	MSD	SPA & Timing
[9]	XC2V80004	-	Bin. Poly.	233	62.5	15365	7.2	Montgomery	Projective	Karatsuba mult.	SPA & Timing
[10]	XCX5VLX50	FSM	Bin. Poly.	233	93.3	3073	-	Binary method	Affine	R2L Shift mult.	-
[11]	XC4VFX100	FSM	Bin. Poly.	571	93.3	12894	224	Montgomery	LD-Proj.	Interleaved mult.	-
[12]	Altera Stratix II	FSM/MCC	Binary Poly.	163	163	14280	11.71	R2L, L2R NAF	LD-Proj.	Itoh-Tsujii inv.	SPA & Timing
[13]	Virtex-4	FSM	Bin. Poly.	163	100	3528	1070	Binary method	Affine	Interleaved mult.	SPA & Timing
[14]	Virtex-6	MCC	Prime	256	60	20.8k	6.1	R2L, L2R NAF	Affine & Proj.	Interleaved mult.	SPA & Timing & Fault
[15]	Balsa	FSM	Bin. Poly.	233	-	$0.8025mm^2$	919	Montgomery	Affine	Karatsuba mult.	SPA & Timing

Table 4.2: Hardware Costs of Point Addition

Hardware Operation	Affine (x,y)	Projective (X,Y,Z)
Inversion	$2\,log\,k + 1$	1
Multiplication	$2\,log\,k + 4$	$6\,log\,k + 10$
Cost: m $\geq 128\,(I : M)$	1:624	1:241

At last, the SPM and security will be reviewed. The scalar point multiplication's speed is calculated on how long a design takes to complete a single m-bit SPM with respective to the frequency of the clock.

Among the listed designs, the Binary Recoding Method reduces the number of point additions recoding highest degree of polynomial $a(x) = a_{m-1}x^{m-1} + ... + a_1x + a_0, x_i \in GF(2)$ [10, 17]. The Right-to-Left & Left-to-Right Non-Adjacent Form (NAF) further reduces point additions with precomputed LUTs in memory [12, 14]. Both are computationally faster designs compared to the Double-and-Add (Algorithm 1) by reducing the amount of point additions by adding a single point double operation. The NAF form algorithm is a derivative of the Recoding method, sharing the same SPM qualities. Algorithm 1 and the Recoding method are equally unprotected against SCAs - these SPMs are never suitable when explicitly fighting SCAs.

The most attractive SPM is the Montgomery ladder method. Montgomery's algorithm is one of the fastest SM algorithm in practise due to the unique mathematical qualities it holds. This makes it the pinnacle of success for recent high-speed architectures. Although this algorithm simplifies SPMs due to it's highly-regular complexion, it is vastly susceptible to modern timing and differential power attacks [3, 35]. Though these reviewed designs state their resistance to certain SCAs, they are not secure against the previous chapter's DPA attacking method especially if paired with a timing or ZVP attack[1]. The importance of the SPM algorithm and point operations dictate the overall security of the processor.

[1]Since zero-value point attacks require the attacker to manipulate the base point pre-programmed in the hardware's memory making it unrealistic to test at this point in this research.

4.2 Secure Scalar Point Multiplication

Since it's discovery in 1987, P. L. Montgomery's ladder [6] has been the staple of hardware designs as the optimized and arguably, the leading dynamic SPM algorithm in ECC. What the ladder makes up for in pure computational speed and regularity, it lacks in immunity from contemporary SCAs.

In 2009, M. Joye proposed a m-ary generalization to the Montgomery ladder which would pave the way for a SCA resistant SPM algorithm [3].

This section will address the security of the two elite, left-to-right (L2R) SPM algorithms with respect to timing and differential power attacks. Practising accelerators initiate countless SPMs in a single ECC protocol and the biggest security vulnerability is the SPM gateway operation.

4.2.1 Montgomery's Algorithm

The high-speed Algorithm 2 is Montgomery's laddering method. It is heavily dynamic and being used as the leading SPM without question. It's vital invariant property $P = Y - X$ in every state leads to these keynote qualities. Montgomery's algorithm computes both (x, y) coordinates in any system i.e. affine or projective, only depending on present and previous x coordinates mathematically proven from [2]. Also, (x, y) coordinates of the next point on the curve can be computed in parallel giving the option of a semi-pipelined design as shown in [7,9].

The highly-regular essence of this popular SM has been thought to be secure because of its invariant states which protects it against simple power analysis and safe-error attacks [6]. The architecture of Montgomery's ladder is extensively explored in [7]. Advancements in malicious attacks against security cores make this algorithm no longer safe.

At first glance of the loop on Lines 2-8, one can see regularity in both states. However, output buses from registers X and Y carry different operations on both

Algorithm 2 Montgomery Scalar Multiplication [2]

Input: Point $P \in E, k = (k_{h-1}k_{h-2}...k_1k_0)_2, k_i \in [0,1], k_{h-1} = 1$

Output: $kP \in E$

1: **Int:** $X = P$; $Y = 2P$;
2: **for** $i = h-2$ down to 0 **do**
3: **if** $k_i = 1$ **then**
4: $X = X + Y$; $Y = 2Y$;
5: **else**
6: $Y = X + Y$; $X = 2X$;
7: **end if**
8: **end for**
9: The final value is $X = kP$

states, Lines 4 & 6, enabling DPA the obvious measurement to release the loop characteristics.

These characteristics, in reference to *Generalized DPA Attack* last chapter, are susceptible to the correlation of generated hypothetical power with the actual output power of main registers X and Y. Since the targeted interior state is known, the HD needs to be calculated from the output data to the hypothetical scalars/keys once the algorithm finishes a single m-bit SPM. Below is a hypothetical situation to break a key establishment with a DPA attack.

Attack Against a Key Establishment

To find the public-private key (scalar k) during a key establishment, successive runs of 5-25 x 10^3 random scalars fed into the SPM system will calculate random output points. The HD between the output scalars, of the output points, and hypothetical power consumption will be calculated. This matrix will be correlated, Equation (2.11), to the output power consumption of the overall output bus and registers X and Y. Knowing the public base point P, the private key will be exposed.

The attacker needs to be aware that each bit of the scalar is dependant on

previously indexed bits therefore all bits need to be analyzed as one set of data which is the opposite compared to AES-128 where each byte is independent. DPA paired with a timing attack focused on output register activity would breakdown the movement of data from $X \leftrightarrow Y$ grounding any system using the Montgomery ladder to lose it's overall authenticity.

4.2.2 Joye's Algorithm with Hardware Design

Being a more secure byproduct of Montgomery's ladder, Joye's SPM, Algorithm 3, possesses many of the great qualities of Algorithm 2. Some of these qualities include regularity, high-speed, and low hardware cost. However it does not have the invariant quality of the ladder. The pertinent difference between them is that register X is active twice sequentially in every state of the SPM loop and there is a point addition correction on register X in the final step. This makes the algorithm regular, but not invariant.

Algorithm 3 Joye's Scalar Point Multiplication (L2R) [3]

Input: Point $P \in E, k = (k_{h-1}k_{h-2}...k_1k_0)_2, k_i \in [0,1]$,
$k_{h-1} = 1$
Output: $kP \in E$

1: **Int:** $X = (k_{h-2} + 1)P$; $Y = 2P$;
2: **for** $i = h - 3$ down to 0 **do**
3: **if** $k_i = 0$ **then**
4: $X = 2X$; $X = X + P$;
5: **else**
6: $X = 2X$; $X = X + Y$;
7: **end if**
8: **end for**
9: $X = X + P$;
10: The final value is $X = kP$

The fact that there is only one register that holds sensitive information makes it impossible to differentiate between state 1, Line 4 or state 2, Line 6. The 1-bit

k_i register, the ith index of the scalar k, dictates the internal and external power consumed by hardware just like Montgomery's ladder. The difference is that internal register X acts as the buffer to the external power consumption rendering a DPA attack obsolete.

If a timing attack were to be employed, it would not be able to characterize any difference from the k_i's behaviour since both sequential commands realized in hardware are blocking statements on the same register that result in identical activity compared to the next $i + 1$ loop index. Completed in a single clock cycle plus a minuscule logic delay, both state 1 and 2 are identical. If it was possible to deploy a fault resulting word or fake operation into the ALU to affect register X, the attacker again would not be able to predict whether the first or second blocking statement in either state 1 or 2 was executed with full certainty.

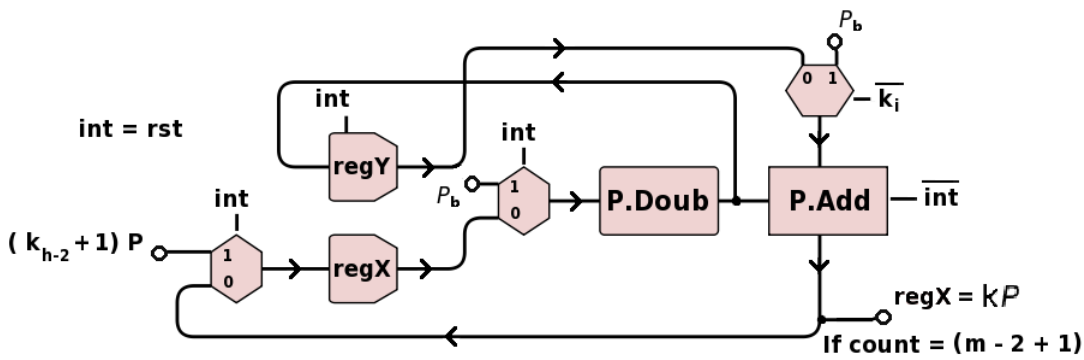


Figure 4.1: Joye's SPM Hardware Block Diagram

Figure 4.1 shows the hardware concept design of Joye's algorithm with minimal complexity to lower the area. The synchronous control unit includes a counter register (*count*), initialization signal/flag (*int*), and 1-bit k_i key register.

If $int = 1$ by a reset (rst) which resets the internal i^{th} index, the hardware will initialize the registers $X = (k_{h-2} + 1)P$ and $Y = 2P$.

In next clock cycle, $int = 0$ will enable the counter and *Pt.Add* where *count* will increment while it jumps between states 1 and 2 of Algorithm 3 on the positive

edge of the clock until reaching $(m - 2 + 1)$. After the final cycle concludes, the next clock will compute the correction on register X. Finally, the output will be available on the following clock cycle in register $X = kP$.

4.2.3 Comparison

The Table 4.3 below shows a comparison of the both Algorithm 2 and Algorithm 3 and how they negate the three major SCAs discussed.

Table 4.3: Comparison of SPM SCA Protection

Montgomery's Algorithm	**Joye's Algorithm**
Invariant-Regular	Regular
NOT resistant to Timing Attacks	Resistant to Timing Attacks
NOT resistant to C, M Safe-Error Attacks	Resistant to C, M Safe-Error Attacks
NOT resistant to Power Analysis	Resistant to Power Analysis

Evidently, Montgomery's algorithm has no resistance to the SCA attacks outlined while Joye's is fortified. There are other SCAs that both algorithms are not fully secure against, for example, M safe-error fault attacks [35]. Certainly Joye's algorithm is not as computationally fast as Montgomery's due to its underlying mathematics, but when the cryptographic cores main purpose is to maintain authenticity, Joye's would be more suitable in small applications requiring 128-bit security.

Ultimately, the security will be maintained at the highest level of operation achieved by the proposed hardware design modeling Joye's SPM. Establishing the rest of the hardware is imperative and will be the subsequent focus with the future goal of an all-programmable system-on-chip (SoC) FPGA implementation.

4.3 Optimized Point Operations

After studying Chapter 2 and point operations' algebra, a more profitable coordinate system can be established for progressive hardware implementations. For example, in the point doubling Equation (2.8) there are 2 multiplications and 1 inversion. This is extremely costly since an arithmetic inversion is the most expensive operation in any ECC ALU implementation [8]. Point doubling and addition Equations (2.8) (2.9) can be easily mapped to a more efficient plane to further improve the functionality of the low-level hardware's operability. All the following derivations can be found from [39].

Recalling point infinity has no distinctive Affine coordinates, point P is mapped to an existing projective plane such that,

$$P(x, y) = P(X, Y, Z),\ Z \neq 0 \in E \tag{4.1}$$

in which point infinity is defined as $\mathbb{O} = (1, 0, 0)$. Any arbitrary point $P(X, Y, Z)$ still carries the characteristics of $\mathbb{O} + P = P + \mathbb{O} = P$. In addition, $-P = (X_1, X_1 + Y_1, Z_1)$ is very similar to the Affine representation of $-P$.

This coordinate system needs to be applied to reduce the amount of finite field inversions discussed in Table 4.2. The most favoured type of projective coordinates is the López-Dehab (LD) representation where $(x, y) = (X/Z, Y/Z^2), Z \neq 0 \in E$ and preferably $Z_b = 1$, to simplify operations[2] [39]. Below, the forward conversion is shown by Equations (4.2) and the curve of Equation (2.5) is now mapped to (4.3) as follows:

$$\begin{aligned} Z_b &= 1 \\ X &= xZ \\ Y &= yZ^2 \end{aligned} \tag{4.2}$$

$$E_K : Y^2 + XYZ = X^3Z + aX^2Z^2 + Z^4,\ a \in E_K \tag{4.3}$$

Using LD-coordinates institutes the point doubling and addition equations to

[2] Z_b is the Z-coordinate of the base point P.

be mapped accordingly with $a = 1$ for an optimal cofactor. The following subsections will cover the hardware developed of the datapath in order to design the point operations appropriately.

The analysis of different projective coordinates are explored in [7]. The memory and power consumption of LD-coordinates are the lowest among the top projective systems reviewed in recent literature[3].

Just as in the synchronous control unit in Joye's algorithm, register *count* in point double and addition will increment every clock cycle to initiate the subsequent operations within datapath upon the *reset* signal. The focus will be on the datapath design rather than the control unit since the designs are parallelized compared to traditional serial designs [8, 12, 19].

4.3.1 Point Double with Datapath Schematic

Equations (2.8) are now mapped to it's LD form with a set of three equations. The resulting point $2P(X_3, Y_3, Z_3)$ requires squaring operations to replace the prior inversion operation. Equations (4.4) are as follows, where $X_3, Y_3, Z_3 \in E_K$.

$$\begin{aligned} Z_3 &= X_1^2 Z_1^2 \\ X_3 &= X_1^4 + Z_1^4 \\ Y_3 &= Z_1^4 Z_3 + X_3(Z_3 + Y_1^2 + Z_1^4) \end{aligned} \tag{4.4}$$

Below shows the schematic of the LD - point doubling circuit. It requires 3 multiplication (M_j), 5 squaring (S_j), and zero inversion operations within the curve's finite field. This design needs all 3 m-bit XOR gates along with all of the other operating blocks, S_j and M_j, to be independent. No more than a single operating block and one m-bit XOR will be computed under one clock cycle. The design shown in Figure 4.2.

[3]Jacobian, Standard, and Montgomery projective coordinates are the other top projective systems besides Lópex-Dehab.

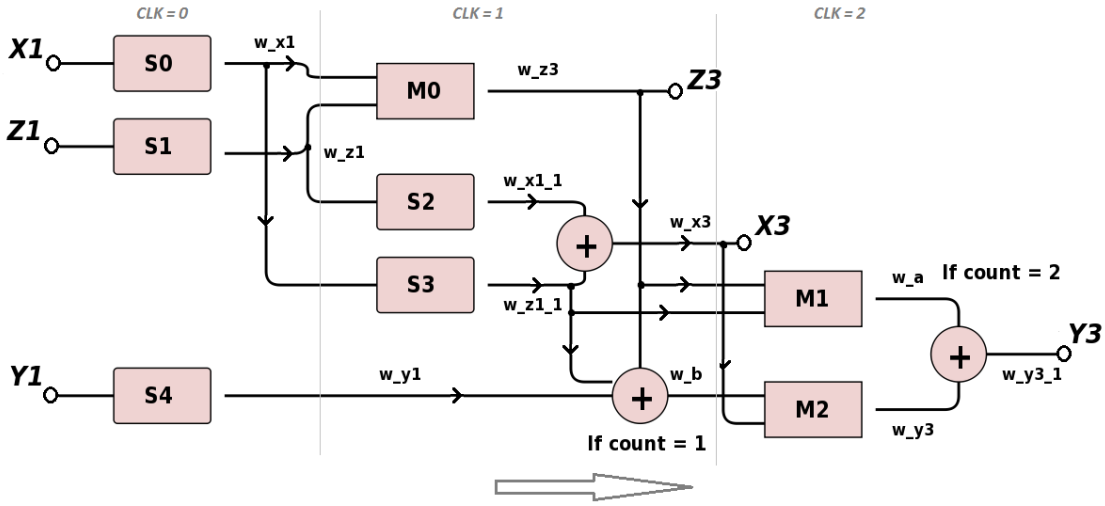


Figure 4.2: LD - Point Double Datapath Schematic

On the first clock cycle, this circuit will square all three inputs X_1, Y_1, Z_1 in parallel. On the next cycle, the outputs X_3 and Z_3 are computed and will be available on $count = 2$. Y_3 is available on following cycle as it is the last computation after X_3 and Z_3. From looking at Equations (4.4), the next point x, y coordinates X_3 and Y_3 depend on prior computations. The critical paths in Figure 4.2 start through the two squaring operations on X_1 or Z_1 that lead to the first XOR on the output of S_2 and S_3. From here, there are multiple paths that end at Y_3 requiring that same amount of logical delay resulting in a critical latency of 1 multiplication and 2 squaring operations.

4.3.2 Point Addition with Datapath Schematic

Lastly, Equations (2.9) are mapped to the projective plane using LD-coordinates. Equations (4.5) show the flow of operations within the set with intermediate registers A-G. The point addition of $P + Q = R(X_3, Y_3, Z_3)$ is as follows:

$$\begin{aligned} A &= Y_2 Z_1^2 + Y_1 & B &= X_2 Z_1 + X_1 \\ C &= Z_1 B & D &= B^2(C + Z_1^2) \\ Z_3 &= C^2 & E &= AC \\ X_3 &= A^2 + D + E & F &= X_3 + X_2 Z_3 \\ G &= (X_2 + Y_2) Z_3^2 & Y_3 &= (E + Z_3)F + G \end{aligned} \tag{4.5}$$

where the datapath's design is below. Intermediate registers A-G are labelled on the hardware corresponding with resulting output. In total, 8 multiplication, 5 squaring, and again, zero inversion operations are required to compute the point addition in this parellelized manner.

Requiring 5 clock cycles to complete, an improvement to the parallel design of 8 cycles in [39], this datapath has been broken into ten sections for increased speed and functionality.

In order, the output coordinates of next point R begin with Z_3 becoming available when $count = 2$. The X_3 is available after the following clock cycle when $count = 3$. The critical latency is dependant on the calculations of M_1, M_2, M_3, onward to the operation of X_3, and ending through M_7 and the last m-bit XOR. The Y_3 is available on the following cycle after $count = 4$. The bottleneck in this design is when computing X_3 during the $count =$ 0-2 cycles due to the three multiplication operations and no squaring. It is quite evident that the point addition operations consumes a more considerable amount of power compared to the point double since there is more than twice as many compulsory multiplication operations. The critical latency is 4 multiplication operations.

4.3.3 Review of Hardware

FPGA designs are crucial because of the dynamic nature of the designing process towards ECC processors. The designs must be robust to provide rapid prototyping to test different key lengths, curve coefficients, the finite field, mixed-coordinates[4], and finite field multipliers. Generally speaking, the more specific the application of the processor, the more efficient it can be.

Both proposed designs utilize a parallel architecture requiring a synchronous state machine controller which can provide the high-level datapath to have efficient

[4]Mixed coordinates are typically used to reduce the number of multiplications using Frobenius maps to lower the critical path latency, instead of using a single projective coordinate system during point operations [12].

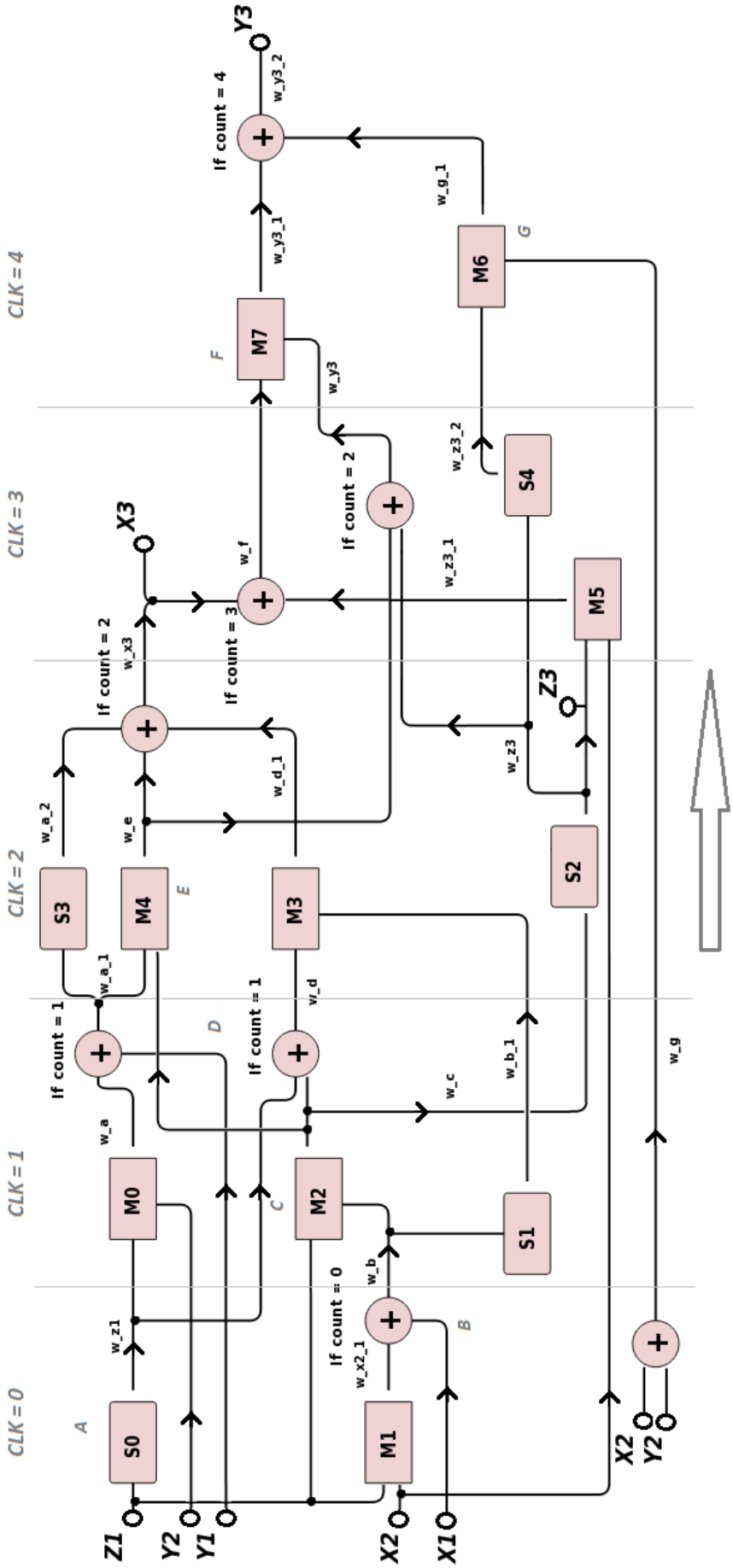


Figure 4.3: LD - Point Addition Datapath Schematic

results. No inversion operations need to be computed due to the LD-coordinates selected. This may seem minuscule, but when implemented on a NIST medium scale curve such at K-233 or K-283 [34], the amount of inversions become unrealistic to implement on an affordable FPGA due to speed and complexity restrictions. The HDL pseudo-code for the datapath designs can be found in *Appendix - Verilog HDL Pseudo Scripts*.

The reasoning for designing both operations with individual squaring and multiplication blocks is because the targeted Kintex-7 FPGA has plenty of space available. The serial multiplier implemented in the following chapter consumes less than 0.5% of area after synthesis and before place & route. What this high-level design lacks in area can be ignored due to a small serialized multiplier enabling the overall architecture to be tested at an average speed SPM.

4.4 Multiplicative Inverse

The isolated inverse operation within this architecture is the last operation computed after the entire SPM is completed. It is only computed once to act as the conversion from LD-coordinates back to Affine. Since the EC scalar point information to be used in ECC protocols resides on the two dimensional plane, the conversion is crucial. The backwards conversion is as follows:

$$\begin{aligned} x_Q &= \frac{X}{Z} \\ y_Q &= \frac{Y}{Z^2} \end{aligned} \tag{4.6}$$

where x_Q and y_Q are the Affine coordinates of the scalar multiplication point $Q = kP$.

To execute this backwards conversion, there are two exclusive inversions $\frac{1}{Z}$ and $\frac{1}{Z^2}$ since $(Z^{-1})^2 \; mod \; 2^m \neq (Z^{-1} \; mod \; 2^m)^2$. To accomplish this, the Extended Euclidean Algorithm (EEA) needs to be computed over $GF(2^m)$.

4.4.1 Binary Extended Euclidean Algorithm

The binary EEA is a simplified version of Euclid's algorithm to find coefficients x and y such that $ax + by = gcd(a, b)$. In the case of finding the multiplicative inverse for polynomials, $a = A(x)$ and $b = P(x)$ where $y = 1$, the inverse is found by solving the previous equation for x. Algorithm 4 can be found in [39] but the modified version below is developed for a logical transition to a hardware implementation.

The output of Algorithm 4 is $A^{-1}(x)$ and is found by these major steps. Within the inner *while loop*s, operand registers U and V are divided by x until they cannot be divided by a whole number, hence $mod\, x = 0$. Both loops can be developed in parallel hardware as long as the $(U, V \neq 1)$ condition is true. A serialized design is of greater benefit since the inversion is, again, only computed once and speed gain would be infinitesimal compared to the overall computational time. This solitary algorithm is the final step and the true output of any projective coordinate based SPM processor.

4.5 High-Level Summary

The most efficient high-level operations were discussed and broken down into related blocks of the progressing design.

Joye's algorithm proved to be superior to Montgomery's ladder from a SPM security aspect. The point doubling and addition were outlined using the LD-coordinates to create a customized datapath circuit for each point operation. Lastly, the ideal binary multiplicative inverse, the EEA, was explained and modified to better fit the use of the proposed architecture.

Algorithm 4 Extended Euclidean Algorithm in Hardware

Input: Primitive Poly. $P(x)$, Poly. $A(x) \in GF(2^m)$

Output: $A^{-1}\ mod\ P(x)$

1: **Int:** $U = A(x)$; $V = P(x)$; $G = 1$; $H = 0$;
2: **while** $(U, V \neq 1)$ **do**
3: **while** $(U \bmod x = 0)$ **do**
4: $U \leftarrow$ shiftRegRight(U);
5: **if** $(G \bmod x = 0)$ **then**
6: $G \leftarrow$ shiftRegRight(G);
7: **else**
8: $G \leftarrow$ shiftRegRight$(G \oplus P)$;
9: **end if**
10: **end while**
11: **while** $(V \bmod x = 0)$ **do**
12: $V =$ shiftRegRight(V);
13: **if** $(H \bmod x = 0)$ **then**
14: $H \leftarrow$ shiftRegRight(H);
15: **else**
16: $H \leftarrow$ shiftRegRight$(H \oplus P)$;
17: **end if**
18: **end while**
19: **if** $[deg(U) > deg(V)]$ **then**
20: $U \leftarrow U \oplus V$; $G \leftarrow G \oplus H$;
21: **else**
22: $V \leftarrow V \oplus U$; $H \leftarrow H \oplus G$;
23: **end if**
24: **end while**
25: **if** U=1 **then**
26: **Output** $\leftarrow G$;
27: **else**
28: **Output** $\leftarrow H$;
29: **end if**

Chapter 5

Low-Level Multiplier Implementation

In this chapter, the implementation of 233 & 283-bit comparable finite field multipliers is presented. As the low-level operations, they are one of the most essential building blocks towards the efficiency of the ECC processor. Surrounded by many options of multipliers for various applications, the classic parallel and a popular serial multiplier will be implemented. A special case of the parallel multiplier will lead to the development of a squaring operator.

The sections will begin with an introduction of choosing the correct key size for relatable applications, dealing with strict throughput constraints, and contrasting both multipliers with their respective synthesis results. The overall SPM design proposed will be outlined to highlight the interconnected parts of the entire hardware design.

5.1 Finite Field Multiplier

Recalling from Table 4.1, the key size ranges from 163-571 bits where the most popular is 233 bits. Recent designs [9, 10, 15] express the large computational advantage of choosing a 233-bit key with a polynomial basis providing the primitive trinomial of $t^{233} + t^{74} + 1$. This trinomial provides a great level of symmetric security while reducing the complexity of the circuit and increasing the speed of any

operation using the field multiplier [14].

The area is reduced from having a single middle element, t^{74}, rather than the NIST approved pentanomial, $t^{283} + t^{12} + t^7 + t^5 + 1$ [34] by over 30% on larger FPGA designs. The use of a pentanomial is the only way to increase key sizes with a polynomial basis to the 283-bit standard level while the 233-bit key has benefits geared towards performance upgrades.

Since the overall goal is to attack the entire proposed architecture, the multiplier serves purposes on a smaller scale which contradicts what the architectures of [5] suggest. Established algorithms that provide speed and area should be further explored for their appropriate merits [39] in different devices, but are presently not scrutinized. From Table 5.1, the two types of multipliers selected can be seen to have very different qualities.

Table 5.1: Parallel vs. Serial Finite Field Multiplier

Multiplier	Field	Clock Cycles	Circuit Complexity (Gates)
Parallel	$GF(2^m)$	1	m^2 ANDs, $m^2 - 1$ XORs
Serial	$GF(2^m)$	m	2 m-bit regs, m ANDs, $m + 1$ XORs

Serial multipliers such as Interleaved [13], Karatsuba [9], and Montgomery sequential methods all accomplish the field multiplication within $[m/32, m]$ clock cycles. This range consists of small increments when processing multiples of 32-bit data streams [40]. While the Interleaved and Karatsuba multiplier are slightly faster than Montgomery's smaller multiplier, shown in Table 4.1, all three with respect have relatively the same performance to size ratio making them all a viable choice.

The following designs will be achieved on a Kintex-7 FPGA having 218,600 LUTs, 437,200 flip-flop registers, and 350,000 programmable logic cells. The Zynq-7000 AP SoC (XC7Z045) embodies this FPGA for seamless data transfer to and from the ECC processing circuit. The on-chip 32-bit Cortex-A9 ARM CPU will require a throughput controller as there are a fixed amount of bonded I/O pins on

the FPGA. Since ECC uses key sizes well over 128 bits, a first in first out (FIFO) controller has been made to mitigate input data traffic.

5.1.1 FIFO for Large Keys

A FIFO is a digital circuit that buffers large input data onto a register stack into equal length words, in this case 32-bit words, to output the words in the sequential order that they were addressed. Specifically, this synchronous 32-bit FIFO is the throughput solution that can be used as the top module for both 233 & 283-bit keys. The FIFO controller and 32-bit buffering registers are shown below in Figure 5.1.

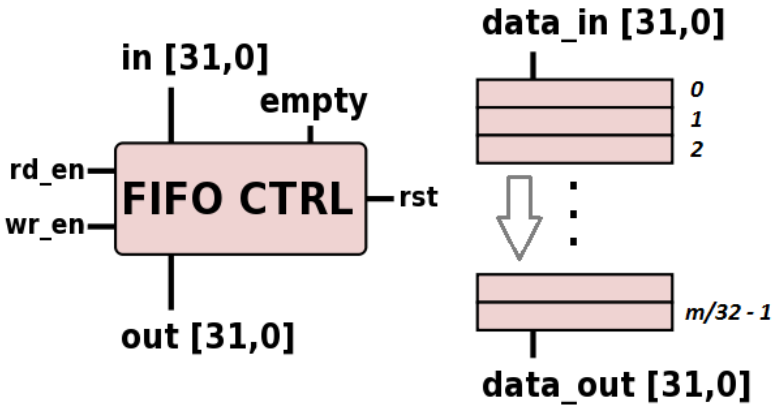


Figure 5.1: 32-bit FIFO Schematic

Having a depth of 8 and 9 bits respectively, the design can be modified by updating the internal counter to accommodate either bit size. This circuit after synthesis consumes 38 LUTs as logic and 43 registers as flip-flops - this circuit is $\leq 0.2\%$ of the total slices making it infinitesimal. The HDL code found for this design can be found in *Appendix - Verilog HDL Scripts: 32-bit FIFO*.

5.1.2 Parallel Multiplication and Squaring

The most primary design for multiplication is the classical parallel in and out design. It consists of a multiplication and reduction stage. The 233 & 283-bit multiplying module (top) has an output of $2^m - 2$ bits that will feed into the final reduction module (bottom) that reduces the polynomial to the original 233 & 283

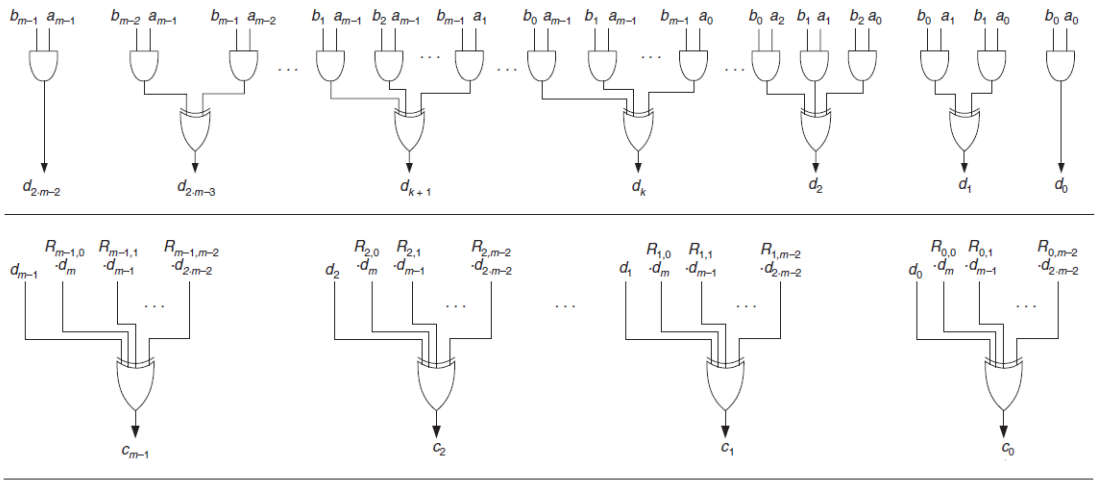


Figure 5.2: Parallel m-bit Multiplier [40]

bits as seen in Figure 5.2 - the circuit diagram shows the hardware required.

Both stages of the overall parallel multiplier will be tested with both key sizes. Having the same structure with larger word lengths will show to be a drastic increase in area. The 233-bit multiplication and reduction circuit after synthesis consumes 127,948 LUTs as logic and 698 registers as flip-flops. This is too large for any FPGA that has high scale modules since it consumes 58.53% of the total slices. The 283-bit multiplier after synthesis consumes 187,235 LUTs as logic and 848 registers as flip-flops. This would consume 85.65% of the total slices which leaves no room for any other operations. This design does not utilize any gates more than once hence sacrificing a lot of area to complete the entire operation within one clock cycle. A more suitable multiplier to be attacked by SCAs would be a serial based design to cut down on the dramatic area consumption.

233-bit Squaring Module

A useful piece of hardware that can be extracted from the 233-bit multiplier is the reduction module for a squaring operator. The parallel squaring circuit is developed by replacing the multiplying module with a smaller, concurrent assignment datapath module. The script can be found in the blocking statement shown below.

```
// Squaring Module
module classic_polySquare(
input [232:0] a,
input [232:0] f_x,
input clk,
output [232:0] z);

    integer i;
    reg [2*233-2:0] d;
// Polynomial Squaring
    always @ (posedge clk)
    begin
        d[0] <= a[0];
        for (i = 1; i <= 232; i = i + 1)
        begin
            d[2*i-1] = 0;
            d[2*i] = a[i];
        end
    end
// Polynomial Reduction
    poly_reduc a1 (d,f_x,clk,z);

endmodule
```

After synthesis the squaring circuit uses 79,343 LUTs as logic, 0 registers as flip-flops consuming 36.3% of the total slices. Even though this percentage is still very high for a small portion of the processor, a modified serial version of the previous point operation circuits could utilize this single cycle squarer for secure high-speed device.

A note on the reduction module is that the primitive polynomial will not need to exceed $m+1$ bits since the highest m-bit polynomial degree always exists hence reducing the size of register for the primitive string.

5.1.3 Montgomery Multiplication and Reduction

The sequential Montgomery multiplier over $GF(2^m)$ is modelled by the following Equation 5.1 where $A, B, R \in GF(2^m)$ [40].

$$C(x) = A(x)B(x)R^{-1} \, mod \, f(x) \tag{5.1}$$

The element R needs to satisfy $\gcd(R, f(x)) = 1$ which is always true in $GF(2^m)$. String R needs to be cleverly selected to reduce the amount of inversions and should be derived from the primitive polynomial $f(x) = x^m + a_{m-1}x^{m-1} + ... + a_1x + a_0$.

If R is selected to model the monomial $R = x^m$, then $R = a_{m-1}, ..., a_1, a_0$. A multiplication by x^m is accomplished by shifting the string left by m bits; therefore an inversion of this monomial is simply a shift to the left by m bits. This multiplier can be easily and efficiently implemented in hardware by using the following Algorithm 5; this algorithm is based on the proposed method in [40] but is translated into hardware operations. The m bits of shifting mean that a single Montgomery multiplication will take exactly m cycles.

Algorithm 5 Binary Montgomery Multiplication in Hardware

Input: $A(x), B(x), f(x) \in GF(2^m)$

Output: $C(x) = A(x)B(x)x^{-m} \, mod \, f(x)$

1: **Int:** $C \leftarrow 0$;

2: $C_0 \leftarrow C[0] \oplus A_iB[0]$;

3: **for** $i = 0$ to $m - 1$ **do**

4: $C = C(x) \oplus A_iB(x)$;

5: $C = C(x) \oplus C[0]f(x)$;

6: $C = shiftRegRight(C)$;

7: **end for**

The algorithm above will loop for m cycles to flag, with it's control unit, when the correct result is available on register C. Line 4 shows that register C is XORd with the register B if the 1-bit register A_i is true. Line 5 shows that if the previous

index $C[0]$ before the loop is true, then the value of register C is XORd with the $(m-1)$-bit primitive polynomial $f(x)$. The last step of the loop on Line 6 shows a single shift right by 1 bits representing the division of register C by monomial x. The script in Verilog HDL is shown below that models the datapath of Algorithm 5 - the remaining code that is the control unit can be seen in *Appendix - Verilog HDL Scripts: Serial Montgomery Multiplier*.

```
// Datapath Module
module mont_datapath(
input [232:0] c, b,
input a_i,
input [232:0] f_x,
output reg [232:0] new_c);

    reg previous_c0;
    integer i;
    always @ (*)
    begin
        previous_c0 <= c[0] ^ (a_i & b[0]);

        for (i = 1; i <= 232; i = i + 1)
            new_c[i-1] <= c[i] ^ (a_i & b[i]) ^ (f_x[i] &
                previous_c0);
        end
        new_c[232] <= previous_c0;
    end
endmodule
```

The multiplier circuit is comprised of the small datapath circuit scripted above and a FSM controller. A register transfer level (RTL) design of the entire 233-bit control unit surrounding the datapath is below in Figure 5.3. The RTL schematic of the multiplier is mostly made of the control unit due to small datapath as described in Algorithm 5 and highlighted below.

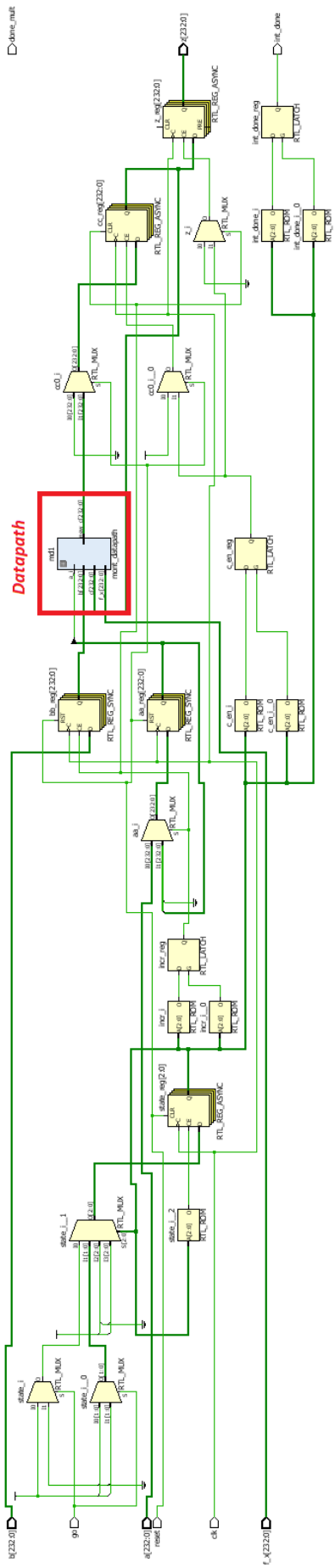


Figure 5.3: 233-bit Montgomery Multiplier RTL Schematic

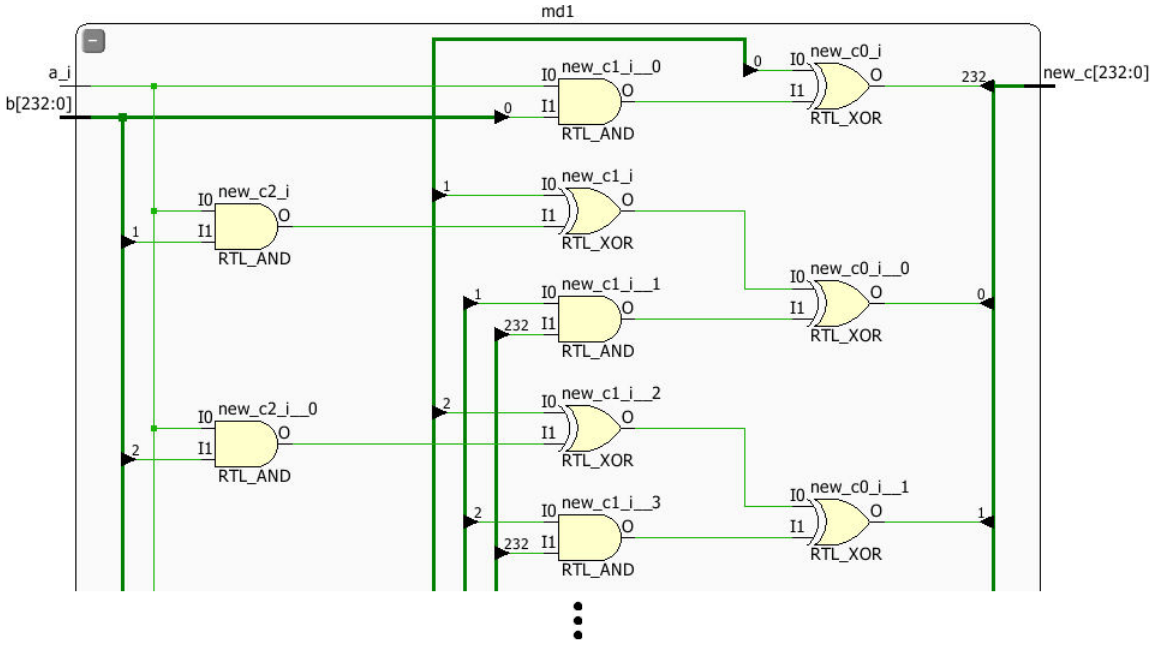


Figure 5.4: 233-bit Montgomery Multiplier RTL Datapath

A fragment of the 233-bit datapath is shown in Figure 5.4. In the middle of the figure it is visible that the AND operation is computed on register A_i and $B[i]$ before XORing the result with the previous $C[0]$ AND $f(x)$ to get the respective register $C[i]$, in the script as new_c[i], as previously explained.

The 233-bit Montgomery multiplication with integrated reduction after synthesis consumes 475 LUTs as logic and 935 registers as flip-flops. This is an amazing result and is as expected from Table 5.1. There is a 58.1% reduction of area complexity since it only is 0.43% of the total slices. Similarly, the 283-bit multiplier only consumes 576 LUTs as logic and 1135 registers as flip-flops with a combined 0.52% of the total area. Respectively, the multipliers take 233 and 283 cycles to complete.

5.2 Summary of the Connected System

The following subsections will examine the multipliers implemented and review the comprehensive architecture.

5.2.1 Multiplier Comparison

Prior to the place-and-route implementation phase in the design, the synthesis results are listed below. All of the designs are synthesized separately, therefore there will be small reductions in sizing after the place-and-routing stage is completed in the future.

Table 5.2: Post Synthesis Multiplier Results on Kintex-7

Multiplier	Control Unit	Area of Hardware (LUT, Reg)	I/O
233-bit Parallel	-	58.53% , 0.16%	257.73%
283-bit Parallel	-	85.65% , 0.19%	312.98%
233-bit Parallel Sq.	-	36.3% , 0%	193.09%
233-bit Montgomery	FSM	0.43% , 0.21%	258.56%
283-bit Montgomery	FSM	0.52% , 0.26%	313.81%

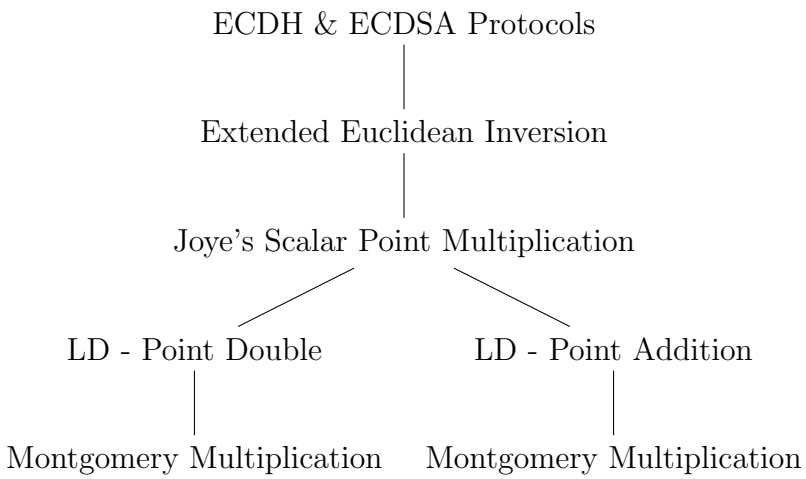
The choice of the two operand multiplier from both compared designs is the binary Montgomery multiplier. In terms of a speed-area ratio and future works of the proposed design, Montgomery's design is far superior. As for the squaring design, Montgomery offers a serialized squaring method that builds off of the pre-existing multiplier which may be more efficient in larger scale devices compared to the parallel design shown above.

These specific multiplier are restricted due to a maximum of 32-bit I/O. Consequently, they need to be serial in serial out (SISO) multipliers since data flow is serially fed into and out of the hardware module. From Table 5.2 it is striking that all designs practically use double the amount of the bonded I/O that the FPGA can physically provide. This is because the FIFO was not wrapped over the highest level module during simulation - the I/O percentage is listed to show the dire need of a top level FIFO module.

A comparison of the singular Montgomery multiplier versus recent serialized multipliers is provided in *Appendix - Verilog HDL Scripts: Serialized Montgomery Multiplier Comparison.*

5.2.2 Overview of Architecture

The novelty of this design revolves around using Joye's algorithm for SPM. To the best of the author's knowledge there are no published FPGA designs that readily use this secure SPM to protect against SCA threats. Below is a tree diagram of the hierarchy throughout the past two chapters and as it falls down to the base finite field multiplier.



Previously, the finite field inversion operation was needed at the lowest level alongside the multiplier, but after using the LD-coordinate, the inversion was the final conversion step of the SPM algorithm. The tree above completes the novel secure ECC SPM processor and solidifies the end of the resulting work.

Chapter 6

Conclusions

In the final chapter, a discussion of the contributions is made along with many areas that should be explored with the aid of this research. These areas will include the development a software platform for a validation of the cryptosystems, an analysis for masking, and hiding indirect data leakages in various cryptographic schemes.

6.1 Summary of Contributions

List of Contributions

1. A platform to develop and test SCAs against a wide range of cryptosystems
2. A successful DPA SCA against AES-128 to show the susceptibilities and weaknesses of cryptographic hardware from multiple aspects and propose an approach to attack other systems
3. Propose a secure, robust, and small scale ECC SPM architecture resistant to modern SCAs
4. A hardware design of a parallel K-233 & K-283 point doubling and addition datapath in López-Dehab coordinates
5. A tested and synthesized hardware design of a dynamic 32-bit FIFO
6. A tested and synthesized hardware design of a 233-bit squaring module for large scale devices

7. A tested and synthesized hardware design of both a parallel and serial multiplier over $GF(2^{233})$ & $GF(2^{283})$

6.2 Future Work

The future work for this research has a number of areas that will highly impact the domain of embedded security.

6.2.1 Hardware Design

Most importantly, the remaining pieces of the hardware design and implementation need to be completed. The design of the extended euclidean algorithm is not new work but is an essential step in recovering useful information from the three dimensional place of projected coordinates. Secondly, the control unit of the point doubling and addition operations needs to be created - this design will be small since they are parallel by design. Further development of a SIPO point operation design would also be needed if the overall implementation does not fit on the desired FPGA. Concluding the hardware designs, a full implementation of Joye's algorithm needs to be developed from the proposed design to compare SCA resistance with it's counterpart, Montgomery's SPM algorithm. Once the SPM algorithms are implemented in hardware, the SCAs can proceed. There will be adjustments of the global clock when the hardware design reaches it's phyical limits and will probably range from 50 to 500 MHz. These designs are highly recommended to be implemented on an existing SoC such as, but not limited to, the Zynq-7000. It is highly attractive due to the seamless software integration of the on-chip ARM processor using pre-existing wrapping software from Xilinx Vivado Design Suite to intercommunicate the FPGA design with the ARM processor.

Note: The acclaimed HDL to be written in the future in most cases cannot be written manually. The original HDL is written in *Appendix - Verilog HDL Scripts*, but has very large *for loop*s which can put a large strain on the design suite's compiler. The solution to this is to unroll every loop in the HDL code to be written from an automated C script that populates every output register from

the multiplying module by creating *.v* files from a file pointer in a *.c* file. An example of this technique is in *Appendix - C Scripts - Convert to Verilog Script* and is highly recommend to be used when writing any HDL for cryptographic purposes.

If once the FPGA designs are fully-functional and can perform a SPM in comparable time to the recent literature of [5], then it would be interesting to create an application specific integrated circuit (ASIC) in either complementary metal-oxide-semiconductor (CMOS) 0.18μm or 65nm technology. Another pathway would be to integrate the ASIC design with an existing micro-controller to test the circuit once fabricated for further exploration within accelerated cryptography.

6.2.2 Software-Hardware Integration Against SCAs

An entire customized cryptographic library is needed to test all possible input over all keys desired with vectors $\geq 2^m$ of the data inputted. Unlike other common software suites such as OpenSSL, the library needs to have the low-level access to change existing multiplying algorithm and scalar point multiplication in an automated fashion to verify hardware results correctly. Alternatively, a potential avenue of research could also include the capabilities to connect the proposed hardware system with the common all-programmable interface of the OpenSSL library.

A software solution of the finite field inversion technique may be more efficient to compute it sequentially after the SPM circuit has results. This could cut the time needed to perform a successful SPM since hardware design can be time consuming.

6.2.3 Masking to Prevent CPA Attacks

One method of hiding sensitive power traces of cryptosystems on a reconfigurable device involves randomizing the inner cipher or high-level process in order to ruin the relationship between how each trace is executed. This is done by performing a random amount of meaningless operations before, during, and after the targeted process. However, there is a couple issues when implementing this method to

protect AES-128 specifically. The first issue is that the AES-128 hardware design is combinational so an entire round of AES-128 is performed in one clock cycle and these fake operations can only be inserted in between rounds. This makes it slightly easier for the hacker to realign the waveforms and re-establish the correct overlap of traces. The second issue is that these fake operations greatly affect the throughput of the system, hence only a finite amount of these operations can be performed.

Another method is to make the power consumption random or equalize throughout traffic throughout the targeted process. Firstly, increasing the noise of the system can be done for randomizing the power consumption. To accomplish this, one would need to run multiple random operations simultaneously. A disadvantage to this sort of modification is that there is no such thing as a truly random number generator in hardware therefore a hacker can still find patterns within the system. Making the power consumption equal at each state within cryptosystems' processes is essentially the only sure way to mask a key or other important information against DPA while maintaining the throughput of the original implementation. This could be accomplished with a optimal switched-capacitor design similar to [30], mentioned in chapter 3, and implemented in CMOS technology

A conceptual but interesting technique to prevent any CPA attack against AES-128 could be created from interconnecting byte-to-byte dependencies with the Add-Round Key operation. This would make the algorithm slower because of the theoretically added serial computations in the Add-Round Key generator, but it would make DPA useless therefore greatly increasing it's security.

In the future when designing an FPGA implementation to protect AES-128, it is crucial to select pins that are located on the same type of I/O bank as the ciphers output pins - ideally the same pins. This ensures the same amount of power is being used to invert a fake output as the AES-128 output and hence, establishing equalized power consumption. The bus changes must occur on the exact same clock edge so that the power is consumed on the same state. These two features

are the reason that a masked implementation of AES-128 is difficult to implement on the SASEBO-GIII board specifically. This board utilizes a 1.5 V I/O bank to transfer AES-128 data and does not have enough output pins available to execute the mask.

Appendix A

DPA Data & Results

A.1 Power Trace to be Attacked

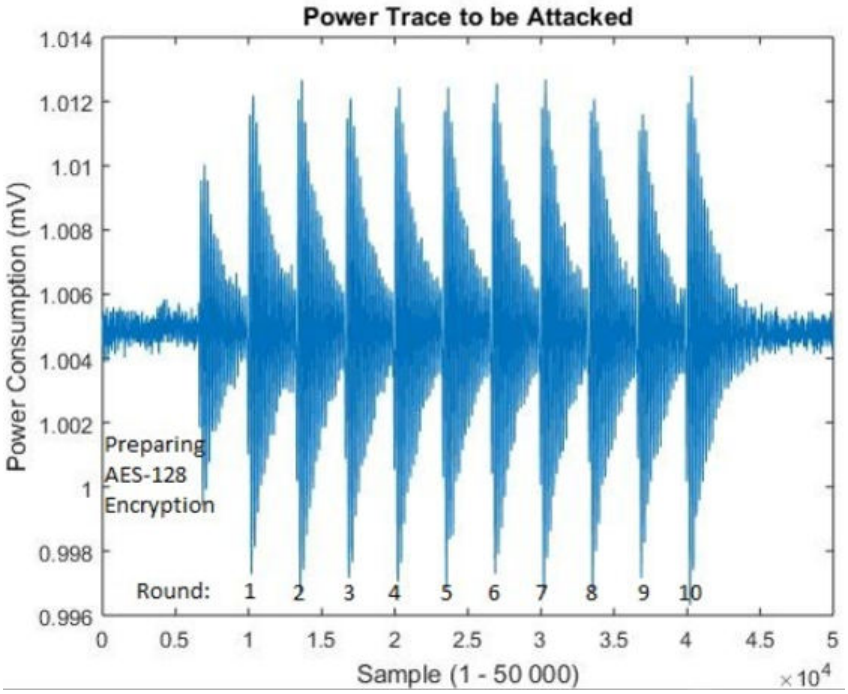


Figure A.1: Example of a Power Trace Captured at 50,0000 Samples

A.2 16-Byte Key Results

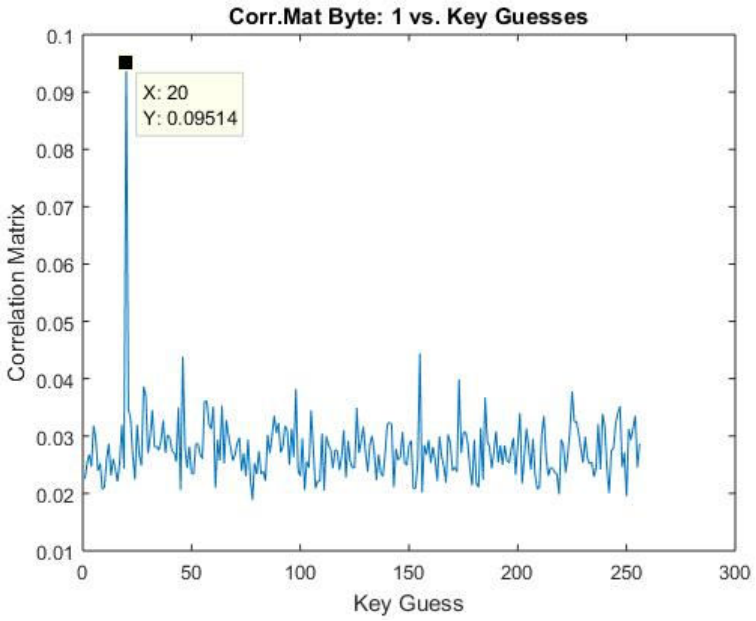


Figure A.2: 15,000 Traces Max Correlation Vector for Byte 1

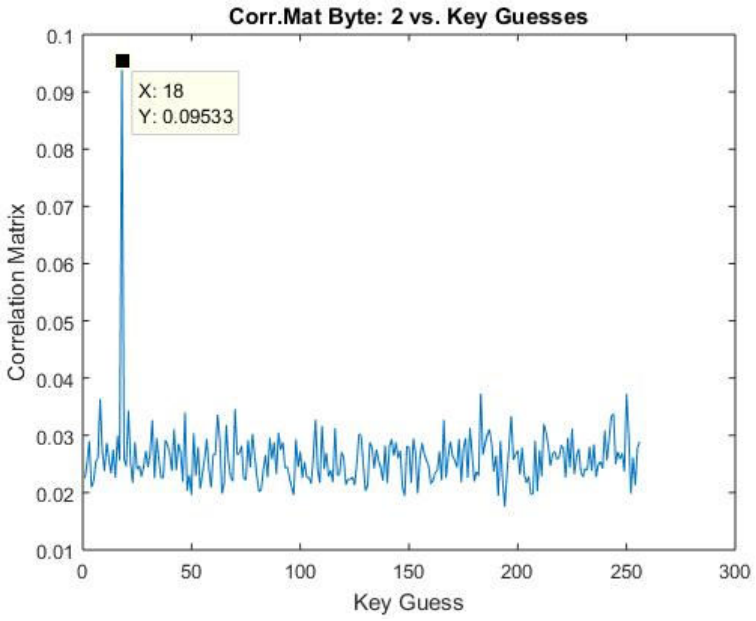


Figure A.3: 15,000 Traces Max Correlation Vector for Byte 2

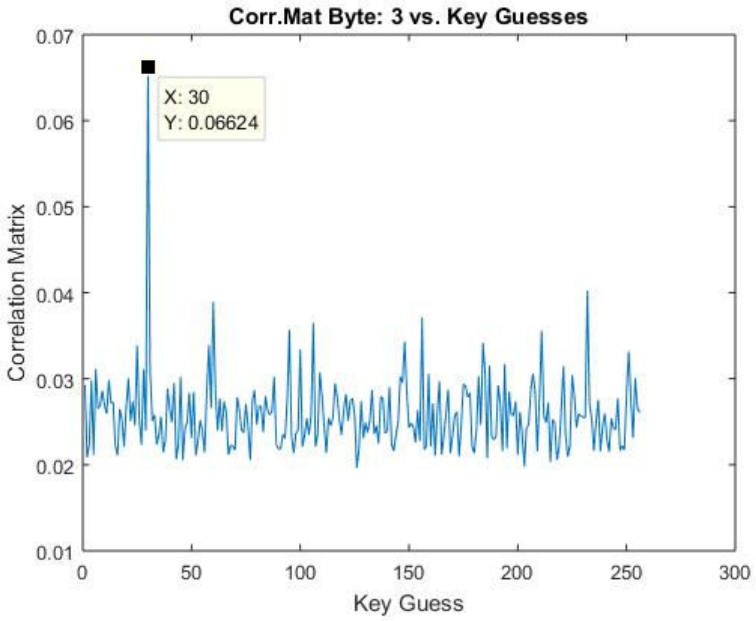


Figure A.4: 15,000 Traces Max Correlation Vector for Byte 3

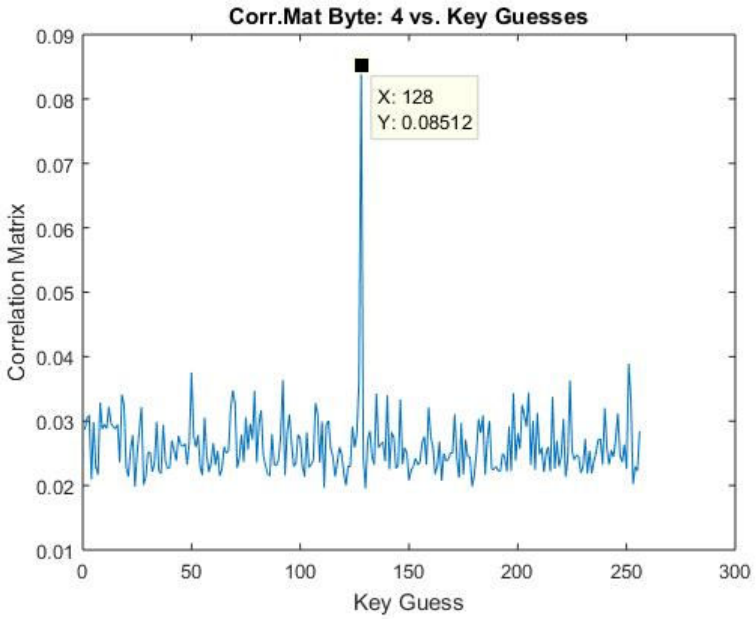


Figure A.5: 15,000 Traces Max Correlation Vector for Byte 4

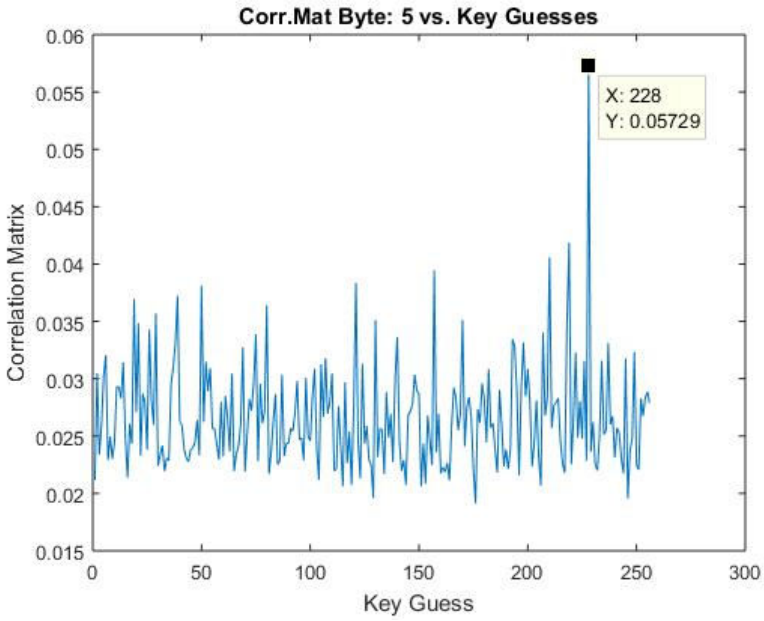


Figure A.6: 15,000 Traces Max Correlation Vector for Byte 5

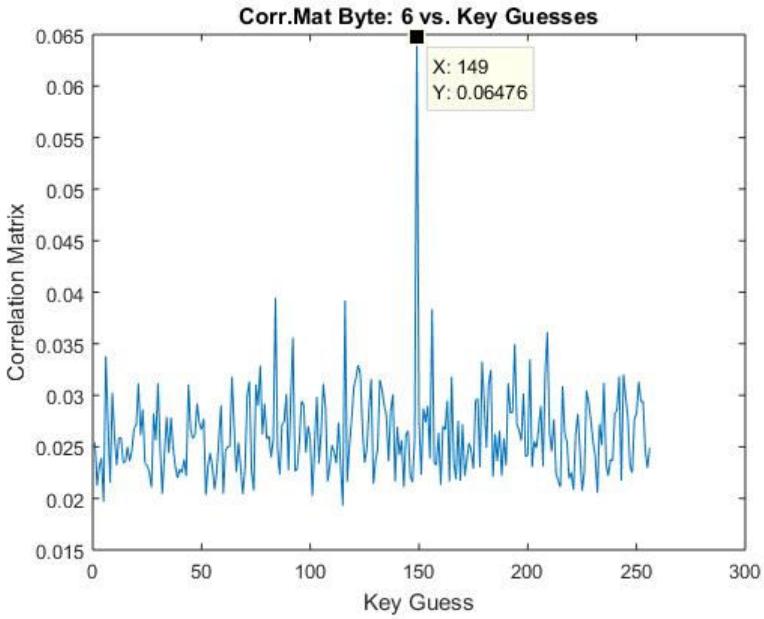


Figure A.7: 15,000 Traces Max Correlation Vector for Byte 6

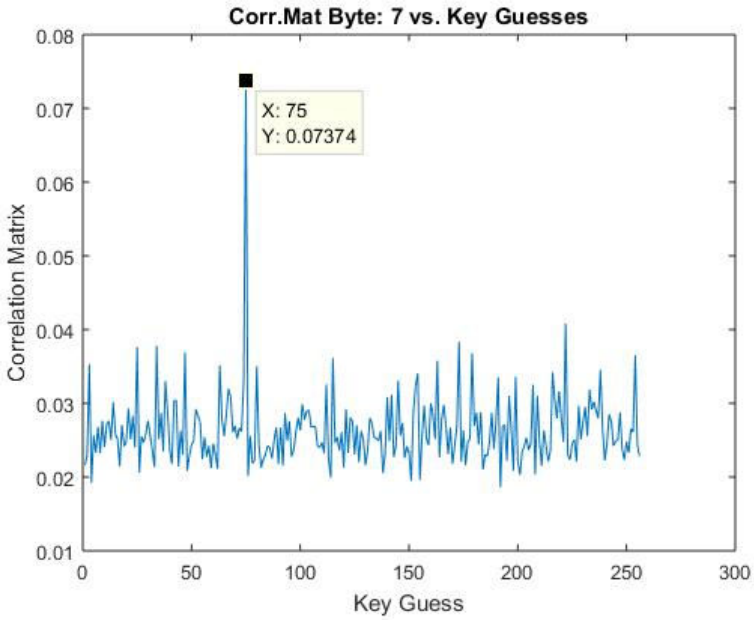


Figure A.8: 15,000 Traces Max Correlation Vector for Byte 7

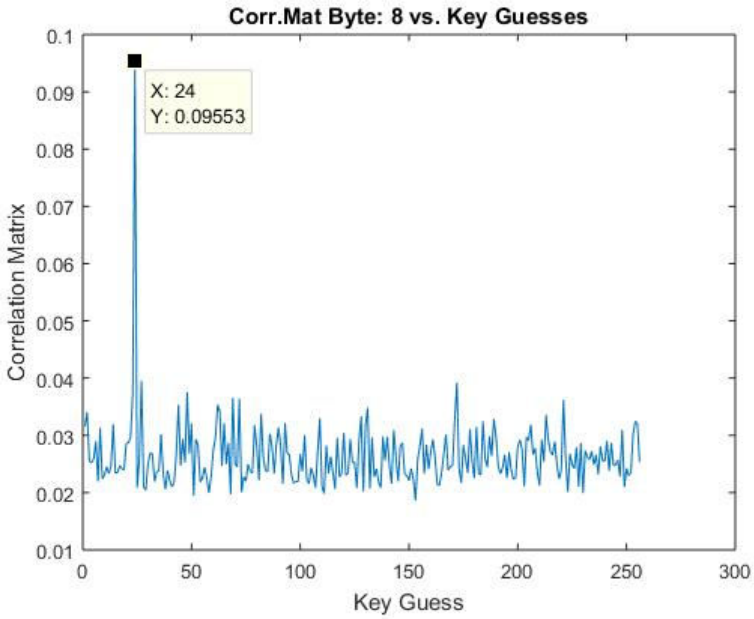


Figure A.9: 15,000 Traces Max Correlation Vector for Byte 8

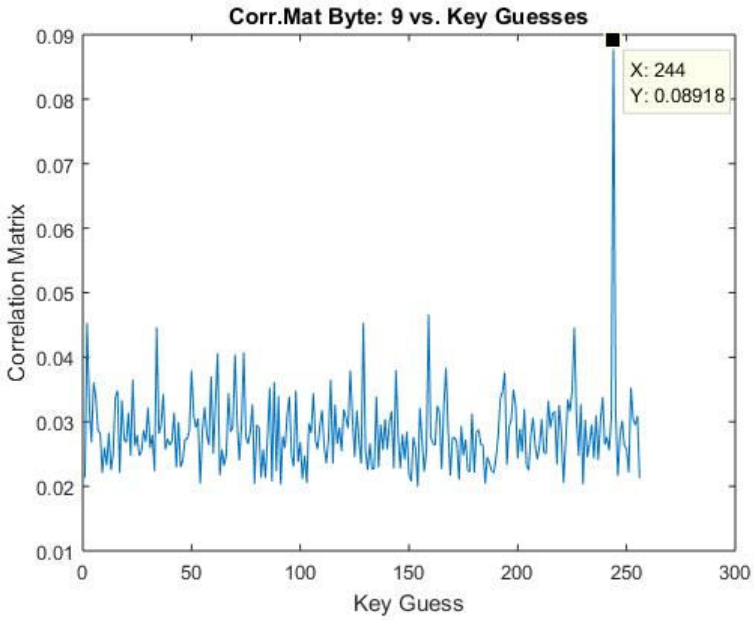


Figure A.10: 15,000 Traces Max Correlation Vector for Byte 9

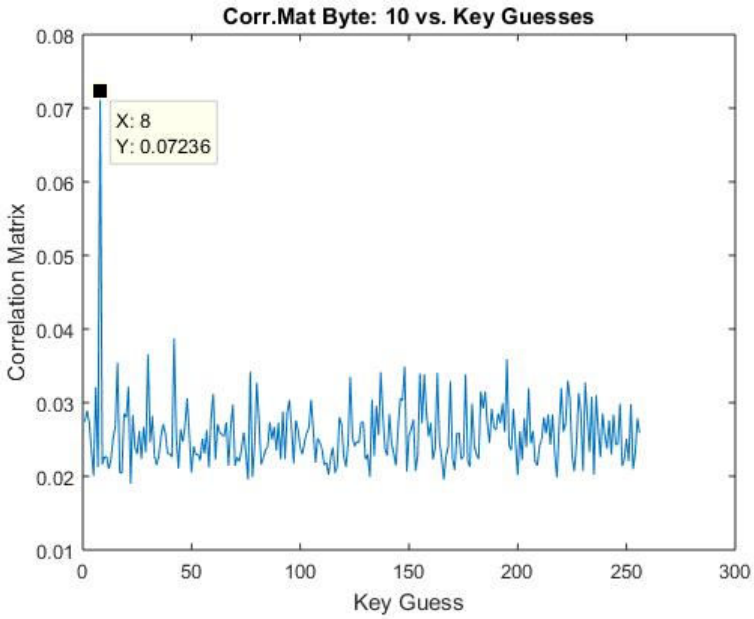


Figure A.11: 15,000 Traces Max Correlation Vector for Byte 10

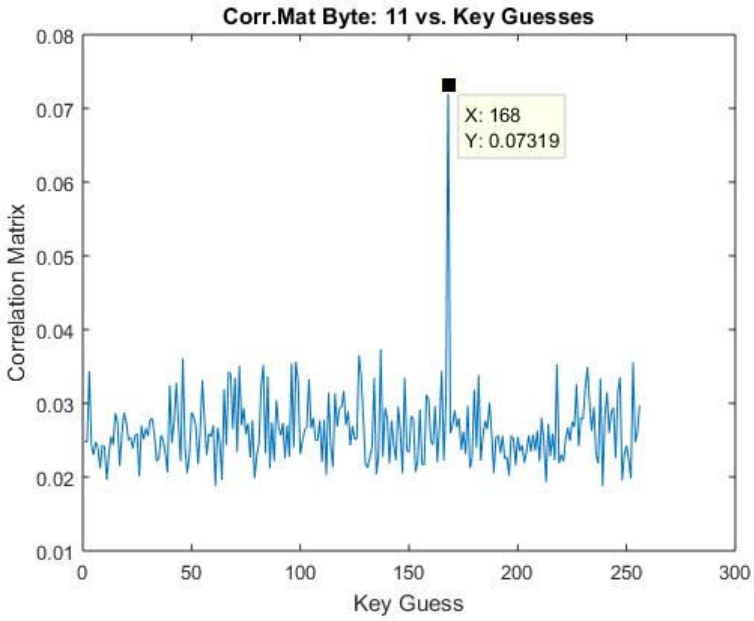


Figure A.12: 15,000 Traces Max Correlation Vector for Byte 11

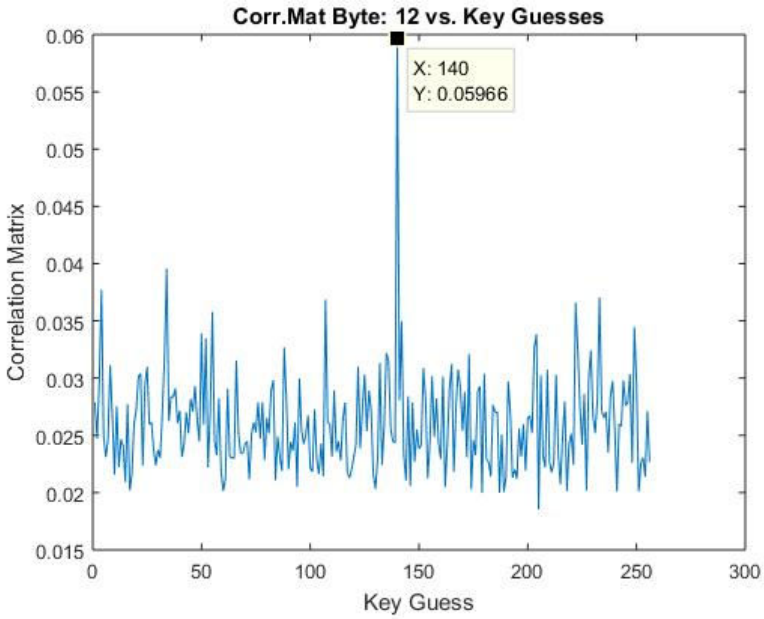


Figure A.13: 15,000 Traces Max Correlation Vector for Byte 12

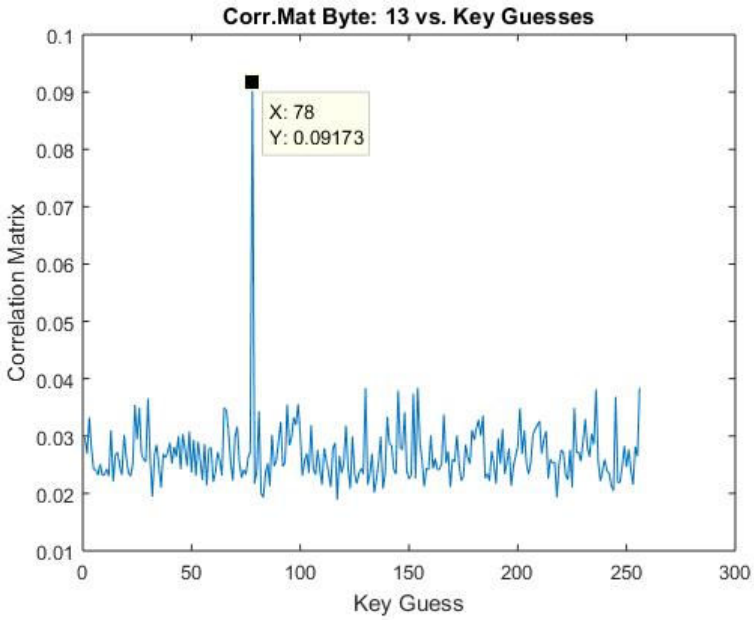


Figure A.14: 15,000 Traces Max Correlation Vector for Byte 13

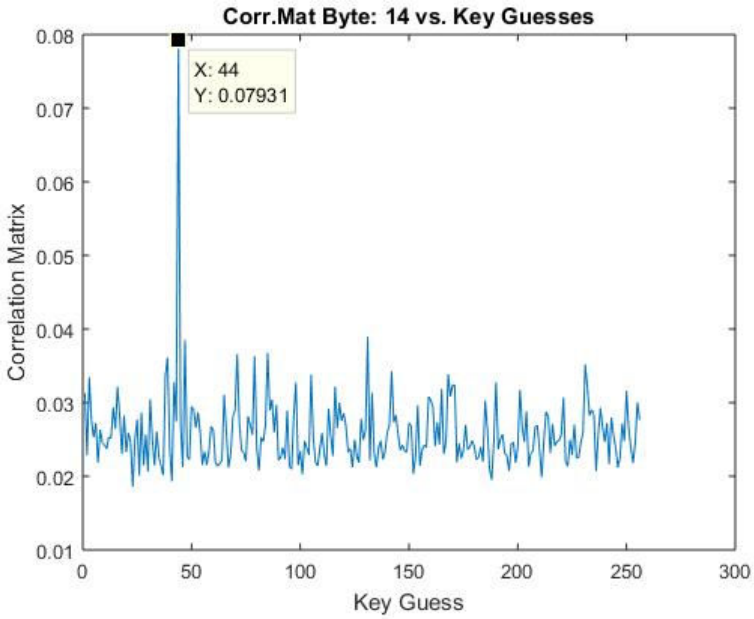


Figure A.15: 15,000 Traces Max Correlation Vector for Byte 14

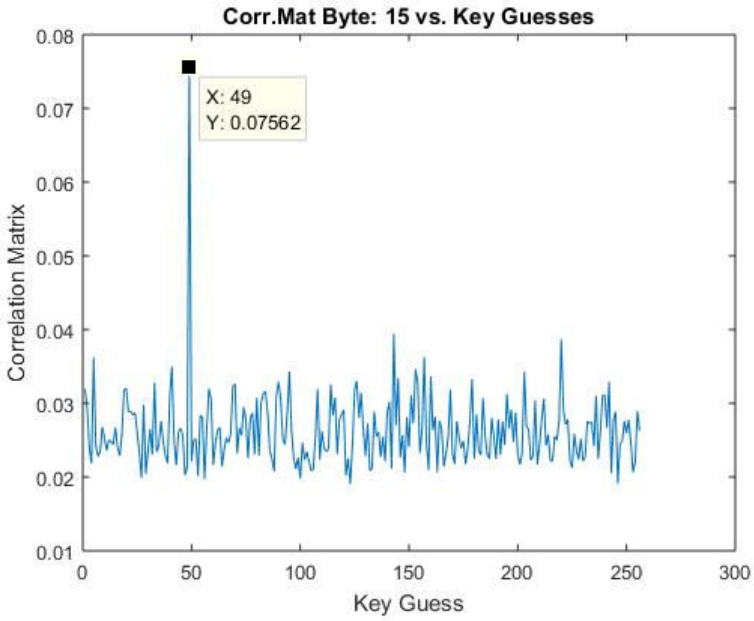


Figure A.16: 15,000 Traces Max Correlation Vector for Byte 15

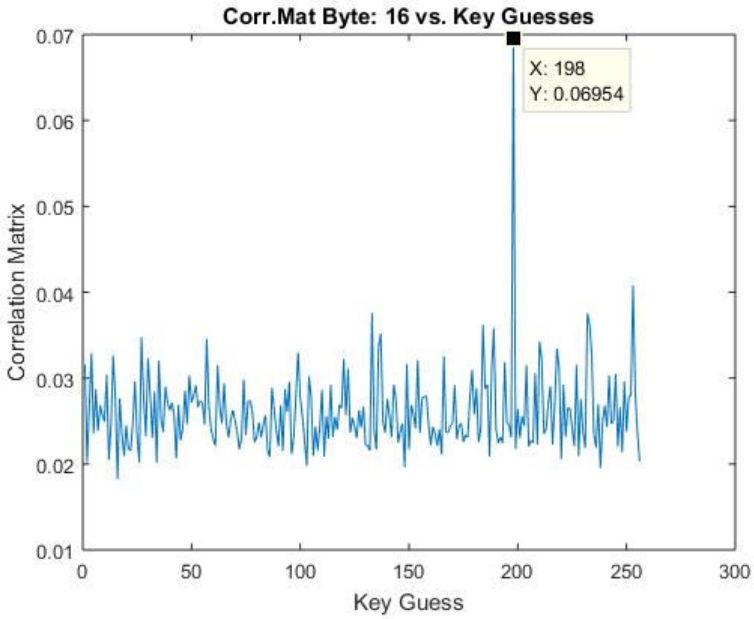


Figure A.17: 15,000 Traces Max Correlation Vector for Byte 16

Appendix B

Matlab Script DPA

Available Upon Request of Author

Appendix C

C Scripts - Verilog Script Generation

C.1 Parallel Multiplier

```
#include <stdio.h>
/*****************************************************
 Name    : mk_poly_mult
 Input   : m-bit
 Output  : poly_mult.v file
 Comment : Generate verilog code for an m-bit parallel
           multiplier
 Engineer: D.R. Lalonde
 *****************************************************/

void mk_poly_mult(int m);

int main(){

 int m = 233;

 mk_poly_mult(m);

 return 0;
```

```
}

void mk_poly_mult(int m){

 FILE *fd;
 int k,i;

 fd = fopen("poly_mult.v", "w");

// Module Declaration
 fprintf(fd, "module poly_mult(\n");
 fprintf(fd, "input [%d-1:0] a,\n", m);
 fprintf(fd, "input [%d-1:0] b,\n", m);
 fprintf(fd, "input clk,\n");
 fprintf(fd, "output reg [2*%d-2:0] d);\n\n", m);

// Module integers and reg's
 fprintf(fd, "integer k,i;\n");
 fprintf(fd, "reg a_b [2*%d-2:0][2*%d-2:0];\n", m,m);  // a & b
     for all a, b [m-1:0]
 fprintf(fd, "reg xor_temp;\n\n");

//-------------------------------------------------------------
// AND Operations ----------------------------------------------
 fprintf(fd, "always @ (*) begin\n");
    // dk = m-1, ... , 0
    for(k = 0; k <= m-1; k++){
       for(i = 0; i <= k; i++){
          // a_b[k][i] = a[i] & b[k-i]
          fprintf(fd, "a_b[%d][%d] = a[%d] & b[%d - %d];\n",
              k,i,i,k,i);
       }
    }
```

```
    // dk = 2*m-2, ... , m
    for(k = m; k <= 2*m-2; k++){
        for(i = k; i <= 2*m-2; i++){
            // a_b[k][i] = a[k-i+(m-1)] & b[i-(m-1)]
            fprintf(fd, "a_b[%d][%d] = a[%d - %d + %d-1] & b[%d -
                (%d-1)];\n", k,i,k,i,m,i,m);
        }
    }
    // d[0] has no XOR operation
    fprintf(fd, "d[0] = a_b[0][0];\n\n");

// ------------------------------------------------------------
// XOR Operations ------------------------------------------
    for(k = 1; k <= 2*m-2; k++){
        if (k <= m-1){
            fprintf(fd, "xor_temp = a_b[%d][0];\n",k);

            for(i = 1; i <= k; i++){
                fprintf(fd, "xor_temp = a_b[%d][%d] ^ xor_temp;\n",
                    k,i);
            }
        }

        else {
            fprintf(fd, "xor_temp = a_b[%d][%d];\n",k,k);

            for(i = k + 1; i <= 2*m-2; i++){
                fprintf(fd, "xor_temp = a_b[%d][%d] ^ xor_temp;\n",
                    k,i);
            }
        }

        fprintf(fd, "d[%d] = xor_temp;\n", k,k);
    }
```

```
 fprintf(fd, "end\n");

 fprintf(fd, "endmodule");
 fclose(fd);
}
```

C.2 Parallel Reduction

```
#include <stdio.h>
/*****************************************************
 Name    : mk_poly_reduc
 Input   : m-bit
 Output  : poly_reduc.v file
 Comment : Generate verilog code for an m-bit parallel
           classical multiplier
 Engineer: D.R. Lalonde
 *****************************************************/

void mk_poly_reduc(int m);

int main(){

 int m = 233;

 mk_poly_reduc(m);

 return 0;

}

void mk_poly_reduc(int m){
```

```
FILE *fd;
int i,j;

fd = fopen("poly_reduc.v", "w");

// Module Declaration
fprintf(fd, "module poly_reduc(\n");
fprintf(fd, "input [2*%d-2:0] d,\n", m);
fprintf(fd, "input [%d:0] f_x,\n", m);
fprintf(fd, "input clk,\n");
fprintf(fd, "output reg [%d-1:0] c);\n\n", m);

// Module integers and reg's
fprintf(fd, "integer i,j;\n");
fprintf(fd, "reg matR [%d-1:0][%d-2:0];\n", m,m);  // a & b for
    all a, b [m-1:0]
fprintf(fd, "reg matR_temp [%d-1:0][%d-2:0];\n", m,m);  // a &
    b for all a, b [m-1:0]
fprintf(fd, "reg xorcount;\n\n");

//-------------------------------------------------------------
// Reduction matrix R ---------------------------------------
fprintf(fd, "always @ (*) begin\n");
   // matR intilization
   for(j = 0; j <= m-1; j++){
      for(i = 0; i <= m-2; i++){
         fprintf(fd, "matR[%d][%d] = 1'b0;\n", j,i);
      }
   }
   for(j = 0; j <= m-1; j++){
      fprintf(fd, "matR[%d][0] = f_x[%d];\n", j,j);
   }

   // matR population
```

```
    for(i = 1; i <= m-2; i++){
      for(j = 0; j <= m-1; j++){
        if(j == 0){
          fprintf(fd, "matR_temp[%d][%d] = matR[%d-1][%d-1] &
              matR[%d][0];\n", j,i,m,i,j);
          fprintf(fd, "matR[%d][%d] = matR_temp[%d][%d];\n",
              j,i,j,i);
        }
        else{
          fprintf(fd, "matR_temp[%d][%d] = matR[%d-1][%d-1] ^
              (matR[%d-1][%d-1] & matR[%d][0]);\n",
              j,i,j,i,m,i,j);
          fprintf(fd, "matR[%d][%d] = matR_temp[%d][%d];\n",
              j,i,j,i);
        }
      }
    }

  // ------------------------------------------------------------
  // Polynomial Reduction -------------------------------------

    for(j = 0; j <= m-1; j++){
      fprintf(fd, "xorcount = d[%d];\n",j);

      for(i = 0; i <= m-2; i++){
        fprintf(fd, "xorcount = xorcount ^ (d[%d+%d] &
            matR[%d][%d]);\n", m,i,j,i);
      }

      fprintf(fd, "c[%d] = xorcount;\n", j);
    }

  fprintf(fd, "end\n");
```

```
 fprintf(fd, "endmodule");
 fclose(fd);
}
```

C.3 Parallel Polynomial Multiplier

```
#include <stdio.h>
/*****************************************************
 Name    : mk_classic_polyMult
 Input   : m-bit
 Output  : classic_polyMult.v file
 Comment : Generate verilog code for an m-bit parallel
           classical multiplier
 Engineer: D.R. Lalonde
 *****************************************************/

void mk_classic_polyMult(int m);

int main(){

   int m = 233;

   mk_classic_polyMult(m);

   return 0;

}

void mk_classic_polyMult(int m){

   FILE *fd;
   int i,j;
```

```
    fd = fopen("classic_polyMult.v", "w");

// Module Declaration
    fprintf(fd, "module classic_polyMult(\n");
    fprintf(fd, "input [%d-1:0] a,\n", m);
    fprintf(fd, "input [%d-1:0] b,\n", m);
    fprintf(fd, "input [%d:0] f_x,\n", m);
    fprintf(fd, "input clk,\n");
    fprintf(fd, "output [%d-1:0] z);\n\n", m);

// Wire declaration
    fprintf(fd, "wire [2*%d-2:0] d;\n", m);

// Polynomial Multiplication
    fprintf(fd, "poly_mult a0 (a,b,clk,d);\n");

// Polynomial Reduction
    fprintf(fd, "poly_reduc a1 (d,f_x,clk,z);\n");

    fprintf(fd, "endmodule");
    fclose(fd);
}
```

Appendix D

Verilog HDL Scripts

D.1 Parallel Polynomial Squarer

```
// poly_reduc.v is needed

module classic_polySquare(
input [282:0] a,
input [282:0] f_x,
input clk,

output [282:0] z

);

integer i;
reg [2*283-2:0] d;
// Polynomial Squaring
always @ (posedge clk)
begin
    d[0] <= a[0];

    for (i = 1; i <= 283-1; i = i + 1)
    begin
        d[2*i-1] = 0;
```

```
            d[2*i] = a[i];
        end
    end

// Polynomial Reduction
    poly_reduc a1 (d,f_x,clk,z);

endmodule
```

D.2 Serial Montgomery Multiplier

```
// Computes the poly multiplication A(x) B(x) R**(-1) mod
    f(x), GF(2**233)
// Output not correct, very close

module mont_datapath(
    input [232:0] c, b,
    input a_i,
    input [232:0] f_x,

    output reg [232:0] new_c
    );

    reg previous_c0;
    integer i;

    always @ (*)
    begin
        previous_c0 <= c[0] ^ (a_i & b[0]);

        for (i = 1; i <= 232; i = i + 1)
            new_c[i-1] <= c[i] ^ (a_i & b[i]) ^ (f_x[i] &
                previous_c0);
```

```
        new_c[232] <= previous_c0;
    end

endmodule

// Montgomery Control
    Unit-------------------------------------------------------------
module mont_mult(
    input [232:0] a,
    input [232:0] b,
    input [232:0] f_x,
    input clk, go, reset,

    output reg int_done, done_mult,
    output reg [232:0] z

    );

/*   wire [8:0] w_aa, w_bb, w_cc, w_new_c;
    always @ (w_aa) aa = w_aa;
    always @ (w_bb) bb = w_bb;
    always @ (w_cc) cc = w_cc;
    always @ (w_new_c) new_c = w_new_c;*/

    reg [232:0] aa, bb, cc;
    wire [232:0] n_c;

// Datapath
    mont_datapath md1(.c(cc), .b(bb), .a_i(aa[0]), .f_x(f_x),
        .new_c(n_c));

    reg incr, shift_right;
```

```
// Counter
    reg [3:0] count;                    // decimal 5
    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            count <= 0;
        else
        begin
            if (incr)
                count <= 0;
            else if (shift_right)
                count <= count + 1;
        end
    end

// Shift Register A
    always @ (posedge clk)
    begin
        if (reset)
            aa <= 0;
        else
        begin
            if (incr)
                aa <= a;
            else
            aa <= {1'b0, aa[232:1]};
        end
    end

// Register B
    always @ (posedge clk)
    begin
        if (reset)
            bb <= 0;
```

```
        else
            if (incr)
                bb <= b;
    end

// Register C
    reg c_en;
    always @ (posedge clk or posedge incr)
    begin
        if (incr | reset)
            cc <= 0;
        else
            if (c_en)
            begin
                cc <= n_c;
                z <= cc;
            end
    end


// FSM
    reg [2:0] state;
    always @ (state)
    begin
        case (state)
            0:  begin
                incr <= 0;
                shift_right <= 0;
                int_done <= 1;
                c_en <= 0;
                end

            1:  begin
                incr <= 0;
```

```
                shift_right <= 0;
                int_done <= 1;
                c_en <= 0;
                end

            2:  begin
                incr <= 1;
                shift_right <= 0;
                int_done <= 0;
                c_en <= 0;
                end

            3:  begin
                incr <= 0;
                shift_right <= 1;
                int_done <= 0;
                c_en <= 1;
                end
        endcase
    end

// Next state
    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            state <= 0;
        else if (clk)
        begin
            case (state)
                0:
                    if (!go)
                        state <= 1;
                    else
                        state <= 0;
```

```
                1:
                    if (go)
                        state <= 2;
                    else
                        state <= 1;
                2:
                    state <= 3;
                3:
                    if (count == 232)
                    begin
                        state <= 0;
                        done_mult <= 1;
                    end
                    else
                        state <= 3;
            endcase
        end
    end

endmodule
```

D.3 32-bit FIFO

```
// 32-bit FIFO for 233 bits of information -> 256-bit
    capability

module sync_fifo (
        input [31:0] in_fifo,
        input rd_en,
        input wr_en,
        input clk,
        input reset,
```

```
        output reg [31:0] out_fifo,
        output empty,
        output full
    );

// 4 bits to count to decimal 8 (depth)
    reg [3:0] p_rd, p_wr;

// Declare the fifo memory (RAM that allow read and write at
    the same time)
// creates an array of 8 elements of 233 bits
    reg [31:0] mem_fifo [7:0];

// Flags
    reg [4:0] counter_fifo;                          // 4
        bits to count to decimal 8 (depth) + 1 bit for space

    assign empty = (counter_fifo == 0);              //
        Completely empty
    assign full = (counter_fifo == 8);               // ''

// Sequential circuit that checks empty & full flags
    always @(posedge clk or negedge reset)
    begin
        if (~reset)
            counter_fifo <= 0;
        else if( (!full && wr_en) && ( !empty && rd_en ) )
            counter_fifo <= counter_fifo;            // If
                read and write
        else if (!full && wr_en)
            counter_fifo <= counter_fifo + 1;        //
                Write -> increment
        else if (!empty && rd_en)
```

```
            counter_fifo <= counter_fifo - 1;          //
                Read -> decrement
        end

    // Sequential circuit - READING
        always @(posedge clk or negedge reset)
        begin
            if(!reset)
                out_fifo <= 0;
            else
                if (!empty && rd_en)                       // Not
                    empty and READ
                    out_fifo <= mem_fifo [p_rd];
        end

    // Sequential circuit - WRITING
        always @(posedge clk)
           if (!full && wr_en)
                mem_fifo[p_wr] <= in_fifo;

    // Sequential circut - read/write POINTERS
        always @(posedge clk or negedge reset)
        begin
            if(!reset)
            begin
                p_wr <= 0;
                p_rd <= 0;
            end
            else
            begin
                // Not full and WRITE -> incr. write pointer
                if( !full && wr_en )
                    p_wr <= p_wr + 1;
```

```
                // Not empty and READ -> decr. read pointer
                if( !empty && rd_en )
                    p_rd <= p_rd + 1;
            end
        end

    endmodule
```

D.4 Serialized Montgomery Multiplier Comparison

This comparison displays the Area-Delay Product of the amount of LUTs and registers present in the Kintex-7 needed after synthesis of the Montgomery multiplier when compared to recent literature. This comparison is not entirely valid due to the architecture not being synthesized as a whole, resulting in the use of this table to be strictly used as a general guide.

Table D.1: Serialized Multiplier Comparison

Multiplier	Key (Bits)	Clock (MHz)	Area-Delay (slice*sec)
Karatsuba [52]	233	625	0.111
Interleaved [53]	283	264	-
Montgomery [54]	233	115.47	1.086
Montgomery [55]	163	132.5	1.098
This work: Montgomery	233	Approx. 250	0.00000564
This work: Montgomery	283	Approx. 250	0.00000684

Appendix E

Verilog HDL Pseudo Scripts

E.1 Binary Extended Euclidean Inversion

```
module EEA_test(
input [282:0] a,
input [283:0] f_x,
input clk,
input reset,

output reg [282:0] z

);

// Datapath
    reg [283:0] r, s, u, v;
    reg [8:0] d;                    // [log283:0]
    reg [283:0] r_q, s_q, u_q, v_q; // New registers
    reg [8:0] d_q;                  // [log283:0]

    always @ (posedge clk)
    begin
        //_______________________________
        // Alg for Inv in GF(2**m): 3 - 6
        if (r[283] == 0)
```

```
begin
    r_q <= {r[282:0], 1'b0};                // cyclic
        shift right 1
    u_q <= {u[282:0], 1'b0};                // ''
    s_q <= s;                               // same
        since rm = 0 - unchanged
    v_q <= v;                               // ''
    d_q <= d + 1;
end

//_______________________________
// when d = 0
else
begin
    if (d == 0)
    begin
        if (s[283] == 1)
        begin                               //
            Combined operations
            r_q <= {s[282:0] ^ r[282:0], 1'b0}; //
                Line: 9, 12, 14
            u_q <= {v[282:0] ^ u[282:0], 1'b0}; //
                Line: 10, 15
        end

        else
        begin
            r_q <= {s[282:0], 1'b0};
            u_q <= {v[282:0], 1'b0};
        end

        s_q <= r;
        v_q <= u;
```

```
            d_q[0] <= 1'b1;                          // d_q
                <= (0=> '1', others => '0'); vdhl
            d_q[283:1] <= 0;
        end

        //--------------------------------
        // when d = otherwise
        else
        begin
            r_q <= r;
            u_q <= {1'b0, u[283:1]};                 //
                Cylc shift left 1, Line: 18 (division)

            if (s[283] == 1)
            begin
                s_q <= {s[282:0] ^ r[282:0], 1'b0}; //
                    Line: 9
                v_q <= v ^ u;                        //
                    Line: 10
            end
            else
            begin
                s_q <= {s[282:0], 1'b0};             //
                    Line: 12
                v_q <= v;
            end

            d_q <= d - 1;                            //
                Line: 19
        end

    end // 1st if

    z <= u[282:0];
```

```
    end

endmodule
```

E.2 Point Double

```
// Pt. Doubling in LD coords
// Y**2 + XYZ = X**3z + X**2Z**2 + a Z**4 LD-Elliptic Curve
   mapping
// // P (X, Y, Z) = Q (X, Y, 1) ... 2P = (X3, Y3, Z3)
/*     1. Z3 = X1**2 Z1**2
     2. X3 = X1**4 + Z1**4
     3. Y3 = b Z1**4 Z3 + X3 (a Z3 + Y1**2 + b Z1**4)
*/
// -------------------------------------------------------

module pt_double (
    input in_x1, in_y1, in_z1,
    input f_x,
    input clk,
    input reset,

    output reg x3, y3, z3,
    output reg infinity

    );

// -------------------------------------------------------
    reg [282:0] x1, y1, z1;
    reg [282:0] a, b, c;
    reg [1:0] count;
    reg done;
```

```
    wire [2*283-2:0] w_x1, w_z1, w_y1, w_z3, w_x1_1, w_z1_1,
        w_x3, w_a, w_b, w_c,
                        w_y3, w_y3_1;

// 1. Z3 = X1**2 Z1**2
// ****************************************************
    classic_polySquare s0 (.a(x1), .f_x(f_x), .clk(clk),
        .z(w_x1));          // clk0
    classic_polySquare s1 (.a(z1), .f_x(f_x), .clk(clk),
        .z(w_z1));          // clk0
    classic_polyMult m0 (.a(w_x1), .b(w_z1), .f_x(f_x),
        .clk(clk), .z(w_z3)); // clk1

// 2. X3 = X1**4 + Z1**4
// ****************************************************
    classic_polySquare s2 (.a(w_x1), .f_x(f_x), .clk(clk),
        .z(w_x1_1));      // clk1
    classic_polySquare s3 (.a(w_z1), .f_x(f_x), .clk(clk),
        .z(w_z1_1));      // clk1

// 3. Y3 = Y3 = b Z1**4 Z3 + X3 (a Z3 + Y1**2 + b Z1**4)
// ****************************************************
    classic_polySquare s4 (.a(y1), .f_x(f_x), .clk(clk),
        .z(w_y1));          // clk0
    classic_polyMult m1 (.a(w_z1_1), .b(w_z3), .f_x(f_x),
        .clk(clk), .z(w_a)); // clk2
    classic_polyMult m2 (.a(w_x3), .b(w_b), .f_x(f_x),
        .clk(clk), .z(w_y3)); // clk2

// What #clk edge is present, for specific wires XOR
    always @ (posedge clk && !done)
    begin
        if (reset)
        begin
```

```
            count <= 0;
            done <= 0;
            x1 <= in_x1;
            y1 <= in_y1;
            z1 <= in_z1;
        end
        else
        begin
            if (count == 1)
            begin
                x3 <= w_x1_1 ^ w_z1_1;       // 2. 1st clock
                    cycle, x1 XOR z1 = X3
                b <= w_z1_1 ^ w_z3 ^ w_y1;   // 3. 1st clock
                    cycle, z1 XOR z3 = b
            end
            if (count == 2)
            begin
                y3 <= w_a ^ w_y3;            // 3. 2nd clock
                    cycle, a XOR wire y3 = y3
            end
        end
        count <= count + 1;
    end

// @ wire changes, make the corresponding register
    available... in order of the design
// 1.
    always @ (w_z3)
        z3 = w_z3;
// 2.
    always @ (w_x3)
        x3 = w_x3;
// 3.
    always @ (w_b)
```

```
        b = w_b;

    always @ (w_y3_1)
    begin
        y3 = w_y3_1;
        done = 1;
    end



    // NEEDS MORE WORK
    // Check for infinity
    always @ (posedge clk && done)
    begin
        if (x3 && y3 == 0)
            if (z3 == 1)
                infinity <= 1;
    end

endmodule
```

E.3 Point Addition

```
// Pt. Addition in LD coords
// Y**2 + XYZ = X**3z + X**2Z**2 + a Z**4 LD-Elliptic Curve
   mapping
// P (X, Y, Z) ~= Q (X, Y, 1) ... xP = P + Q =(X3, Y3, Z3)
/*      1. A = Y2 Z1**2 + Y1      2. B = X2 Z1 + X1
      3. C = Z1 B              4. D = B**2 (C + a Z1**2), a =
         1
      5. Z3 = C**2             6. E = A C
      7. X3 = A**2 + D + E     8. F = X3 + X2 Z3
      9. G = (X2 + Y2) Z3**2   10. Y3 = (E + Z3) F + G
*/
```

```
// -------------------------------------------------------

module pt_add (
    input in_x1, in_y1, in_z1,
    input in_x2, in_y2,
    input f_x,
    input clk,
    input reset,

    output reg x3, y3, z3

    );

// -------------------------------------------------------
    reg [282:0] x1, y1, z1, x2, y2;
    reg [282:0] a, b, c, d, e, f, g, y3_1;
    reg [1:0] count;
    reg done;

    wire [282:0] w_z1, w_a, w_b, w_x1, w_x2_1, w_x2, w_y2,
        w_g, w_a_1, w_d, w_c, w_a_2, w_e,
                    w_d_1, w_x3, w_z3, w_y3, w_z3_1, w_z3_2,
                        w_f, w_g_1, w_y3_1, w_y3_2;

// 1. A = Y2 Z1**2 + Y1
// *******************************************************
    classic_polySquare s0 (.a(z1), .f_x(f_x), .clk(clk),
        .z(w_z1));          // clk0
    classic_polyMult m0 (.a(w_z1), .b(y2), .f_x(f_x),
        .clk(clk), .z(w_a));  // clk1

// 2. B = X2 Z1 + X1
// *******************************************************
```

```
    classic_polyMult m1 (.a(z1), .b(x2), .f_x(f_x),
        .clk(clk), .z(w_a));    // clk0

// 3. C = Z1 B
// ********************************************************
    classic_polyMult m2 (.a(z1), .b(w_b), .f_x(f_x),
        .clk(clk), .z(w_c));   // clk1

// 4. D = B**2 (C + a Z1**2)
// ********************************************************
    classic_polySquare s1 (.a(w_b), .f_x(f_x), .clk(clk),
        .z(w_b_1));       // clk1
    classic_polyMult m3 (.a(w_d), .b(w_b_1), .f_x(f_x),
        .clk(clk), .z(w_d_1)); // clk1

// 5. Z3 = C**2
// ********************************************************
    classic_polySquare s2 (.a(w_c), .f_x(f_x), .clk(clk),
        .z(w_z3));        // clk1

// 6. E = A C
// ********************************************************
    classic_polyMult m4 (.a(w_c), .b(w_a_1), .f_x(f_x),
        .clk(clk), .z(w_e)); // clk2

// 7. X3 = A**2 + D + E
// ********************************************************
    classic_polySquare s3 (.a(w_a_1), .f_x(f_x), .clk(clk),
        .z(w_a_2));     // clk2

// 8. F = X3 + X2 Z3
// ********************************************************
    classic_polyMult m5 (.a(x2), .b(w_z3), .f_x(f_x),
        .clk(clk), .z(w_z3_1)); // clk2
```

```
// 9. G = (X2 + Y2) Z3**2
// ********************************************************
    classic_polySquare s4 (.a(w_z3), .f_x(f_x), .clk(clk),
        .z(w_z3_2));     // clk3
    classic_polyMult m6 (.a(w_z3_2), .b(w_g), .f_x(f_x),
        .clk(clk), .z(w_g_1)); // clk4

// 10. Y3 = (E + Z3) F + G
// ********************************************************
    classic_polyMult m7 (.a(w_f), .b(w_y3), .f_x(f_x),
        .clk(clk), .z(w_y3_1)); // clk3


// What #clk edge is present, for specific wires XOR
    always @ (posedge clk)
    begin
        if (reset)
        begin
            count <= 0;
            done <= 0;
            x1 <= in_x1;
            y1 <= in_y1;
            z1 <= in_z1;
            x2 <= in_x2;
            y2 <= in_y2;
        end
        else
        begin
            if (count == 0)
            begin
                b <= x1 ^ w_x2_1;
                g <= x2 ^ y2;
            end
```

```
            if (count == 1)
            begin
                a <= w_a ^ y1;
                d <= w_z1 ^ w_c;
            end
            if (count == 2)
            begin
                x3 <= w_a_2 ^ w_e ^ w_d_1;
                y3_1 <= w_e ^ w_z3;
            end
            if (count == 3)
                f <= w_x3 ^ w_z3_1;
            if (count == 4)
                y3 <= w_g_1 ^ w_y3_1;
        end
        count <= count + 1;
    end

// @ wire changes, make the corresponding register available
// CLK 0
// 2.
    always @ (w_b)
        b = w_b;
// 9.
    always @ (w_g)
        g = w_g;
// CLK 1
// 1.
    always @ (w_a_1)
        a = w_a_1;
// 4.
    always @ (w_d)
        d = w_d;
// CLK 2
```

```
// 7.
    always @ (w_x3)
        x3 = w_x3;
// 10.
    always @ (w_y3)
        y3_1 = w_y3;
// CLK 3
// 8.
    always @ (w_f)
        f = w_f;
// CLK 4
// 10.
    always @ (w_y3_2)
        y3 = w_y3_2;


endmodule
```

Bibliography

[1] Dell EMC, "A Cost-based Security Analysis of Symmetric and Asymmetric Key Lengths", *RSA Laboratories*, 2010.

[2] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization", *Mathematics of Computation*, vol. 48, no. 177, pp. 243– 264, 1987.

[3] M. Joye, "Highly regular m-ary powering ladders", *International Workshop on Selected Areas of Cryptography*, Springer Berlin Heidelberg 2009.

[4] D. Freeman, "Pertinent Side Channel Attacks on Elliptic Curve Cryptographic Systems", *IEEE Transactions on Energy Conversion*, vol. 26, no. 1, pp. 55-63, March 2011.

[5] I. H. Hazmi, F. Zhou, F. Gebali, "Review of Elliptic Curve Processor Architectures", *IEEE*, 78-1-4673-7788-1/15, 2015.

[6] M. Joye, S. Yen, "The Montgomery Powering Ladder", *Laboratory of Cryptography and Information Security (LCIS)*, Springer-Verlag Berlin Heidelberg 2003

[7] A. Sghaier, M. Zeghid, B. Bouallegue, A. Baganne, M. Machhout, "Area Time Efficient Hardware Implementation of Elliptic Curve Cryptosystem", 2015

[8] Y. Dan et al., "High-performance hardware architecture of elliptic curve cryptography processor over gf(2$\hat{1}$63)," *Journal of Zhejiang University Science A*, vol. 10, no. 2, pp. 301–310, 2009.

[9] C. Puttmann et al., "Hardware accelerators for elliptic curve cryptography," *Advances in Radio Science*, vol. 6, no. 10, pp. 259–264, 2008.

[10] M. Amara and A. Siad, "Hardware implementation of elliptic curve point multiplication over gf(2m) for ecc protocols," *International Journal for Information Security Research (IJISR)*, vol. 1, no. 3, 2011.

[11] M.A.Fayed, "A security coprocessor for next generation IP telephony: architecture, abstraction, and strategies". University of Victoria, 2007.

[12] K. Jarvinen, "Optimized fpga-based elliptic curve cryptography processor for high-speed applications," *INTEGRATION, the VLSI journal*, vol. 44, no. 4, pp. 270–279, 2011.

[13] M. Morales-Sandoval, "A reconfigurable and interoperable hardware architecture for elliptic curve cryptography," *Ph.D. dissertation, Tesis de Doctorado, Instituto Nacional de Astrofısıca, Optica y Electronica, Mexico*, 2008

[14] K. Ananyi et al., "Flexible hardware processor for elliptic curve cryptography over nist prime fields," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, vol. 17, no. 8, pp. 1099–1112, 2009.

[15] S. Zeidler et al., "Design of a low-power asynchronous elliptic curve cryptography coprocessor," in *Electronics, Circuits, and Systems (ICECS), 2013 IEEE 20th International Conference* on., pp. 569–572 IEEE, 2013.

[16] T. Akishita, T. Takagi, "Zero-Value Point Attacks on Elliptic Curve Cryptosystem," *Sony Corporation, Ubiquitous Technology Laboratories*, Technische Universitat Darmstadt, Fachbereich Informatik, Germany

[17] R. Karri, K. Wu, P. Mishra, Yongkook Kim, "Fault-based side-channel cryptanalysis tolerant Rijndael symmetric block cipher architecture," *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium*, 2001.

[18] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their applications*, Cambridge University Press, 1994.

[19] A. Irwansyah, V.P. Nambiar, M. Khalil-Hani, "An AES Tightly Coupled Hardware Accelerator in an FPGA-based Embedded Processor Core ," *2009 International Conference on Computer Engineering and Technology*, 2009.

[20] H.W. Lenstra, R.J. Schoof Jr., "Primitive normal bases for finite fields," *Mathematics of Computation*, 48: 217–231, 1987.

[21] A. F. Diego, S. L. Paulo, M. Barreto, R. E. Jefferson, "A note on high-security general-purpose elliptic curves," *Computer Science Dept, University of Brasília*, Cryptology ePrint Archive, Report 2013, 647 (2013).

[22] S. Ezzouak, M. Elamrani, A. Azizi, "Improving Pollard's Rho Attack on Elliptic Curve Cryptosystems" *IEEE Transactions*, 978-1-4673-1520-3/12, ©2012 IEEE

[23] P. C. Kocher, "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems," *Proc. CRYPTO*, vol. 1109, pp.104–113, 1996

[24] P. C. Kocher, J. Jaffe, B. Jun, "Differential Power Analysis," technical report, 1998; *Advances in Cryptology - Crypto 99 Proceedings*, Lecture Notes In Computer Science Vol. 1666, M. Wiener, ed., Springer-Verlag, 1999.

[25] M. Alioto, S. Member, M. Poli, S. Rocchi, "A General Power Model of Differential Power Analysis Attacks to Static Logic Circuits," *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, VOL. 18, NO. 5, May 2010.

[26] W. Shan, X. Fu, Z. Xu, "A Secure Reconfigurable Crypto IC With Countermeasures Against SPA, DPA, and EMA," *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, vol. 34, no. 7, July 2015.

[27] K. Tiri, I. Verbauwhede, "A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation," *Proc. Conf. Design, Automation and Test in Europe, IEEE Computer Society, Washington*, DC, pp. 10246, 2004.

[28] Z. Toprak, Y. Leblebici, "Low-power current mode logic for improved DPA-resistance in embedded systems," *Proc. IEEE Int. Symp. Cir. Sys.*, pp. 10591062, 2005.

[29] M. W. Allam, M. I. Elmasry, “Dynamic current mode logic (DyCML): A new low-power high-performance logic style,” *IEEE J.Solid-State Circuits*, vol. 36, no. 3, pp. 550558, Mar. 2001.

[30] C. Tokunaga, D. Blaauw, “Securing Encryption Systems With a Switched Capacitor Current Equalizer,” *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, vol. 45, no. 1, January 2010

[31] D. A. Osvik1, A. Shamir, E. Tromer2 “Cache Attacks and Countermeasures: the Case of AES,” Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel revised 2005

[32] D. Bernstein “Cache-timing attacks on AES” Department of Mathematics, Statistics, and Computer Science (M/C 249) The University of Illinois at Chicago, 2005

[33] X. Duan, Q. Cui, S. Wang, H. Fang, G. She, ”Differential Power Analysis Attack and Efficient Countermeasures on PRESENT,” *2016 8th IEEE International Conference on Communication Software and Networks*, 2016

[34] NIST ”Recommended Elliptic Curves for Federal Government Use,” http://csrc.nist.gov, 2004.

[35] H. Yue, ”Efficient Scalar Multiplication Against Side Channel Attacks Using a New Binary Representation”, *1st Seminar - University of Windsor*, 2016.

[36] M. Yasuda, ”On the Strength Comparison of ECC and RSA”, *SHARCS 2012 (Special-Purpose Hardware for Attacking Cryptographic Systems)*, Fujisa Laboratories Ltd., 2012.

[37] H. Wu, ”AES: Advanced Encryption Standard”, *Chapter 5: Data Security and Cryptography - University of Windsor*, 2015.

[38] F. K. Gürkaynak, Side Channel Attack Chapter 3: Secure Cryptographic Accelerators, 2006

[39] Rodriguez-Henriquez, F., Saqib, N.A., Diaz Pérez, A., Koc, C.K., ”Cryptographic Algorithms on Reconfigurable Hardware”, Springer, 2007.

[40] Jean-Pierre Deschamps, José Luis Imaña, Gustavo D. Sutter, "Hardware Implementations of Finite-Field Arithmetic", The McGraw-Hill Companies, Inc., 2009.

[41] C. Paar, J. Pelzl, "Understanding Cryptography", Springer, 2010.

[42] S. Mangard, E. Oswald, T. Popp, "Power Analysis Attacks – Revealing the Secrets of Smart Cards", Springer, 2007.

[43] V. Miller, "Use of Elliptic Curves in Cryptography", *Advances in Cryptology-CRYPTO 85 Proceedings*, Springer, pp. 417-426, 1986.

[44] N. Koblitz, "Elliptic Curve Cryptosystems", *Mathematics of Computations*, vol. 48, no. 177, pp. 203-209, 1987.

[45] Leboeuf, Karl Bernard, "GPU and ASIC Acceleration of Elliptic Curve Scalar Point Multiplication" (2012). Electronic Theses and Dissertations. Paper 5367.

[46] (2016) Cryptography Stack Exchange. [Online]. Available: http://crypto.stackexchange.com

[47] (2016) Safe Curves - Choosing safe curves for elliptic curve cryptography. [Online]. Available: https://safecurves.cr.yp.to

[48] (2016) Internet stats - Live internet feed. [Online]. Available: http://www.internetlivestats.com/internet-users

[49] (2016) NSA - CSA - NSA Security Assurance. [Online]. Available: http://www.nsa.gov/what-we-do/information-assurance/

[50] (2016) DPA Contest, "DPA Contest v4" [Online] Available: http://www.dpacontest.org/home/ Accessed July 2016.

[51] (2016) Chip Whisperer, "Open-Sourced SCA tools" [Online] Available: https://newae.com/tools/chipwhisperer/ Accessed June 2016.

[52] R. Bilal and M. Rajaram, "Design and evaluation of parallel, scalable, curve based processor over binary field," WSEAS Transactions on Computers, vol. 10, no. 10, pp.353–365, 2011.

[53] M.A.Fayed, "A security coprocessor for next generation IP telephony: architecture, abstraction, and strategies". University of Victoria, 2007.

[54] R. Bilal and M. Rajaram, "Design and evaluation of parallel, scalable, curve based processor over binary field," WSEAS Transactions on Computers, vol. 10, no. 10, pp. 353–365, 2011.

[55] Y. W. R.Li, "Fpga based unified architecture for public key and private key cryptosystems," Frontiers of Computer Science, vol. 7, no. 3, pp. 307–316, 2013.

Vita Auctoris

Dylan was born in May 1994, in Windsor, Ontario. He received his B.A.Sc. and M.A.Sc. degrees in Electrical and Computer Engineering from the University of Windsor in 2016 and 2017 respectively - Windsor Ontario, Canada.

His research interest includes cryptography, side-channel attacks, information and algorithm security, automotive cyber-security, FPGA and ASIC accelerators, and analog circuit design.