Design Methods for Cryptanalysis

Lyndon V. Judge

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Patrick R. Schaumont, Chair
Peter M. Athanas
Leyla Nazhandali

November 29, 2012
Blacksburg, Virginia

Keywords: Design method, Bluespec, Hardware software co-design, FPGA, Elliptic curve cryptography (ECC), Pollard rho, Prime field arithmetic, Implementation attack, Side-channel analysis (SCA), Fault attack

Design Methods for Cryptanalysis

Lyndon V. Judge

(ABSTRACT)

Security of cryptographic algorithms relies on the computational difficulty of deriving the secret key using public information. Cryptanalysis, including logical and implementation attacks, plays an important role in allowing the security community to estimate their cost, based on the computational resources of an attacker. Practical implementations of cryptanalytic systems require complex designs that integrate multiple functional components with many parameters.

In this thesis, methodologies are proposed to improve the design process of cryptanalytic systems and reduce the cost of design space exploration required for optimization. First, Bluespec, a rule-based HDL, is used to increase the abstraction level of hardware design and support efficient design space exploration. Bluespec is applied to implement a hardware-accelerated logical attack on ECC with optimized modular arithmetic components. The language features of Bluespec support exploration and this is demonstrated by applying Bluespec to investigate the speed area tradeoff resulting from various design parameters and demonstrating performance that is competitive with prior work. This work also proposes a testing environment for use in verifying the implementation attack resistance of secure systems. A modular design approach is used to provide separation between the device being tested and the test script, as well as portability, and openness. This yields an open-source solution that supports implementation attack testing independent of the system platform, implementation details, and type of attack under evaluation. The suitability of the proposed test environment for implementation attack vulnerability analysis is demonstrated by applying the environment to perform an implementation attack on AES. The design of complex cryptanalytic hardware can greatly benefit from better design methodologies and the results presented in this thesis advocate the importance of this aspect.

That this work received support from the National Science Foundation (NSF) and National Institute of Standards and Technology (NIST)

Acknowledgments

It is with sincerest gratitude that I acknowledge the support and help of my advisor Professor Patrick Schaumont. His dedication, enthusiasm, and immense knowledge have been a constant source of inspiration for me. I have learned so much from him and with his guidance I have grown both as a student and an individual. This work would not have been possible without his mentorship and encouragement.

I am honored to have Professors Peter Athanas and Leyla Nazhandali as my advisory committee. I am very grateful to each of them for the time they have given to serve on my committee, as well as their valuable guidance and suggestions on my work.

I would like to acknowledge the National Science Foundation and express my sincere gratitude for its support of my research through grant 0644070. I am also grateful to the National Institute of Standards and Technology for its support under the American Recovery and Reinvestment Act (NIST-ARRA) through grant 60NANB10D004.

Special thanks to the many faculty and staff of the Bradley Department of Electrical and Computer Engineering at Virginia Tech. In particular, I would like to thank Professor Paul Plassmann and Bob Lineberry for challenging me academically and encouraging me to advance my knowledge and skills beyond my comfort level. I am also grateful to Mary Taylor for helping me navigate the administrative requirements for completing my degree.

This thesis would not be possible without the support and hard work of my collaborators: Suvarna Mane, Cagil Kendir, and Michael Cantrell. It has been a pleasure working them and I have learned so much from each of them. I also thank other members and alumni of the Secure Embedded Systems group, Dr. Zhimin Chen, Dr. Abhranil Maiti, Dr. Xu Guo, Mostafa Taha, Ambuj Sinha, Srikrishna Iyer, Aydin Aydsu, and Nahid Farhady, for their moral support and guidance throughout my studies.

Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced encryption standard
API	Application programming interface
BSV	Bluespec System Verilog
CMOS	Complementary metaloxidesemiconductor
CPA	Correlation power analysis
CPU	Central processing unit
DES	Data encryption standard
DFA	Differential fault analysis
DPA	Differential power analysis
DSP	Digital signal processor
DUT	Design under test
ECC	Elliptic curve cryptography
ECDLP	Elliptic curve discrete logarithm problem
FIFO	First in, first out queue
FPGA	Field programmable gate array
FSB	Front side bus
GPU	Graphics processing unit
HDL	Hardware description language
IP	Internet protocol

NIST	National Institute of Standards and Technologies
RAM	Random access memory
RTL	Register transfer level
SCA	Side-channel analysis
SECG	Standards for Efficient Cryptography Group
SHA	Secure hash algorithm
SIMD	Single instruction, multiple data
USB	Universal serial bus

Chapter 1

Introduction

Recent technological advances have sparked proliferation in networked and mobile computing. This has increased the demand for strong information security requirements to protect sensitive data. Cryptographic algorithms ensure confidentiality of data through encryption and are an essential component of secure systems. The security of cryptographic algorithms depends on secrecy of the key used to encrypt data. To ensure adequate security, designers must assume that the attacker knows all details of the cryptographic algorithm and lacks only the value of the secret key [31] [32]. When the attacker discovers the key, he has broken the system and can easily decrypt all past and future messages.

Modern cryptography includes symmetric (private) and asymmetric (public) key cryptographic algorithms. Symmetric key cryptography requires that both the sender and receiver of encrypted data share the same secret key. The most common construction of symmetric key algorithms generates encrypted output, also known as ciphertext, by performing many rounds of computation that combine secret key material with the plaintext using a combination of substitution and permutation operations. Standard symmetric key algorithms include the Advanced Encryption Standard (AES) and Data Encryption Standard (DES). Asymmetric key algorithms, first proposed by Diffie and Hellmen [20], use two different keys: a public key, for encryption, and a private key, for decryption. Each user has a pair of distinct, but mathematically related keys. RSA and Elliptic Curve Cryptography (ECC) are among the most widely used asymmetric key algorithms. The security of asymmetric key cryptography relies on the computational infeasibility of determining the private key using knowledge of the public key. For example, ECC relies on the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP), a hard mathematical problem with exponential solution complexity [40], and discovery of a move efficient solution could render existing ECC-based systems insecure.

It is often very difficult to prove absolute security of a cryptographic algorithm and many algorithms derive security from assumptions on the practical or computational limitations of an attacker. Security vulnerabilities can occur due to discovery of new attack methods,

reduction in the complexity of existing attacks, and improvement in the computational efficiency of known attacks. Since it is not possible to know the true capabilities of malicious attackers, researchers analyze cryptographic algorithms from the attacker's point of view to identify potential weaknesses. While it may not be possible to prevent all vulnerabilities, this research allows the security community to discover attacks on widely used algorithms before they can be exploited by an attacker. Therefore, research on state-of-the-art attack methods, and their computational requirements, is critical to ensure that system designers are aware of security vulnerabilities in cryptographic algorithms. The National Institute of Standards and Technologies (NIST), which establishes national standards for cryptographic algorithms, has recognized the importance of this work to ensure security. During its selection process for the Advanced Encryption Standard (AES) and Secure Hash Algorithm (SHA), NIST actively solicited research on security vulnerabilities in the candidate algorithms.

Cryptanalysis is the field dedicated to studying attacks on cryptographic algorithms. This includes brute force attacks, logical attacks, and implementation attacks. Brute force attacks try all possible values for the secret key in order to break the system with average complexity of 2^{n-1}, where n is the key size in bits. All cryptographic algorithms are vulnerable to these attacks and security against them is achieved by ensuring the key space is large enough that is is impractical for an attacker to perform exhaustive search. Logical attacks exploit mathematical properties of a cryptographic algorithm to derive the secret key with less complexity than brute force attacks. Implementation attacks exploit vulnerabilities in the physical implementation of a cryptographic algorithm, rather than properties of the algorithm itself. Unlike other attack methods, demonstration of a successful implementation attack indicates insecurity in the system design, rather than the cryptographic algorithm itself. A secure algorithm, such as the current encryption standard AES, can still be vulnerable if its physical implementation is insecure. Numerous countermeasures have been proposed to protect the system implementation from attacks.

Cryptanalysis uses highly optimized systems to implement attacks and approximate the capabilities of an attacker. Due to the inherent complexity of the cryptographic algorithms they target, cryptanalytic systems typically require a complex design with many different parameters. Design space exploration is essential to master large, complicated designs, including cryptanalytic systems. The most successful cryptanalytic systems are those that can be easily adapted for such experimentation. However, due to interdependencies between components, minor changes to the system architecture or design space parameters can result in time consuming design changes and verification. As a result, these explorations are often skipped, leaving a possible suboptimal design. In addition, the design process for high performance cryptanalytic systems typically focuses on optimization of a few performance-critical components. However, implementation of the complete system may require additional components, which increase the overall design cost without improving performance. Since the total resources available for system design are limited, this reduces the resources that can be applied to optimize critical elements. Therefore, achieving an optimal cryptanalytic design requires use of solid methodologies to construct the system.

1.1 Contribution

This thesis proposes methods that can be applied at different stages in the design process to facilitate easier construction of complex cryptanalytic systems that achieve high performance. The goal of these improved design methods is to maximize the flexibility of the resulting system for design space exploration, while minimizing the design effort required to implement the system. This allows designers to focus their efforts on optimization of the performance-critical aspects of a complex cryptanalytic engine. The proposed methodologies are demonstrated by describing the design process for two complex cryptanalytic systems and providing measured implementation results for each.

First, the potential improvements to the hardware design process due to changing the hardware description language are investigated in the context of a system to implement a logical attack on ECC. Standard hardware description languages (HDLs), such as Verilog and VHDL, describe behavior at the register transfer level (RTL). The low abstraction level increases the design cost of hardware systems by requiring that designers explicitly express all control of and interactions between system components. Bluespec is used to show how the unique language features raise the abstraction level of hardware design and overcome the pitfalls of traditional HDLs. A generalized high performance architecture for multiplication over prime field is proposed and performance of the architecture is characterized by exploring the design space to determine the optimal integer basis for polynomial representation. Bluespec is also used to explore the system design space by varying multiple parameters of the ECDLP system including batch vector size and number of point addition modules. Measured results are presented for several alternative implementations of a secp112r1 ECDLP in order to demonstrate the impact of system parameters on performance. The optimized system achieves performance that is competitive with prior work, which provides credible evidence that a higher abstraction level will greatly improve hardware design methodologies for complex cryptanalytic systems.

A modular approach to system design is also presented. This approach is applied to construct a test environment for implementation attack vulnerability analysis. The test environment design is focused on providing separation between the device being tested and the test script, as well as portability and openness. This results in a test environment that can evaluate resistance against implementation attacks regardless of the device platform or the type of attack performed. The proposed test environment allows integration of expertise from both system designers and implementation attack experts, is suitable for use in standardized security analysis, and enables fair comparison of the implementation attack resistance of different designs. The proposed environment is used to perform an implementation attack on an embedded microprocessor implementation of AES. Measured results of the attack are presented in order to demonstrate the suitability of the test environment for implementation attack vulnerability analysis of a practical design. Construction of a suitable test setup is often a time consuming part of implementation attack testing. The proposed test environment improves the design process by allowing focus on the most critical elements, such as the

implementation attack technique and design of the device under test.

1.2 Thesis Organization

This thesis is organized as follows.

Chapter 1: Introduction. The thesis begins by introducing the concepts of cryptography and cryptanalysis that are central to this work. The motivation of this research is to identify the current state-of-the-art in cryptanalysis and describe the influence of design methodology on cryptanalytic system performance. This chapter also presents a summary of the two main contributions of this thesis: a hardware-accelerated engine to solve the ECDLP and a modular test environment for implementation attacks.

Chapter 2: The Elliptic Curve Discrete Logarithm Problems. Design of he hardware-accelerated engine to solve the ECDLP requires an understanding of the foundations of elliptic curve cryptography and finite field arithmetic. This chapter provides the background for this work, as well as a summary of prior work on solving the ECDLP.

Chapter 3: Modular Arithmetic Units for a Hardware ECDLP Engine. An ECDLP engine requires primitives to perform finite field modular arithmetic. Custom hardware modular arithmetic units are designed to achieve high performance. In this chapter, the design process for constructing and optimizing these modules is described. A novel architecture for a high performance generalized modular multiplier is presented and applied to perform arithmetic in GF($2^{128} - 3$).

Chapter 4: Design of a Hardware-Accelerated ECDLP Engine in Bluespec. This chapter introduces Bluespec and describes its advantages over traditional HDL design. The ECDLP engine is used as a case study to demonstrate how the key concepts of Bluespec can be used to improve the design process of a complex hardware system. A complete hardware-accelerated engine for solving the ECDLP is constructed using Bluespec and the design process is detailed in this chapter.

Chapter 5: Cryptanalytic Performance of the ECDLP Engine. Design space exploration is critical to achieving a high performance design. This chapter presents measured results for exploration of the impact of various design parameters on overall system performance. After the optimal design configuration is determined, results of this work are compared with previously published solutions to the ECDLP. This shows that the proposed design is the fastest hardware ECDLP engine targeting elliptic curves over a prime field and that a multicore implementation is competitive with software solutions targeting parallelized architectures such as the Cell processor.

Chapter 6: Implementation Attacks. This chapter provides the context for the design of a modular test environment for implementation attacks. Implementation attacks are defined and their key differences from logical cryptanalytic attacks are described. The most common

implementation attack strategies are introduced and the key features of each attack method are identified. The critical design challenges in securing systems against these attacks are also discussed.

Chapter 7: A Modular Test Environment for Implementation Attacks. This chapter presents a solution to the problem of implementation attack vulnerability analysis; a modular test environment. A modular design approach is used to ensure that the test environment is independent of the system under evaluation, as well as the specifics of the target attack. The design is described, as well as the methodologies used to construct it.

Chapter 8: Practical Demonstration of the Modular Test Environment. Here, the proposed test environment is applied to perform a power analysis side-channel attack on an embedded microprocessor implementation of AES. Use of the test environment is illustrated by detailing the test setup used to implement the attack and presenting actual attack results. Application of the test environment to other classes of implementation attacks proposed in the literature is also discussed.

Chapter 8: Conclusion. This chapter draws final conclusions based on the research and results presented in this thesis. The impact of high-level, flexible design methods on cryptanalysis is extrapolated from results presented in this thesis.

1.3 Related Articles

This work has been presented in the following previously published manuscripts.

> S. Mane, L. Judge, P. Schaumont, "An Integrated Prime-field ECDLP Hardware Accelerator with High-performance Modular Arithmetic Units," 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig), December 2011. DOI: 10.1109/ReConFig.2011.12 [39].
>
> L. Judge, P. Schaumont, "A Flexible Hardware ECDLP Engine in Bluespec," Workshop on Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS), Washington, DC, March 2012 [30].
>
> L. Judge, S. Mane, P. Schaumont, "A Hardware-Accelerated ECDLP with High-performance Modular Multiplication," International Journal of Reconfigurable Computing, vol. 2012, Article ID 439021, 14 pages, 2012. DOI:10.1155/2012/439021 [29].
>
> L. Judge, M. Cantrell, C. Kendir, P. Schaumont, "A Modular Testing Environment for Implementation Attacks," Workshop on Redefining and Integrating Security Engineering at ASE/IEEE International Conference on Cyber Security 12 (RISE), December 2012 [28].

Chapter 2

The Elliptic Curve Discrete Logarithm Problem

Elliptic curve cryptography (ECC), independently introduced by Miller [42] and Koblitz [34], is a public key cryptosystem that has now found a significant place in academic literature, practical applications, and security standards. Its popularity is mainly because shorter key sizes offer high levels of security relative to other public key cryptosystems, such as RSA. ECC defines an elliptic curve as the set of points satisfying

$$y^2 = x^3 + ax + b \tag{2.1}$$

along with the point at infinity. These points form an abelian group under addition with the point at infinity as the additive identity. When the elliptic curve is defined over a finite field, the set of points is necessarily finite and, together with the point at infinity and the point addition operation, forms a cyclic group. The total number of points on the curve is defined as its order l. An elliptic curve is characterized by the public domain parameters a and b, a public generator point G of prime order, and a finite field GF(p) or GF(2^m). Standard curves have been defined by NIST [46] and the Standards for Efficient Cryptography Group (SECG) [55]. Each user of ECC choses a private key n, a random scalar such that $0 \leq n < l$, and computes a public key P, where $P = [n]G$ and $[n]$ denotes the scalar multiplication with n.

This chapter defines the ECDLP and introduces the Pollard rho method, the fastest known solution [27]. Optimizations to the Pollard rho algorithm are discussed, along with the underlying modular arithmetic operations required to solve the ECDLP.

2.1 Definition of the ECDLP

The security of ECC relies on the difficulty of Elliptic Curve Discrete Logarithmic Problem (ECDLP) [8], which is defined as follows. Let p be a prime and $\mathbb{F}_p = \mathrm{GF}(p)$. Given the elliptic curve E over $\mathbb{F}_p$ of order $l = |E(\mathbb{F}_p)|$, let $S \in E(\mathbb{F}_p)$ be a point of order l. Solving the ECDLP requires finding the private key, an integer n, given the public key P and the generator point $G \in \langle S \rangle$ such that

$$P = [n]G. \tag{2.2}$$

Because points on an elliptic curve defined over a finite field have no known mathematical relationship other than through point addition, it is very difficult to compute the inverse of the scalar multiplication, also know as the discrete logarithm, and find n.

2.2 Pollard Rho Algorithm

The Pollard rho algorithm [50] uses a pseudorandom iteration function $f : \langle S \rangle \to \langle S \rangle$ to solve the ECDLP. It conducts a pseudorandom walk by starting from a random seed point on the curve, $X_0 = a_0P + b_0G$ for random $a_0, b_0 \in \mathbb{Z}$, and generating subsequent points using the iteration function $X_{i+1} = f(X_i)$. Since the points on the curve form a closed group under addition, the iteration function will eventually produce the same point twice, resulting in a cycle.

The name of the algorithm, rho, expresses the Greek letter ρ, which shows a walk ending in a cycle. Cycles can be efficiently detected using Floyd's cycle-finding method or Brent's cycle-finding algorithm [13]. The collision point, which gives the solution to the ECDLP, is located at the starting point of the cycle. Therefore the underlying idea of this algorithm is to search for two distinct points on the curve such that $f(X_i) = f(X_j)$. Due to the birthday paradox, assuming a random iteration function, the expected number of iterations to find collision is $\sqrt{\frac{\pi \cdot |\langle S \rangle|}{2}}$ [50].

The effectiveness of the Pollard rho method depends on the randomness of the iteration function. Several variations of iteration functions have been proposed and these proposals have been analyzed by Teske [58] [57] and Bos et al [12]. Teske proposes an additive iteration function

$$f(X_i) = X_i + R_i(X_i), \tag{2.3}$$

where R_i is an index function $\langle S \rangle \to \{0, 1, ..., r-1\}$ and each element $R_j = a_jP + b_jG$, for random values $a_j, b_j \in \mathbb{Z}$ [58]. Based on analysis by Teske [58] [57] and Bos et al [12], the additive iteration function is more similar to a truly random walk than Pollard's original function and other proposed variants. Furthermore, an additive walk with $r \geq 16$ is very close

to a true randomness and achieves speedup of 1.25X over Pollard's iteration function [57]. Thus, this work performs an additive walk and chooses $r = 16$.

During an additive walk, a collision occurs when two points are found such that

$$X_i + R_i = X_j + R_j$$

Based on the definition of the index function R_i, the collision points can be rewritten as

$$c_i P + d_i G = c_j P + d_j G,$$

where $c_i = \sum_{k=0}^{i} a_k$ and $d_i = \sum_{k=0}^{i} b_k$. The solution to the ECDLP can then be obtained as

$$n = \left[\frac{c_i - c_j}{d_j - d_i}\right] \bmod l \tag{2.4}$$

2.2.1 Parallelization

Van Oorschot and Wiener [59] describe a technique that allows use of parallel Pollard rho walks to accelerate computation of the ECDLP. The idea is to define a subset of $\langle S \rangle$ as distinguished points, points which have some distinguishing property. Each parallel walk starts from a distinct random seed point of the form $X_0 = a_0 P + b_0 G$ and continues a Pollard rho walk until it encounters a point that satisfies the distinguishing property. Due to the properties of the index function of the additive iteration function, once a collision occurs between two parallel walks, the walks will continue to be identical. Therefore, checking only distinguished points for collisions reduces search overhead, while ensuring that all collisions between walks are detected.

Whenever a distinguished point is found, it is transmitted to a server along with the random seed point that generated the walk. The server maintains a database of all distinguished points received from the clients. When a new distinguished point is received, the server searches the database to determine whether the same distinguished point has been generated from a prior walk. Once a collision is detected, the server derives the secret key using the collision point and the distinct seed points of the colliding walks. The parallel Pollard rho method distributes random walks among multiple processing clients and shares all distinguished points found with a central server that performs a collision search. This technique results in a linear speed up as the number of clients increases.

The expected number of distinguished points required to find a collision is a fraction of the expected path length. This depends on the density of distinguished points in a point set $\langle S \rangle$, which in turn depends on the chosen distinguishing property. Consider a distinguishing property defined as a point with d-bits fixed in its y-coordinate. This results in a probability of 2^{-d} that a given point is a distinguished point and $\frac{|\langle S \rangle|}{2^d}$ total distinguished points on the

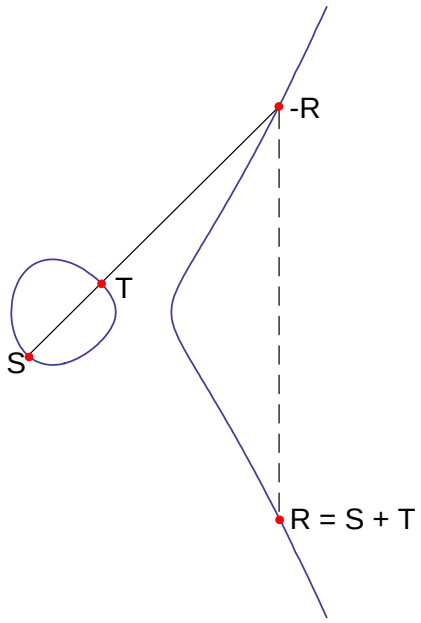


Figure 2.1: Geometric representation of elliptic curve point addition

curve. Each walk will require 2^d steps on average to find a distinguished point. Based on the birthday paradox, the number distinguished points required to find a collision is

$$\phi \geq \frac{\sqrt{\frac{\pi \cdot |\langle S \rangle|}{2}}}{2^d}. \tag{2.5}$$

Since each distinguished point is generated from a walk with a unique random seed point, the number of parallel walks required to solve the ECDLP is also ϕ.

2.3 Point Addition

Each iteration of the Pollard rho algorithm requires a point addition of the current point, X_i, with a precomputed combination of P and G, $R_i(X_i)$. Addition of two distinct elliptic curve points for an elliptic curve, $S + T = R$, is defined geometrically as shown in Figure 2.1. To compute the addition, a line is drawn through the two points S and T. The line intersects exactly one other point of the elliptic curve. The intersection point is $-R$, which can be reflected across the x-axis to find R.

Algebraically, adding two points $S = (x_S, y_S)$ and $T = (x_T, y_T)$ gives the sum $R = (x_R, y_R)$,

Table 2.1: Arithmetic operations for point addition: S + T = R

Operation	**Function performed**
Modular Subtraction	$t1 = y_S - y_T$
Modular Subtraction	$t2 = x_S - x_T$
Modular Inversion	$t2 = t2^{-1}$
Modular Multiplication	$t4 = t2 \cdot t1 : \lambda$
Modular Squaring	$t1 = t4 \cdot t4$
Modular Addition	$t3 = x_S + x_T$
Modular Subtraction	$t1 = t1 - t3 : x_R$
Modular Subtraction	$t2 = x_R - t1$
Modular Multiplication	$t3 = t2 \cdot t4$
Modular Subtraction	$t3 = t3 - y_S : y_R$

where

$$x_R = \lambda^2 - (x_S + x_T), \tag{2.6}$$

$$y_R = \lambda(x_S - x_R) - y_S, \tag{2.7}$$

$$\lambda = \frac{y_S - y_T}{x_S - x_T}. \tag{2.8}$$

For an elliptic curve defined over a finite field GF(p), the geometric relationship between points shown in Figure 2.1 does not exist. The algebraic computation of point addition remains the same, with all arithmetic performed as operations in GF(p). Each point addition requires 2 multiplications modulo p, 1 modular squaring, 5 modular subtractions, 1 modular addition, and 1 modular inversion. The sequence of arithmetic operations for a point addition is shown in Table 2.1.

By Fermat's Little Theorem, $z^{-1} = z^{p-2}$ for $z \in$ GF(p). Thus, computing modular inversion requires modular exponentiation, making inversion the most computationally expensive operation of the point addition. Optimization at the system level can reduce the cost of inversion through the use of Montgomery's trick [43]. Montgomery's trick makes it possible to share the cost of an inversion among M computations by performing inversion on a vector of M points simultaneously. Since the computational cost of inversion is two orders of magnitude greater than other arithmetic operations, this results in significant savings over computation of M individual modular inversions. Montgomery's trick is applied to the

ECDLP system by implementing a vectorized point addition datapath that computes M random walks on a single point addition core and shares the inversion cost across all walks.

Despite optimizations, inversion remains an expensive operation. Since exponentiation is achieved using multiplication, the cost of inversion is directly tied to the cost of modular multiplication. Therefore, design of an efficient modular multiplier is an important aspect of the system to maximize overall performance.

2.4 Field Arithmetic in GF(p)

Since ECC uses elliptic curves defined over a finite field, all point operations, including point addition, are computed using finite field arithmetic. Addition and multiplication, as well as their inverses, subtraction and division, are defined within the field. All arithmetic operations within the field GF(p) take their result modulo p.

Standards for elliptic curve cryptography have been defined for curves over both binary and prime fields [46] [55]. Binary fields can be represented with a *polynomial basis* or *normal basis*. In general, hardware implementations favor polynomial basis because it allows simplified reduction through the use of an irreducible polynomial of the form $x^{m-1}+...+x^2+x^1+x^0$ [27]. Prior work has confirmed the idea that binary field curves are better suited for hardware platforms. A comparison of ECDLP engines by [41] shows a significant speedup for binary field ECDLP over prime field ECDLP implemented on an FPGA platform. The performance discrepancy between ECDLPs for binary field and prime field curves is due to differences in the required arithmetic operations for each field and the use of search optimization techniques for selected curves.

The Pollard rho method is the most efficient known attack on ECC for curves defined over both binary and prime fields, but the cost of the attack varies based on the properties of the finite field arithmetic. Elliptic curves defined over prime fields require conventional integer arithmetic operations followed by costly reduction modulo p. However, the properties of curves defined over GF(2^m) allow optimizations that can reduce the cost of point addition. Additions and subtractions in the binary finite field are reduced to XOR operations with no carry and cost of modular reduction is reduced to parallel XOR gates. For comparison, reduction modulo p requires sequential additions of carry bits and has a cycle cost approximately equal to that of multiplication. Although attacks on curves over GF(2^m) can be considerably faster than attacks on curves over GF(p), NIST security standards recommend approximately equivalent field sizes for both binary and prime field curves at a given security level [46].

This work targets secp112r1, a standard 112-bit elliptic curve defined over GF($\frac{2^{128}-3}{76439}$) [55]. Prime field arithmetic is performed using a redundant 128-bit polynomial representation in an affine coordinate system, as proposed by Bernstein et al. [5] and Bos et al. [11]. The integers in secp112r1 are represented redundantly in the ring $R = \mathbb{Z}/q\,\mathbb{Z}$, where $q = p * 76439$. This

allows reduction to be performed modulo $2^{128}-3$, rather than modulo $\frac{2^{128}-3}{76439}$. All arithmetic is performed over the 128-bit field using following method.

Reduction for the unbalanced coefficient $q = 2^{128} - 3$ requires a constant multiplication and an addition. Assuming $A = A_1 \cdot 2^{128} + A_0$, then $A \bmod q = A_0 + 3 \cdot A_1$. Integers represented redundantly mod q can be converted into a canoncial form in GF(p) by multiplying with $a = 76439$. Let $a|q$ and $p = q/a$. Then $v \bmod p \equiv v \cdot a \bmod q$. Therefore, starting with a unique representation in GF(p), arithmetic is performed in R and the results are canoncialized in R to a unique representation by multiplying with $a = 76439$. Similar redundant representation can be used to simplify reduction for any prime $p = (2^n - m)/k$, where $k \neq 0$, by performing arithmetic modulo $q = 2^n - m$ and canonicalizing results by multiplication with k.

2.5 Related Work

The software solution proposed by Bernstein et al. on Cell CPUs is the fastest existing solution to the ECDLP for the secp112r1 elliptic curve [5]. It uses the negation map and non-integer polynomial-basis arithmetic to report a speedup over a similar solution by Bos et al. [11]. Both of these software solutions use prime field arithmetic in an affine coordinate system exploit the SIMD architecture and rich instruction set of the Cell CPU. Another software solution by Bos et al. [10] describes the implementation of parallel Pollard rho algorithm on the Cell CPU that attempts to solve the ECC2K-130 Certicom challenge. High performance ECDLP solutions using GPU platforms have also been proposed in [4].

Fan et al. propose use of normal-basis, binary field multiplication to implement a high-speed attack on ECC2K-130 [21] using Spartan-3 FPGAs. This solution outperforms attacks on the same curve using GPUs [4] and Cell CPUs [10], demonstrating the suitability of FPGA platforms for solving large binary field ECDLPs. Another binary field solution uses the COPACOBANA FPGA cluster [37] to target a 160-bit curve [23]. Güneysu et al. propose an architecture to solve ECDLP over prime fields using FPGAs and analyze its estimated performance for different ECC curves ranging from 64 to 160 bit fields [24].

Bulens et al. propose an FPGA solution to attack the ECC Certicom challenge for GF(2^{113}) [41]. Though they discuss the hardware-software integration aspects of the solution, the authors did not confirm whether their system is operational. Güneysu et al. discuss some aspects of system-level integration for their prime-field ECDLP system [24]. A three-layer hybrid distributed system is described by Majkowski et al. to solve ECDLP over binary field [38]. It uses the general purpose computers with FPGAs and integrates them with a main server at the top level.

Chapter 3

Modular Arithmetic Units for a Hardware ECDLP Engine

A Pollard rho step corresponds to a point addition in GF(p) that consists of four subtractions, one addition, four modular multiplications, and one inversion. This requires design of modular multiplication, modular addition/subtraction, and modular inversion units. Subsequent sections explain the architecture of these arithmetic modules in detail.

3.1 Modular Multiplication

In this section, a novel architecture for modular multiplication in a prime field is presented. Typically, hardware solutions use binary field arithmetic, primarily due to the assumption that binary field avoids the costly carry propagation of prime field arithmetic. However, the proposed design demonstrates that prime field arithmetic can also be efficiently implemented in hardware.

This work targets the 112-bit secp112r1 elliptic curve in GF($\frac{2^{128}-3}{76439}$), but demonstrates that the proposed architecture can be generalized to perform multiplication modulo any prime of the form $(2^n - m)/k$.

3.1.1 Algorithm

Bernstein et al. [5] use a non-integer basis for polynomial representation of data to achieve an efficient software implementation. In this work, an integer basis representation is chosen to make the partial product computation uniform across all coefficients, which also makes the design scalable over larger fields. Each l-bit operand is represented as $\sum_{i=0}^{n_a-1} x_i \cdot 2^{i \cdot l_a}$, where

$n_a = l/l_a$, l is the length of the operand in bits, and l_a is the integer basis. For the secp112r1 curve, $l = 128$ and optimization of the quantity l_a is discussed in Section 5.5

The approach to modular multiplication presented in this work is based on the design proposed by Güneysu and Paar in [22]. The proposed algorithm uses schoolbook multiplication to compute $R = F \cdot G$, for F, $G \in \mathrm{GF}(q)$. Schoolbook multiplication requires computation of n_a^2 partial products in $O(n^2)$ time, where

$$R = \sum_{i=0}^{2n_a} 2^{i \cdot n_a} \sum_{j=0}^{i} F_j \cdot G_{i-j} \tag{3.1}$$

Computation of the partial products is parallelized to perform n_a multiplications in parallel by multiplying one field of one operand, G, with all fields from the other operand, F. As proposed by Güneysu and Paar in [22], placing G in a shift register and F in a rotating register ensures that each multiplier produces aligned results, that can be directly accumulated. The proposed design adds an important optimization of the reduction step. Reduction modulo q is integrated into the multiplication by multiplying the most significant field of F by m when rotating it to the least significant field. In order to accommodate the multiplication by m of each field of F, each field of F is represented with $l_a + \log_2 m$ bits.

Each partial product is an $l_a + \log_2 m \times l_a$-bit multiplication and produces a $2l_a + \log_2 m$ bit result. After accumulation, each field of the result can be up to $2l_a + \log_2 m + \log_2 n_a$ bits. Each field is reduced to l_a bits using two parallel carry chains. The first carry chain is integrated into partial product accumulation and accumulates the lower l_a bits of partial product i with the upper bits of partial product $i-1$. The second carry chain adds the lower bits of each accumulated partial product in column i to the upper bits of the accumulated partial product in column $i-1$. In both carry chains, reduction modulo q is achieved by multiplying the upper bits of the most significant field, $i = n_a - 1$, by m before adding them to the least significant field, $i = 0$. The final l-bit result is obtained by concatenating the l_a-bit reduced outputs. The parallel carry chains overlap the multiplication and reduction stages of the the algorithm to achieve significantly lower latency than reported in prior work [22].

3.1.2 Hardware Architecture

Figure 3.1 depicts the hardware architecture used to implement modular multiplication for the secp112r1 elliptic curve with general field size parameters $l = 128$, l_a, and n_a. This architecture targets a Xilinx Virtex 5 FPGA platform and makes use of dedicated DSP blocks for high performance arithmetic. Each DSP block includes a 25x18-bit multiplier and the FPGA fabric provides dedicated routing paths for high-speed connections between DSP blocks [62]. The DSP blocks allow computation of each $l_a + 2 \times l_a$-bit partial product multiplication and l_a+4-bit accumulation in a single clock cycle. The modular multiplication takes two 128-bit inputs, F and G, divided into n_a l_a-bit fields and produces a 128-bit output

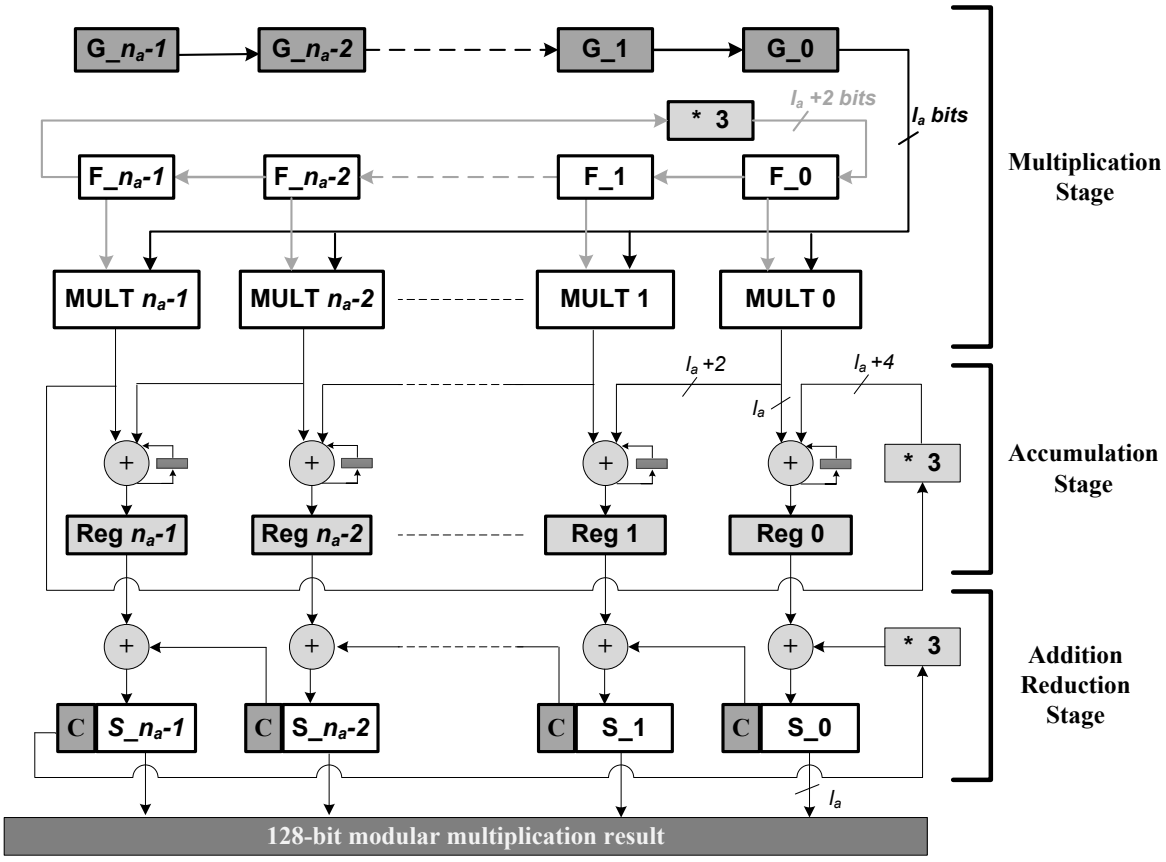


Figure 3.1: Modular multiplication architecture

of the product reduced modulo $2^{128}-3$. There are n_a multipliers required to compute partial products of the l_a-bit coefficients. Since the n_a^2 partial products are required, it takes n_a multiplication cycles to compute the full unreduced multiplication result.

Reduction modulo $q = 2^{128} - 3$ adds to the cycle cost of a modular multiplication. By multiplying the rotating operand F_{l_a-1} by 3, reduction is performed in parallel with the multiplication. For the 128-bit field with $l_a = 16$ and $n_a = 8$, the architecture requires eight multiplier-adder columns and takes eight cycles of multiplication and 12 iterations of reduction to obtain the reduced product. This results in a total cost of 20 cycles per modular multiplication. However, the proposed algorithm reduces this cost by overlapping reduction with partial product multiplication and accumulation using the previously described parallel carry adder chains. Thus, the cost has been reduced to 14 cycles, a significant improvement in latency over the architecture proposed by Güneysu and Paar in [22], which takes 70 clock cycles for 256-bit modular multiplication.

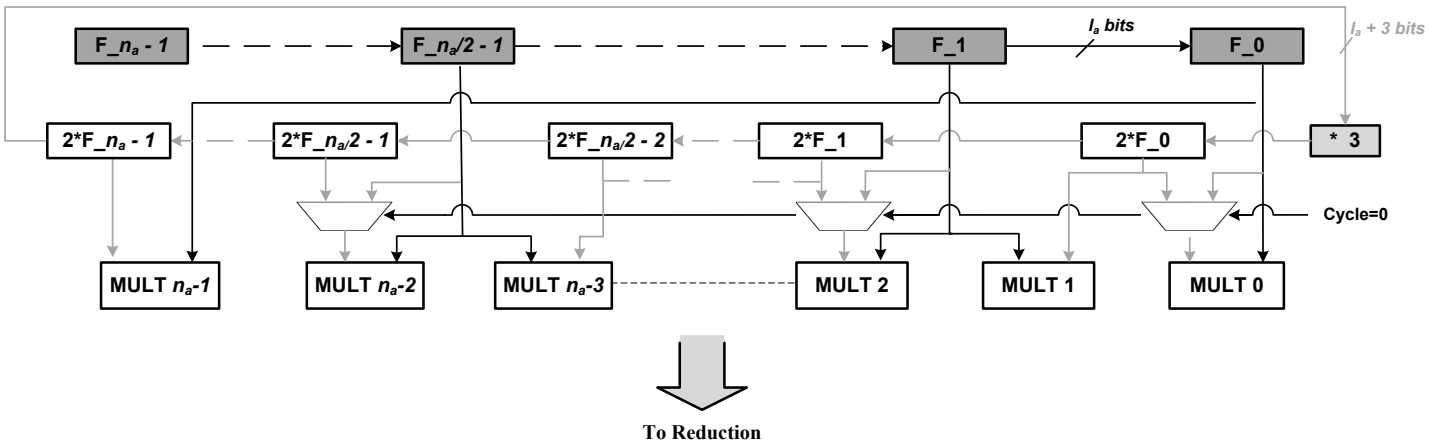


Figure 3.2: Dedicated squaring architecture

3.2 Addition and Subtraction

An integer basis polynomial representation is used to perform addition and subtraction modulo q using high-speed DSP blocks in the FPGA. A field size of 16 is selected, which allows use of eight parallel DSP adder/subtracters to compute the sum of the two 128-bit operands. This design requires one cycle for the addition/subtraction and one additional cycle for reduction modulo q.

3.3 Squaring

An inversion involves a total of 137 modular multiplications, 75% of which are squaring operations. A dedicated squaring unit allows specific optimizations that take advantage of the properties of squaring and reduces the time required to compute an inversion. Consequently, a dedicated module optimized for squaring provides significant acceleration of the point addition operation.

Squaring is a special case of the general multiplication of two 128-bit operands F and G. During squaring $F = G$ and all off-diagonal partial products, represented as $F_i \cdot F_j$ for $i \neq j$, are computed twice by the conventional schoolbook multiplication method used in the modular multiplier. To reduce the computational cost of squaring, the operand control structure of the multiplier design is altered to avoid computing any partial product more than once. The operand control structure for the dedicated squaring module is shown in Figure 3.2.

The operand F is represented using the same integer basis for polynomial representation previously described for the modular multiplier. F is copied into a shift register and a rotating register, each with n_a l_a-bit fields for polynomial representation, to achieve aligned accumulation of partial products. As shown in Figure 3.2, the first cycle of multiplication computes the partial products on the diagonal, that is $F_i * F_j$ where $i = j$. All remaining partial products are off-diagonal. Since off-diagonal partial products must be computed twice, each field of the rotating register is multiplied by 2 before computing these partial products. The multiplication by 2 is implemented as a bit shift left at negligible cost. This reduces the number of partial product multiplications from n_a^2 to $\frac{n_a(n_a+1)}{2}$, which is a 1.78X improvement for $l_a = 16$. The accumulation and addition reduction stages for the squaring module are identical to those used in multiplication.

3.4 Inversion

From Fermat's little theorem, it follows that the modular inverse of $z \in \mathbb{F}_p$ can be obtained by computing z^{p-2}. Therefore, computation of the modular inverse requires exponentiation to the $p-2$ power, which is achieved through successive square and multiply operations. For the secp112r1 curve, the inverse of z is computed as $z^{\frac{2^{128}-3}{76439}-2}$. Exponentiation is computed using a variant of the left to right binary method, also known as square and multiply method [40]. This method scans a binary representation of the exponent left to right considering one bit at a time. A squaring is performed for each bit in the exponent and an additional multiplication by the input operand is performed whenever the current bit is 1. The basic algorithm is shown as Algorithm 1. Due to dependencies between each operation, parallelization of the algorithm is not possible; it needs 112 squarings and 59 multiplications to compute an inversion for the secp112r1 curve. These figures are improved by applying specialized techniques to optimize the inversion operation.

Algorithm 1 Square and multiply modular exponentiation [40]

Input: $z \in \mathbb{F}_p$, $e[t:0]$
Output: $z^e \bmod p$
  $r \leftarrow 1$
  for $i = t \rightarrow 0$ **do**
    $r \leftarrow r \cdot r \bmod p$
    if $e[i] = 1$ **then**
      $r \leftarrow r \cdot z \bmod p$
  return r

3.4.1 Windowing Optimization

The sliding window method [40] is applied to reduce the number of squarings and multiplications required for inversion. The windowing method is an optimization of the square and multiply algorithm that considers multiple bits of the exponent simultaneously. Computing z^{p-2} with window size k requires precomputation of z^2, as well as the first $2^{k-1}-1$ odd powers of z. This method considers up to k bits of the exponent in each iteration and performs a multiplication with one of the precomputed powers of z based on the value of those bits. The sliding window method for modular exponentiation is shown as Algorithm 2.

Algorithm 2 Sliding window modular exponentiation [40]

Input: $z \in \mathbb{F}_p$, $e[t:0]$, $k \geq 1$
Output: $z^e \bmod p$
  // Precomputation
  $q_0 \leftarrow z$, $s \leftarrow z \cdot z \bmod p$
  for $i = 1 \rightarrow 2^{k-1} - 1$ **do**
    $q_i \leftarrow q_{i-1} \cdot s$
  // Inversion
  $r \leftarrow 1$
  $i \leftarrow t$
  while $i \geq 0$ **do**
    if $e_i = 0$ **then**
      $r \leftarrow r \cdot r \bmod p$
      $i \leftarrow i - 1$
    else
      Find longest string $e[i:l]$ where $e[l] = 1$ and $i - l + 1 \leq k$
      $r \leftarrow r \cdot q_{e[i:l]} \bmod p$
      for $j = 0 \rightarrow (i - l + 1)$ **do**
        $r \leftarrow r \cdot r \bmod p$
      $i \leftarrow i - l - 1$
  return r

Optimal performance is achieved with a window size of four, which allows inversion with only 108 squarings and 29 modular multiplications. This results in a total of 137 operations, rather than 171 operations without windowing, an improvement of 1.25.

3.4.2 Vectorization Optimization

Montgomery's trick [43] is used to further reduce the cost of an inversion by allowing multiple modular inverses to be computed together for significant latency savings. Montgomery's trick is based on the observation that given $(z_1 \cdot z_2 \cdot ... \cdot z_M)^{-1}$, the individual inverses $z_1^{-1}, z_2^{-1}, ..., z_M^{-1}$

Algorithm 3 Vector inversion using Montgomery's trick [43]

Input: $z_1, z_2, ...z_M \in \mathbb{F}_p$
Output: $z_1^{-1}, z_2^{-1}, ...z_M^{-1} \in \mathbb{F}_p$
// Preprocessing
$x_1 \leftarrow z_1$
for $i = 2 \rightarrow M$ **do**
$x_i \leftarrow x_{i-1} \cdot z_i \bmod p$
// Inversion
$y_M \leftarrow x_M^{-1}$
// Postprocessing
for $i = M - 1 \rightarrow 1$ **do**
$y_i \leftarrow y_{i+1} \cdot z_{i+1} \bmod p$
$z_{i+1}^{-1} \leftarrow y_{i+1} \cdot x_i \bmod p$
$z_1^{-1} \leftarrow y_1$
return $z_1^{-1}, z_2^{-1}, ...z_M^{-1}$

can be easily computed. This allows computation of M modular inverses using $3(M-1)$ multiplications and one inversion. Although the cost of a single inversion is two orders of magnitude higher than multiplication, this approach allows the cost of inversion to be shared across M operations. For a large vector size M, this results in a marginal cost for inversion that is comparable to that of other arithmetic operations required for point addition. The algorithm for vector inversion using Montgomery's trick is shown as Algorithm 3.

This method is applied by implementing a vectorized inversion module that applies Algorithm 3 for any vector size M. To take advantage of vectorized inversion, the ECDLP engine uses a vectorized point addition datapath that performs random walks on a vector of points simultaneously. A vector size of 32 yields a speedup of 19X over 32 individual inversions. Optimization of the vector size is discussed in Chapter 5.

Chapter 4

Design of a Hardware-Accelerated ECDLP Engine in Bluespec

Bluespec is a high level hardware description language that aims to improve the hardware design process. This chapter introduces the key concepts of Bluespec, describes the main differences between Bluespec and traditional HDLs, and demonstrates the use of Bluespec to design a flexible hardware ECDLP engine. Several factors, including separation of interface from behavior and flexible specification of rule-based control, seem to confirm the suitability of Bluespec for complex design tasks like an ECDLP. By using Bluespec, the proposed design achieves competitive performance and is easily adaptable for design space exploration.

4.1 Bluespec

Bluespec [49] is an electronic system level hardware synthesis toolset that claims faster and more accurate hardware design at a higher abstraction level than HDLs. Bluespec is centered on the Bluespec System Verilog (BSV) hardware description language. BSV introduces several unique programming constructs to raise the abstraction level of the hardware design process without sacrificing the designer's control over system microarchitecture or resource usage. In addition to BSV, Bluespec provides a compiler and simulator to allow generation of fully synthesizable Verilog and cycle-accurate simulation of BSV designs.

Bluespec is an attractive alternative to HDLs for hardware design. Bluespec supports a higher abstraction level, which decreases design time and allows design effort to be focused on microarchitecture design and optimization. Since BSV separates interfaces from modules, designers can define generic interfaces for use by multiple lower level modules. Once all interfaces have been defined, the designer can define modules in any order independent of their position in the design hierarchy. Rule-based behavioral specification avoids the need for designers to explicitly define all control logic required for the system. As long as

the structure of each rule is functionally correct, the Bluespec compiler will ensure correct system-level behavior through rule scheduling. Hardware designs expressed using Bluespec can be designed more quickly and modified more easily than equivalent designs expressed in HDL.

Bluespec designs use a module hierarchy similar to that of HDL designs, but they are constructed differently. Each module includes a datapath, behavior, and an interface. The datapath consists of instantiations of lower level components used in the module such as wires, registers, and user-defined modules. The behavior of a module is defined using rules and methods to control data manipulation by datapath elements. The interface encapsulates the input and output behavior of the module by means of methods. Two key elements of Bluespec not found in classic HDL are interface methods and rules.

Interface methods define all input and output operations that can be performed by a module. Methods are classified into three types: Action methods that cause state changes, Value methods that return data values, and ActionValue methods that both cause state changes and return data. Action and ActionValue methods accept zero or more arguments as inputs to the module, while Value and ActionValue methods return a value as module output. The Bluespec compiler determines the implicit execution conditions for each method and adds the appropriate control signals to the design to enforce these conditions. This allows designers to ensure correct behavior without explicitly defining the handshaking logic required to synthesize that behavior in hardware.

Behavior of Bluespec modules is expressed using rules. A rule consists of one or more actions guarded by a Boolean condition. Each rule is executed atomically, which guarantees that all actions in the rule either execute simultaneously in parallel or do not execute. An advantage of atomicity is that each rule can be evaluated individually at any given point in time to determine how the state of the module is impacted. Designers can define each rule individually without explicitly specifying the relationships between rules in the module. Designers must ensure that all actions required to execute in the same cycle are included in a single rule and that no rule contains conflicting actions. As long as these conditions are satisfied, the Bluespec compiler will determine a schedule of rule execution that preserves the functional behavior of the design. The compiler also prevents conflicting rules from firing at the same time; the order of execution for conflicting rules can be manually specified by the designer or automatically assigned by the compiler based on overall schedule requirements of the design.

In HDLs, it is common practice to use a ready signal to indicate completion of a multicycle operation. BSV provides the specialized *Maybe* datatype to encapsulate a ready signal and result value into a single type. Rather than maintaining the control as a separate signal from the data, designers can assign both simultaneously by either tagging the Maybe-typed variable invalid or assigning a data value. To facilitate the use of the Maybe type, the Bluespec library includes methods to read data from and check the validity of Maybe-typed variables.

4.2 Designing the Modular Multiplier in Bluespec

The properties of Bluespec are illustrated by presenting a Bluespec implementation of the modular multiplier described in Section 3.1. To aid this discussion, relevant code segments are included in the text and the complete code listing is provided in Appendix A.

4.2.1 Design Hierarchy

The hierarchy of the Bluespec design is based on the block diagram in Figure 3.1. Low level components required by the modular multiplier are multipliers, accumulators, and adders. Each component receives two input operands and produces a result after one clock cycle. In BSV, the input operands are provided via an Action method that accepts two arguments and the output is returned via a Value method. Since the result is computed in one clock cycle, no explicit control or rules are required.

The single-cycle 18x16-bit multiplier is shown in Listing 4.1. The multiplier interface includes two methods; *load*, which provides the input operands and *result*, which returns the product. The interface methods are defined within the multiplier module as shown. Operand multiplication is performed immediately during a load. The result is stored in the *product* register and returned by the *result* method. Note that appending a zero-bit at the most-significant-bit ensures an unsigned multiplier. Since this is a single cycle design, no additional control is required to define the module behavior. The pragmas *synthesize* and *always_ready* control mapping of the Bluespec design into Verilog. The *synthesize* pragma directs the compiler to generate a separate Verilog module, rather than inlining instantiations of the BSV module. The *always_ready* pragma prevents generation of an additional ready signal for the specified interface methods.

Listing 4.1: Single cycle 18x16 multiplier

```
interface Multiplier;
  method Action load(Bit#(18) f, Bit#(16) g);
  method Bit#(34) result();
endinterface

(* synthesize *)
(* always_ready = load, result *)
module mkMultiplier(Multiplier);
  Reg#(Bit#(34)) product <- mkReg(0);

  method Action load(Bit#(18) f, Bit#(16) g);
    product <= {0, f} * {0, g};
  endmethod

  method Bit#(34) result();
    return product;
  endmethod
endmodule
```

The low level multipliers and accumulators make use of the high-speed DSP blocks in the Virtex-5 FPGA to complete the multiplications and partial product accumulations in a single clock cycle.

4.2.2 Interface

The interface of the modular multiplier, shown in Listing 4.2, is fairly straightforward; the multiplier receives two 128-bit input operands and produces a 128-bit output 14 clock cycles later. The *load* method provides the two 128-bit operands to the module and begins the modular multiplication computation. The *result* method returns the final products using the Maybe datatype. Since modular multiplication is a multiple clock cycle operation, the Maybe-typed return value can only be read when the result is valid, 14 clock cycles after the operands are received.

Listing 4.2: Modular multiplier: Interface

```
interface ModMul;
  method Action load(Bit#(128) f_in, Bit#(128) g_in);
  method Maybe#(Bit#(128)) result();
endinterface
```

4.2.3 Control

Control logic for the modular multiplication is implemented using rules. The atomicity of BSV rules allows rules for each computation stage to be designed independently, relying on the Bluespec compiler to handle scheduling of and interactions between rules. Computation of the modular multiplication is controlled by rules *do_mult*, *do_acc*, and *do_reduce*. These rules load the low level arithmetic components to perform the multiplication, accumulation, and addition reduction stages of the algorithm. An additional rule *wrap* implements the multiplication by three required to reduce the carry bits from the leftmost multiplier-adder column modulo $2^{128} - 3$. None of these rules uses a Boolean condition to guard execution; they execute every clock cycle without any concept of control state.

As an example, the rule used to control the multiplication computation stage is shown in Listing 4.3. As shown, the *do_mult* rule uses the low-level 18x16-bit multipliers $m0$ to $m7$ to compute the partial products of the operands and performs the rotate and shift operations on f and g.

Listing 4.3: Modular multiplier: Multiplication stage

```
rule do_mult;
  f <= {f[125:0], mod_f[17:0]};
  g <= g >> 16;
  m0.load(f[17:0], g[15:0]);
  m1.load(f[35:18], g[15:0]);
  m2.load(f[53:36], g[15:0]);
  m3.load(f[71:54], g[15:0]);
  m4.load(f[89:72], g[15:0]);
  m5.load(f[107:90], g[15:0]);
  m6.load(f[125:108], g[15:0]);
  m7.load(f[143:126], g[15:0]);
endrule
```

The validity of the Maybe-typed result is controlled using an enabled counter that assigns a valid output value after 14 clock cycles. Listing 4.4 shows the rules used to control assignment of final product of the modular multiplication. The rule *incr_counter* controls the cycle counter, which is incremented every clock cycle until it reaches 15. During the fourteenth clock cycle, the output is tagged valid and assigned the concatenated value of the reduced 16-bit coefficients by rule *assign_res*.

Listing 4.4: Modular multiplier: Result assignment

```
1  rule incr_counter (rcnt < 4'd15);
2    rcnt <= rcnt + 1;
3  endrule
4
5  rule assign_res;
6    if (rcnt == 4'd14)
7      cout <= tagged Valid ({c7.result()[15:0], c6.result()[15:0],
           c5.result()[15:0], c4.result()[15:0], c3.result()[15:0],
           c2.result()[15:0], c1.result()[15:0], c0.result()[15:0]});
8    else
9      cout <= tagged Invalid;
10 endrule
```

The Bluespec compiler derives an execution schedule for all rules in the module based on explicit and implicit conditions of each. The explicit conditions are the Boolean conditions used to guard rule execution. Implicit rule conditions are derived by the Bluespec compiler based on the handshaking conditions and logic required implement rules and interface methods. For example, rule *do_mult* does not have any explicit condition, but it both reads and writes operand registers f and g. The load method also writes to f and g, storing the input arguments as the operands. This produces an implicit condition on *do_mult*. Implicit conditions can prevent a rule from executing in the same cycle as a conflicting rule or method. In general, the Bluespec compiler always assigns higher priority to interface methods than module rules, so rules that conflict with a method will not execute when the method is called, regardless of the explicit rule condition.

4.3 System Architecture

This section presents a full system to attack secp112r1 and describes its design using Bluespec. The proposed design exploits the high abstraction level supported for Bluespec designs to implement a robust system that is easily adaptable to the architectural changes required

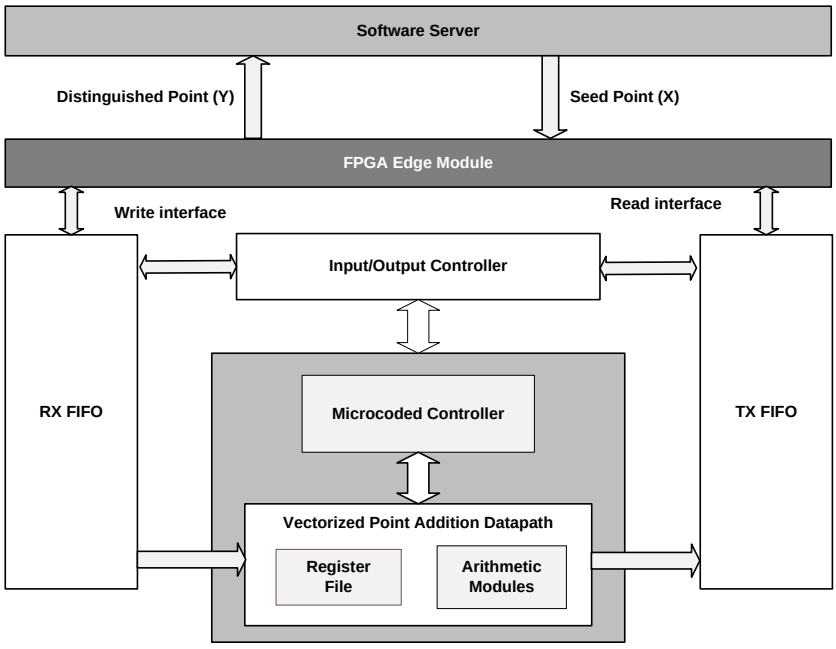


Figure 4.1: Block diagram of microcoded ECDLP architecture

to explore the design space. Bluespec allows the designer to work at a higher abstraction level to implement the proposed design directly from a natural language interpretation of the specification. The system is designed to work in conjunction with a software driver that provides seed points and performs collision search on distinguished points. The ECDLP engine includes a complete point addition datapath that can be parallelized to perform multiple point addition steps simultaneously, as well as a communication interface to a central server. Each point addition core supports vectorized inversion with a variable vector size. The complete code listing for the ECDLP system is available at https://sourceforge.net/p/ecdlpbluespec/.

4.3.1 Hardware Implementation

A diagram of the proposed hardware design is shown in Figure 4.1. The design consists of a communication interface, point addition datapath, and microcoded controller. The communication interface includes the FPGA edge module, Input/Output Controller, and TX and RX FIFOs. Together these components allow transfer of seed points from the software server to the point addition datapath and distinguished points from the point addition datapath to the software server.

The point addition datapath uses high performance modular arithmetic units for multiplication, addition, subtraction, and inversion mapped onto high-speed DSP blocks in the FPGA. The microcoded controller sends control signals to datapath components to implement the sequence of operations required for point addition. The controller supports multiple parallel

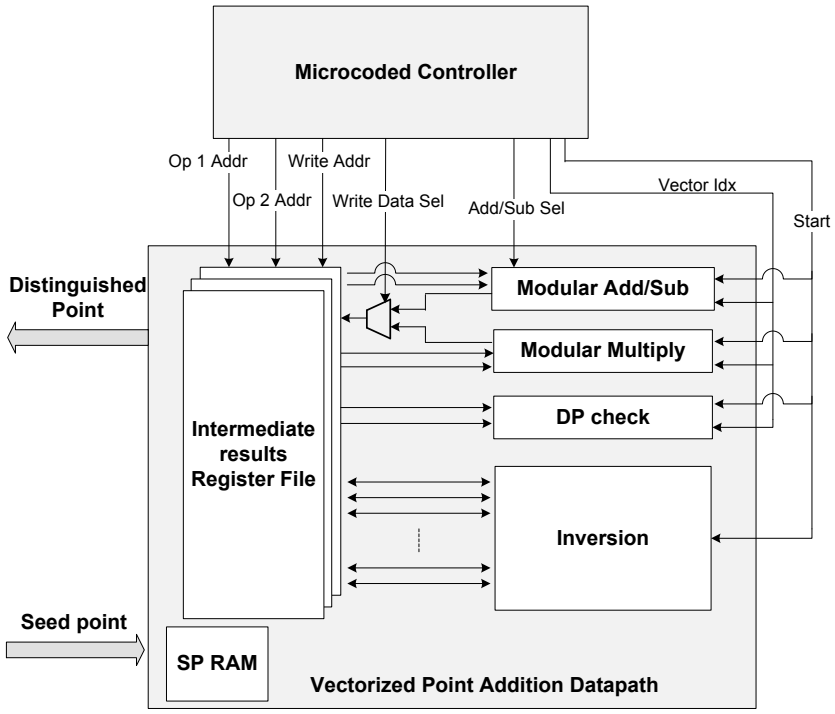


Figure 4.2: Vectorized point addition datapath with microcoded controller

point addition datapaths executing in SIMD fashion.

4.3.2 Vectorization

The datapath consists of modular arithmetic operators and memory. Each of these modules is designed to support vectorized point additions. A block diagram of the vectorized datapath is shown in Figure 4.2. As shown, each of the arithmetic modules interface directly to memory in order to load operands and store results. The modular add/sub, modular multiply, and distinguished point check modules are non-vectorized components and operate on inputs from one vector element at a time. In order to complete each arithmetic operation over the vectorized walks, computation is serialized and repeated M times. Conversely, the inversion module receives inputs from all M vector elements simultaneously and computes the inverse of all inputs together using the vector inversion algorithm described in Section 3.4.2.

Based on the sequence of arithmetic operations required for point addition, shown in Table 2.1, eight registers *(t1, t2, t3, t4, Px, Py, Qx, Qy)* are required to hold the intermediate results for each point addition. For vector size of M, M-entry register files are used to store these intermediate results. Each of the memories is 128 bits wide and has a depth equal to the vector size M. The memory address corresponds to the vector index for each stored data

value. These register files are mapped to the distributed RAMs available in the Xilinx Virtex 5 FPGA. An additional 256-bit M-entry memory is used to store the original seed points for each parallel walk. When a distinguished point is found, that point, along with the original seed point, is sent to the Input/Output controller for transmission to the software server.

Control signals for the vectorized datapath are generated by the microcoded controller. The controller iterates through all vector elements for a given operation before proceeding to the next operation. The vectorized datapath is implemented using register files provided in the Bluespec library to store starting points and temporary results for the point addition. The number of entries in each register file is equal to the vector size. Adjustment of this parameter is supported by the RegFile Bluespec library module. Since all control logic is specified using rules, vector size can be modified without any changes to the control.

4.3.3 Microcoded Controller in Bluespec

A microcoded controller is implemented to operate alongside the vectorized point addition datapath and provide the control signals necessary to compute vectorized point addition. This allows control of the execution flow of the low-level arithmetic operations for a point addition without modification of the datapath. The interconnections between the point addition datapath and microcoded controller are shown in Figure 4.2.

The proposed implementation is functionally equivalent to a conventional HDL microcoded controller, except that it uses Bluespec rules to implement the point addition controller at a higher abstraction level. Each microinstruction is encoded onto a single Bluespec rule. The rule condition maintains the order of operations and rule actions provide control signals to the point addition datapath. An internal program counter is used to control iteration through vector elements and the order of the field operations of a point addition.

Interface

The controller interface allows loading seed points and returning distinguished points. The interface, shown in Listing 4.5, provides the *loadRq* and *dpSendRq* methods to indicate when a seed point needs to be loaded or a distinguished point has been found. The return values of these methods are monitored by the Input/Output controller, which sends seed points and reads distinguished points using the *loadSeed* and *retDpSp* methods, respectively. In order to avoid stalling or missing points while waiting to send a distinguished point, the controller stores distinguished points in a FIFO. When the Input/Output controller reads a distinguished point, the first point is removed from the FIFO and returned to the software server. The FIFO is implemented from the Bluespec library and its size can be easily adjusted to accommodate the number of parallel point additions performed by the system.

Listing 4.5: Interface for microcoded controller

```
interface Pactrl;
  method Action loadSeed(Bit#(256) seedpt);
  method Action waitForSeed();
  method Action reset();
  method Action deqDpSp();
  method Bit#(512) retDpSp();
  method Bool loadRq();
  method Bool dpSendRq();
endinterface
```

Control

One Bluespec rule is required to implement each field operation of the point addition. As an example, the rule for the first field operation of a point addition, a subtraction of $y_S - y_T$, is given in Listing 4.6. For all operations, the rule condition ensures that the rule only executes when the program counter indicates the appropriate operation and the previous datapath operation is complete, as indicated by the *fire_cond* signal. The actions within the rule assert control signals for the point addition datapath to perform the operation. The arguments of *loadAddSub* in Listing 4.6 are control signals add/subtract operation select, operand 1 address, operand 2 address, register file write data select, register file write address, and vector element index. Other field operations also provide these control signals.

Listing 4.6: Microcoded control for field operations of a point addition

```
//Counter to iterate through vector element ids
rule vec_cnt_ctrl(incrCond);
  vecCnt <= vecCnt + 1;
endrule

//  1: t1 = Sy - Ty
rule op1_ctrl(oper == 4'd1 && fire_cond);
  pa0.loadAddSub(0, 5, 7, 1, vecCnt);
endrule
```

Internal state management for the microcoded controller uses additional rules to handle iteration through vector elements and incrementation of the operation counter. Vector iteration is controlled using a simple counter that increments whenever the previous datapath

operation is complete (indicated by the *incrCond* signal), as shown in rule *vec_cnt_ctrl* from Listing 4.6. After control signals for the current operation have been asserted for all vector elements, the operation counter increments and proceeds to the next operation. Operation and vector iteration control rely heavily on the use of the Maybe datatype to signal completion of datapath operations.

4.4 Parallel Pollard Rho Walks

The proposed design has been extended to support parallel Pollard rho walks by implementing multiple point addition cores on the FPGA. Each point addition datapath receives a separate set of seed points and performs a vectorized walk. The cores execute in SIMD configuration and are operated by a single microcoded controller. The controller monitors the state of all cores to ensure proper handling of seed points and distinguished points. Using Bluespec, the number of parallel point addition cores can be easily modified by instantiating additional datapaths and adding actions to the microoperation rules to provide required control signals to each core.

Chapter 5

Cryptanalytic Performance of the ECDLP Engine

The suitability of Bluespec for design space exploration is demonstrated in the context of a complex ECDLP machine for secp112r1. The proposed design has been implemented for various vector sizes between 1 and 64 with the number of parallel point addition cores varied between one and four. The high abstraction level of Bluespec allows these changes to be made easily. Performance results for each variation are compared and analyzed to determine the optimal parameter values for the ECDLP system. An advantage of the proposed solution is that it implements a complete hardware-software system and reports measured results from the actual ECDLP execution that includes all components and overhead associated with the system. The correctness of the proposed design is demonstrated by using seed points of low order (2^{50}, suggested in [5]), to achieve a collision with an expected walk length of 2^{25} point addition steps.

5.1 Implementation Platform

The complete ECDLP system is implemented on a Nallatech co-integrated hardware software platform. Figure 5.1 depicts the architecture of the Nallatech system. It consists of one quad-core Xeon E7310 processor and three Virtex-5 FPGAs (1 xc5vlx110, 2 xc5vsx240t). A fast North Bridge integrates high-speed components, including the Xeon, FPGAs, and main memory. A slower South Bridge integrates peripherals into the system, including the hard disk. Both the Xeon and the FPGA can directly access system memory using a Front Side Bus (FSB). The FPGA performs the computationally expensive Pollard rho iterations, while the Xeon processor manages the central database of distinguished points and executes collision search. Communication between software and hardware is carried out only for the exchange of seed points and distinguished points, which minimizes the communication

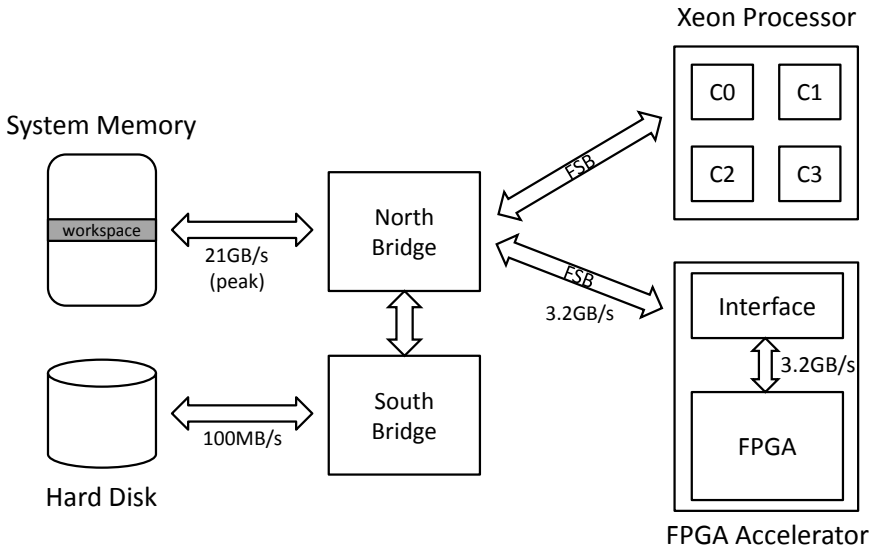


Figure 5.1: Nallatech system

overhead.

The Xeon processor executes a software driver (in *C*) and manages the software interface to the FSB. The software driver handles the communication interface with the FPGA, seed point generation, and storage and search of distinguished points. Software access to data transferred over the FSB is enabled through the workspace, a section of main memory allocated at the start of program execution.

When program execution starts, the software calls APIs to configure FPGA, initialize the FSB link, and allocate the workspace memory. It then starts an attack by generating random seed points on the curve and sending them to the FPGA over the FSB. Every point has x- and y- coordinates of 128-bit length each.

For each seed point received, the hardware computes a Pollard rho walk until a distinguished point is found, then returns the distinguished point along with the original seed point to the software. When the software receives a seed point-distinguished point pair from the FPGA, it performs a collision search over all received distinguished points. Once a collision is detected, it computes the secret scalar and reports the solution to the ECDLP. Since the software takes care of the central database of distinguished points, the collision search is conducted in parallel with hardware computation of the Pollard rho walks.

Table 5.1: Implementation results for ECDLP with variable vector size and cores.

	One Core		Four Core	
Vector Size	Speed $[PA/s]$	Area $[slices]$	Speed $[PA/s]$	Area $[slices]$
1	53K	4407	214K	12,391
8	300K	5977	1.20M	17,705
16	450K	6928	1.80M	22,200
32	598K	6619	2.38M	29,478
64	717K	9692	2.87M	35,232

5.2 Design Variations

This section evaluates the impact of vector size and number of point addition cores on performance and area of the proposed design by implementing both a single-core and four core version for vector sizes from 1 to 64. Measured results are given in Table 5.1. As shown, best performance is achieved by the designs with the largest vector size. Due to the additional storage required for vectorization, the area required to implement each design also increases with the vector size. Therefore, determination of optimal design parameters must account for the tradeoff between performance and area.

Comparison of results from the one and four core implementations shows that increasing the number of cores produces a linear increase in both performance and area. This is expected because point addition cores are implemented as parallel instantiations of the point addition datapath. The number of clock cycles required for a point addition step remains constant, but the addition of cores allows that cost to be spread over a larger number of points, resulting in a higher number of total point additions per second. In addition, these results show that for absolute performance, more cores is preferable over a larger vector size. Consider the four core implementation with vector size 8 and the one core implementation with vector size 32. Both generate 32 points per iteration, but the four core implementation achieves twice as many point additions per second at 2.5 times the area.

5.3 Speed/Area Tradeoff

Determination of the optimal design parameters for the proposed ECDLP system requires analysis of the performance and area of each alternative architecture. The goal is to find the configuration that best utilizes the limited area of the FPGA to achieve top performance. Thus, this analysis considers not only the speed of each variation, but also the area require-

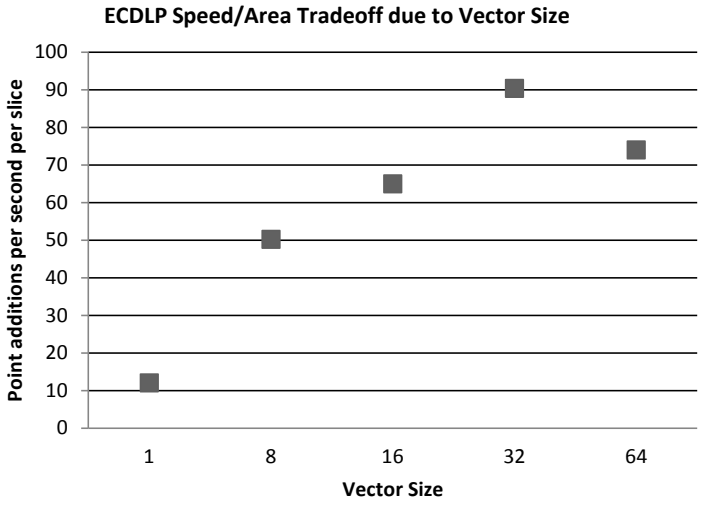


Figure 5.2: Evaluation of the performance per unit area

ments. Evaluation of the speed/area tradeoff using the metric point additions per second per slice is shown in Figure 5.2. The proposed ECDLP design achieves the best results, 90 point additions per second per slice, for vector size of 32. For larger vector sizes, the extra storage required to increase the vector size outweighs the performance benefit of larger vectors, resulting in a degradation of overall performance per unit area. In this system architecture, vectorization is entirely independent of the number of cores. Thus, the optimal design can be realized by implementing the maximum number of cores, each with vector size 32.

5.4 Comparison with Prior Work

Due to inherent differences between elliptic curves defined over binary and prime fields, performance comparison among various ECDLP implementations is not straightforward. Performance figures are also influenced by a variety of other factors, including target platform, curve size, and coordinate system. To minimize the impact of the target platform and coordinate system, performance of different ECDLP implementations is compared based on the total number of point addition operations performed per second.

Table 5.2 compares the proposed ECDLP solution with prior software implementations. It shows that a multicore implementation of the proposed design with 16 point addition cores is competitive with existing software solutions.

Comparison with prior hardware ECDLP implementations is shown in Table 5.3. The

Table 5.2: Comparison with software ECDLP implementations

Platform	**Latency per Iteration** $[ns]$	**Performance** $[PA/s]$
Cell processor @3.192GHz, secp112r1 curve [5]	113 (362 cycles)	8.81M
Cell processor, @3.192GHz, secp112r1 curve [11]	142 (453 cycles)	7.04M
Cell processor, @3.192GHz, ECC2K-130 Binary Field [10]	233 (745 cycles)	4.28M
This work, @100MHz secp112r1	1,670 (167 cycles)	single core: 598K 16 cores: 9.57M

proposed design achieves high performance relative to other prime field solutions. When adjusting for curve size, the proposed design demonstrates significant speedup over prior designs [23] [24].

Comparison with existing binary field solutions is also shown. However, it is difficult to make a fair comparison due to the significant differences between binary and prime field arithmetic. In particular, the properties of binary field arithmetic and reduction are well-suited for hardware designs, which allows them to achieve much higher performance.

The solution proposed by Fan et al. in [21] reports 111M point addition iterations per second. The authors of [21] claim their system is capable of solving ECC2K-130 within a year using five COPACOBANA machines. Similarly, the solution reported in [41] achieves 20M iterations per second. It is assumed that the performance difference between these systems and the proposed design is due to factors including binary field arithmetic, different curve sizes, and use of pipelined architectures.

5.4.1 Secp112r1 solutions

In terms of absolute performance, the proposed design is slower than previous published solutions targeting this curve. For example, [11] solves ECDLP for secp112r1 using Cell processors to perform point additions in 453 clock cycles, which yields 7.04M point additions per second. [5] improves this solution, computing point additions in 362 cycles and achieving 8.81M point additions per second. The performance gap between these prior works and the proposed design is due to the limited clock frequency of the hardware design. The hardware ECDLP system runs at 100 MHz, while the Cell processors used by [5] and [11] run at 3.125 GHz. In terms of cycles per point addition, the proposed design is competitive, with the single core implementation with vector size 32 requiring 167 clock cycles per point addition.

Table 5.3: Comparison with hardware ECDLP implementations (per core)

Platform	**Target Curve**	**Performance**	**Area**		
		[*PA/s*]	[*Slices*	*BRAMs*	*DSPs*]
Spartan-3 [21]	Binary (130-bit)	111M	26,731	20	0
Spartan-3 [41]	Binary (113-bit)	20M	13,900	18	0
Spartan-3 [23]	Prime (160-bit)	46.80K	3,230	15	0
Spartan-3 [23]	Prime (128-bit)	57.80K	2,520	16	0
Spartan-3 [24]	Prime (160-bit)	50.12K	2,660	*not given*	0
Virtex-5, this work	Prime (112-bit)	598K	6,619	9	48

Furthermore, using 16 cores with a vector size of 32, the design can compute an estimated 9.57M point additions per second.

5.5 Optimization of the Modular Multiplier

To achieve optimal performance, the impact of the integer basis for the polynomial representation on the latency of the modular multiplier design is evaluated. The performance of the modular multiplier is heavily influenced by the value selected for l_a, which determines the length of each field in the operands and the number of partial products to be computed. The proposed multiplier architecture relies on fast computation of partial products using the DSP blocks available on the FPGA, so performance is also impacted by the ability of the selected field size to efficiently use these resources.

Modification of the multiplier architecture to support different field lengths for polynomial representation is straightforward requiring only changes to the number of multiplier-adder columns used. The number of multiplier-adder columns is given by n_a, which is related to the field length l_a. Increasing the field length requires fewer columns in the architecture and reduces the number of partial products computed. The results of this analysis are given in Table 5.4. As shown, increasing the length of the field of the polynomial representation reduces the number of cycles required for the modular multiplication. However, using larger

Table 5.4: Modular multiplier performance

Field length l_a	**Time** [*Cycles*]	**Max Frequency** [*MHz*]	**Area** [*Slices,DSPs*]	**Latency per Iteration** [μs]	**System Performance** [*PA/s*]
8	21	181	263, 18	1.40 (253 cycles)	714K
16	14	181	217, 10	0.92 (167 cycles)	1.09M
32	7	104	253, 20	1.25 (130 cycles)	800K
64	3	68	370, 36	1.41 (96 cycles)	708K

fields also degrades the maximum clock speed of the design. The 64-bit field size requires the fewest cycles per modular multiplication, but achieves worse overall performance due to the low maximum clock frequency.

The impact of the modular multiplier design on overall point addition performance is also evaluated. These results are also shown in Table 5.4. Modular multiplication is the dominant operation in each point addition, with multiplication requiring approximately 85% of the cycle count for each iteration. Thus, decreasing the cycle count of the modular multiplication operation has significant impact on the overall performance of the proposed design. The results show that best overall performance is achieved for field length of 16-bits when the full system runs at 180MHz. However, since maximum clock frequency of the complete design is 100MHz, field length of 32 bits is the optimal value. This gives 1.29X increase in system performance (PA/s) relative to 16-bit field size.

Güneysu and Paar's architecture described in [22] targets 256-bit prime arithmetic over two fixed NIST primes. The proposed solution shows an improvement in terms of latency for the important ECC arithmetic operations. Assuming the cycle cost for 256-bit arithmetic is twice of that for 128-bit arithmetic (worst case scenario), the proposed architecture has cycle cost of 14 for a modular multiplication and 260 for the point addition. This is a 5X latency improvement for modular multiplication and a 3X improvement for point addition relative to Güneysu and Paar's design [22].

5.6 Extension to Other Curves

The ECDLP system, as well as modular arithmetic modules, is designed to use a generalized architecture that can be adapted to multiple standard elliptic curves over GF(p), where p has the form $(2^n - m)/k$. This is demonstrated by adapting the modular multiplier architecture to be compatible with the NIST standard P-192 curve [46], which uses the prime $p = 2^{192} - 2^{64} - 1$. The multiplier architecture for this curve is shown in Figure 5.3. The

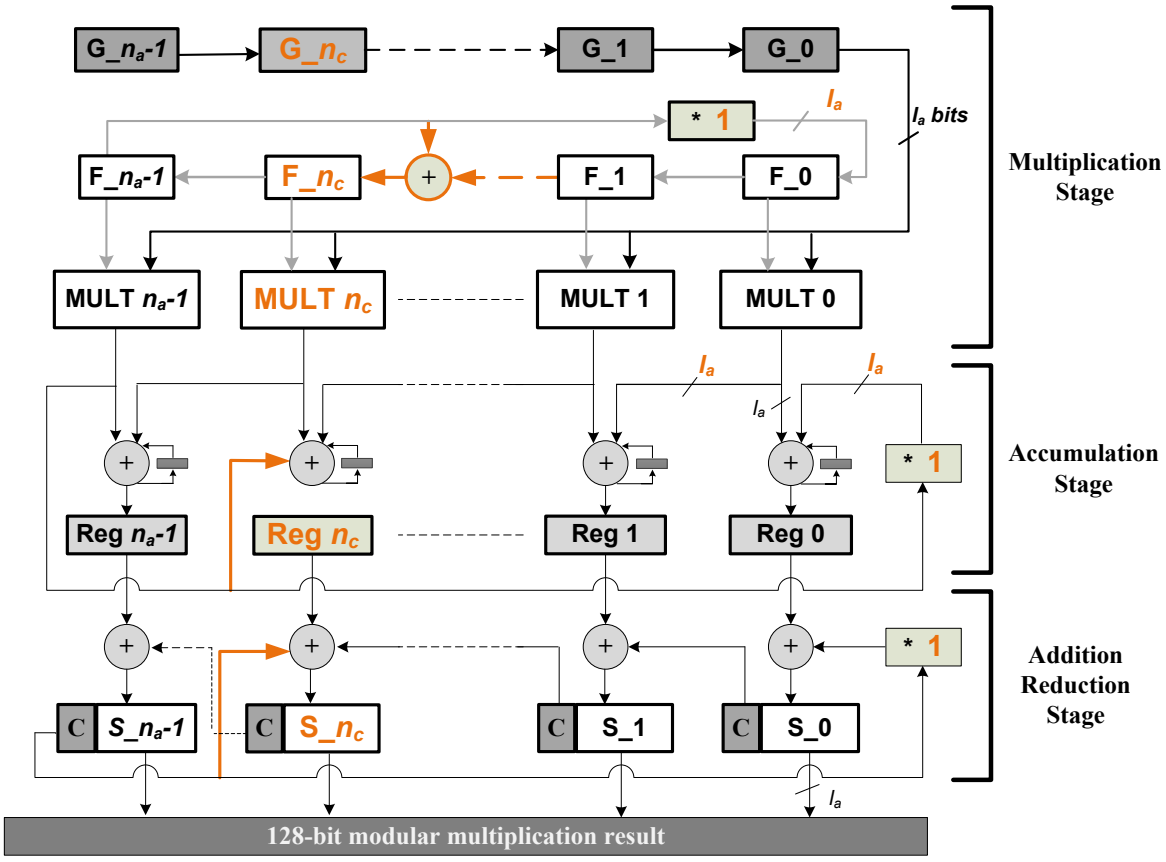


Figure 5.3: Modular multiplication architecture: Extension to NIST P-192 curve

multiplication for this 192-bit NIST curve is similar to that of secp112r1 curve, but the 192-bit curve needs additional multiplier-adder columns. In addition, P-192 uses $m = 2^{64} - 1$, rather than $m = 3$, which requires the following reduction operation. With polynomial field size l_a, n_c is defined as the field containing 2^{64}, which is $n_c = \frac{64}{l_a}$. For $A = A_{n_a-1} \cdot 2^{192} + A_{n_c} \cdot 2^{64} + A_0$, the reduction modulo p is $A \bmod p = (A_{n_c} + An_a - 1) \cdot 2^{64} + (A_0 + A_{n_a-1})$.

The modular multiplier architecture is modified as follows to make it compatible with P-192 curve.

- Multiplication factor 3 is replaced by 1 in the rotation path of multiplication stage as well as in the addition reduction feedback path. This reduces the number of bits in each field of operand F to l_a-bits and the length of partial product outputs to $2l_a$-bits.
- In the rotation path, operand field F_{n_a-1} is fed back to two F_{n_c} and F_0. An adder is introduced before F_{n_c} for this purpose. Thus, $F_{n_c} = F_{n_c-1} + F_{n_a-1}$.

- In the accumulation stage, carry bits of highest accumulator output (*Reg* $n_a - 1$) are fed back to both *Reg* n_c and *Reg 0*. Similarly in the addition reduction stage, carry bits of S_{n_a-1} are folded and added back to both S_{n_c} and S_0.

The adapted architecture has a latency of 17 clock cycles at 193 MHz for $l_a = 16$ with area of 364 Virtex 5 FPGA slices and 12 DSP cores. This is comparable to the multiplier performance for curve secp112r1 (reported in Section 5.5) and demonstrates the flexibility of the proposed design to accelerate modular multiplication for general prime field elliptic curves.

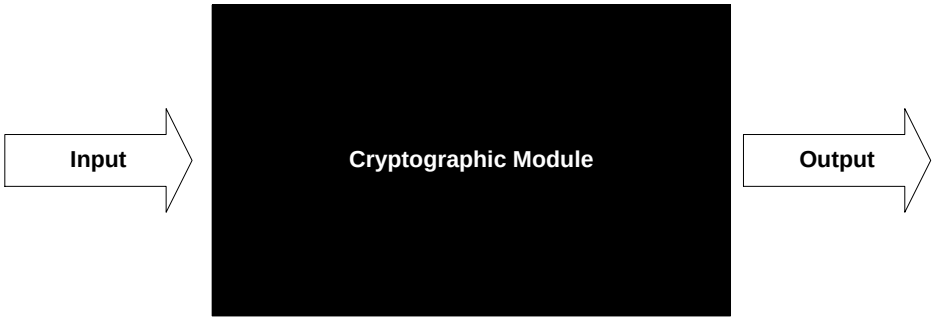
Chapter 6

Implementation Attacks

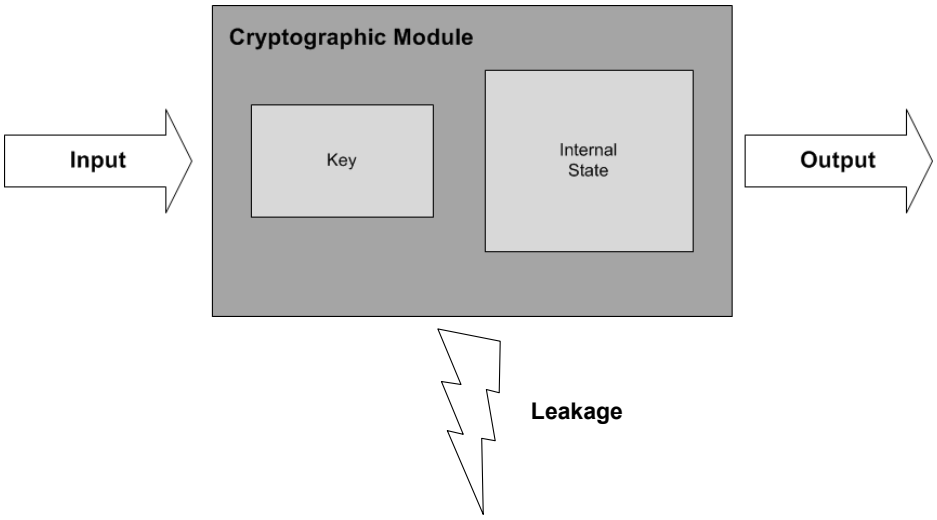
Implementation attacks are one of the most serious security threats facing cryptographic systems. These attacks exploit weaknesses in the hardware or software implementation of a system to reveal sensitive information, such as the encryption key or internal state variables, used by a cryptographic module. Implementation attacks pose a unique challenge to the security community, which has traditionally focused on theoretical security of algorithms and protocols. Classical cryptanalysis assumes that a system is secure if the cryptographic algorithms and protocols implemented are secure. Implementation attacks violate this assumption by attacking the system implementation, which includes the hardware architecture, software design, and device platform, rather than the cryptographic algorithm itself.

The difference between implementation attacks and logical cryptanalysis is illustrated by Figure 6.1. When performing vulnerability analysis, logical cryptanalysis regards the system as a black box and considers attacks using only information available from outside the system, such as inputs and outputs. As long as these external values cannot be used to derive secrets, the system is secure. However, due to the physical properties of device hardware, the system can provide additional information to the attacker in the form of leakage. Leakage is defined as information about device operation due to the hardware implementation that can be obtained through observation or manipulation of the system. The presence of leakage breaks the classical black box model and causes the system to operate as a gray box with leakage providing limited visibility into system behavior. Cryptographic systems designed based on the black box model assume that internal state variables used during computation are not known to the attacker. The visibility provided by device leakage can reveal these variables and lead to complete system breakage.

Any cryptographic device that can be physically possessed or observed by an attacker is potentially vulnerable to implementation attacks, regardless of the security of the underlying algorithm or protocol employed by the device. Embedded systems are particularly vulnerable to implementation attacks because they often operate in such conditions, but servers and desktop computers can also be at risk. Despite the use of cryptography to pro-



(a) Logical cryptanalysis: Black box model



(b) Implementation attack: Gray box model

Figure 6.1: System view for logical and implementation attacks

tect sensitive data, devices are vulnerable to implementation attacks and successful attacks have been demonstrated for standard security algorithms including AES and RSA. Furthermore, Clavier shows that an unknown cryptographic algorithm is not a significant barrier to implementation attacks [17]. This creates a significant security risk for devices employing cryptographic algorithms to protect sensitive data. Implementation attacks are an active area of research and the sophistication of implementation attack techniques continues to increase rapidly, meaning that systems currently thought to be secure may soon become vulnerable.

Attacks are characterized by on the level of access an attacker has to the system (invasiveness), as well as the influence he exerts over system operation. Invasiveness of implementation attacks varies from noninvasive, which requires access to only the inputs and outputs of the

target device, to invasive, which requires depackaging the chip and making direct electrical connections with internal components. Attack cost and complexity is heavily influenced by the invasiveness of the attack method. Implementation attacks are also identified based on the behavior of the attacker within the system. Passive attacks merely observe system operation, whereas as active attacks directly manipulate the target device to change its behavior.

6.1 Types of Attacks

Implementation attacks are classified based on the way information is gathered from the target device. The most common types of attacks are side-channel, fault, and probing. Side-channel attacks are passive attacks in which the attacker observes the device under normal operation and analyzes characteristics such as execution time, power consumption, and electromagnetic radiation to determine a secret value. Fault attacks cause specific errors in system behavior that reveal secret information. Probing attacks use probes placed within a chip to spy on its internal values and derive, or in some cases directly read, a secret value. Due to their low cost and noninvasiveness, side-channel attacks are the most common implementation attacks, although noninvasive fault attacks are also popular. In the following sections, each major type of implementation attack is introduced.

6.1.1 Side-channel

Side-channel attacks exploit the fact that hardware used in many systems unintentionally leaks information by exhibiting a dependency between internal data and externally observable characteristics. By definition, side-channel attacks are passive and noninvasive. Commonly studied sources of leakage include power consumption [36], electromagnetic radiation [51], and execution time [35]. Most leakage is due to the properties of the microprocessor architectures and CMOS transistors used in modern hardware. For example, execution time of a microprocessor instruction often varies based on the value of the operands and where they are stored within the memory hierarchy. In addition, CMOS transistors consume power during output switching, resulting in correspondence between power consumption and internal data.

Some systems are vulnerable to simple side-channel attacks, which allow the attacker to directly read a secret value from a device by observing its leakage during a single encryption operation. Other systems require require more complex analysis that observes one or more leakage sources over many operations and derives secret values using statistical analysis. Differential Power Analysis (DPA), introduced by Kocher et al. in 1999 [36], is the most commonly used side-channel attack that relies on statistical analysis of leakage. DPA requires a data collection phase, during which leakage is observed over many operations, and a data

analysis phase that processes the data to reveal the secret key. When performing data analysis, the attacker targets a low-level operation that combines a known value, such as plaintext input, with a secret value, such as a bit or byte of the encryption key. The attacker then chooses a guess for the target secret value and classifies leakage traces based on the expected output of the target operation. If the guess is correct, there will be a statistical difference of means between the two classifications of traces. By repeating the process for all possible values of the targeted secret, the attacker can determine its actual value. DPA is a powerful statistical technique that assumes only that the target operation produces leakage that differs in a statistically significant way based on the operation output.

Correlation Power Analysis (CPA) [14] is an advancement of DPA that relates variations in power consumption during a secure operation to a leakage model defined based on the internal state of algorithm under attack. Common leakage models include Hamming weight of a state variable and Hamming distance between a state variable and its prior value. CPA can attack any operation that combines a fixed secret value with a known variable. The output of the target operation is assumed to be related to the actual device power consumption based on the selected leakage model. To perform CPA, the statistical correlation between the actual device power consumption and estimated power consumption given by the leakage model is compared for each possible value of the fixed secret. The actual secret value can then be determined as the value that shows high correlation between the actual and estimated power consumption. To reduce the impact of noise on the results, actual power consumption is replaced with an average of actual power consumption over many traces before calculating the correlation with estimated power consumption.

6.1.2 Fault

Fault attacks are active attacks that manipulate a device to cause errors in its output that can be used to derive a secret value. These attacks depend on the attacker's ability to inject faults that produce predictable changes in device behavior. Many different methods have been proposed for fault injection and these can be broadly classified as noninvasive, semi-invasive, or invasive based on the level of device access required.

Noninvasive Fault Attack

Noninvasive fault attacks involve injecting faults on global inputs and observing the effect on device output. Common noninvasive attacks induce glitches on device power supply or clock [3]. Power supply glitches, when timed properly, can corrupt data read from memory or cause a microprocessor to skip instructions. By inserting a glitch on the clock, an attacker can reduce the time between clock pulses and cause operations to be skipped or registers within the circuit to store unintended intermediate values. Changing the operating temperature of the device is also considered a form of noninvasive fault attack and can inhibit or

corrupt memory access operations [3]. Once faulty outputs have been observed, Differential Fault Analysis (DFA) is used to determine the target secret value. DFA generally involves statistical comparison of the faulty and correct outputs, but the specifics of the analysis depend on the cryptographic algorithm under attack.

Since they only require access to device inputs and outputs, faults injected using noninvasive methods affect the entire chip globally and require precise timing to ensure a predictable and consistent impact on the device. Nonetheless, noninvasive fault attacks have been successfully applied to break implementations of secure algorithms including RSA [1] [2] and DES [7].

Semi-invasive Fault Attack

Semi-invasive fault attacks inject faults into internal chip components without direct electrical contact or damage to the chip. These attacks require that the chip be depackaged, but allow the passivation layer to remain intact. Optical fault injection, proposed by Skorobogatov and Anderson in 2002 [54], is the most common approach to performing semi-invasive attacks. Optical faults can be injected by applying ionizing radiation to specific targeted transistors using UV lights, lasers, or photo flashes [54]. The radiation causes the targeted transistors to change state, allowing an attacker to selectively flip memory bits. This allows an attacker to control program execution, by targeting instruction memory, or internal state data, by targeting RAMs. The primary advantage of semi-invasive attacks over other fault injection techniques is that they allow targeted attacks on specific components within the device without incurring the high equipment cost of invasive techniques.

Invasive Fault Attack

Invasive fault attacks use the most powerful fault injection techniques available to an attacker. Invasive fault injection requires depackaging the chip, removing the passivation layer, and making direct electrical contact with on-chip components. Depending on the specific fault injection technique, chip behavior may be permanently altered during this type of attack. Focused ion beams and laser cutters are frequently used to add and remove wires in on-chip circuits. Specialized microprobes can also be used to inject faults directly onto wires in the chip.

Invasive fault attacks require expensive equipment, as well as a significant time investment. In addition, invasive fault injection is constantly becoming more difficult due to decreasing feature size and increasing on-chip transistor density. For these reasons, invasive attacks are the least common type of fault attack.

6.1.3 Probing

Probing attacks are similar to invasive fault attacks, in that they require complete access to chip internals. However, unlike fault attacks, probing is a passive form of implementation attack that uses probes placed directly on internal chip wires to read secret values. A probing attack requires several steps, including chip depackaging to expose the chip and allow direct electrical connections, layout reconstruction to derive the circuit schematic and identify major functional blocks, and probe placement to read secret values. Unlike side-channel attacks, probing attacks rarely require complex analysis to determine the target secret value, since it can usually be read directly through the probe. Therefore, the primary challenge for an attacker is determining the placement of probes within the chip. Due to equipment and time required, there is a high cost associated with each probe placement. To manage cost, the attacker must devote significant resources to studying the chip to ensure optimal placement. Common attack strategies aim to place probes on memory address or data buses. Despite their high cost, probing attacks have been shown to be effective against both public key [26] and private key [26] [53] cryptosystems.

6.2 Countermeasures

Due to the growing threat of implementation attacks, secure devices must implement countermeasures to protect sensitive data. The specific countermeasures required to secure a certain device depend on a variety of factors including the level of access available to an attacker and the properties of the implementation platform. Effective countermeasures require precise modifications to system design and can be difficult to implement properly. In addition, there is a large knowledge gap between system designers, who are not typically knowledgeable about implementation attacks and countermeasures, and attackers, who possess significant expertise. This can lead designers to underestimate a system's vulnerability to implementation attacks and put sensitive data at risk.

Since implementation attack vulnerability results from design decisions made when mapping a cryptographic algorithm or protocol from its specification to the device platform, countermeasures typically require significant changes to the system architecture. This suggests that implementation attack risk mitigation requires a secure by design approach that includes these risks in system requirements and ensures that countermeasures are included in the initial design phase. All countermeasures must be properly tested to ensure mitigation of implementation attack vulnerability without introducing additional risks.

6.2.1 Vulnerability Testing

Implementation attack vulnerability testing is fundamentally different from other forms of security testing. A system's vulnerability to implementation attacks is heavily influenced by system design and architecture, as well as the characteristics of the device platform. The exact impact of these factors is difficult to predict outside of an attack scenario, which makes it impossible to evaluate implementation attack risk using common white-box testing techniques such as static source code analysis and data-flow analysis. Instead, implementation attack resistance testing requires a dynamic approach that tests the complete system implementation together with the platform.

There are commercial systems available that can be used to assess vulnerability to implementation attacks. The DPA Workstation [18] developed by Cryptography Research, Inc. is a specialty tool that performs side-channel analysis, specifically differential power or electromagnetic analysis, on embedded systems. The DPA Workstation includes an environment for collection of data from the device under attack and proprietary software that can perform side-channel analysis on all major standard ciphers. A similar system is Inspector [52] from Riscure, which supports both side-channel analysis and fault analysis. The Inspector platform includes custom measurement and fault injection hardware along with proprietary software to perform complete side-channel and fault attacks on standard ciphers. Brightsight also offers an implementation attack toolset that includes platforms for side-channel and fault analysis [15].

Side Channel Analysis Resistant Framework (SCARF) [33] is an academic tool developed by the Electronics and Telecommunications Research Institute and intended for use in research on countermeasures for side-channel and fault attacks. SCARF includes a number of custom evaluation boards that can be used to test attack resistance in devices such as smartcards, microprocessors, and FPGAs. However, SCARF only supports testing of the specific device models included in its custom evaluation boards. The Flexible Open-source Board for Side-channel analysis (FOBOS) [60] developed at George Mason University is an academic platform under development for implementation attack resistance testing. FOBOS aims to provide an open-source platform that can be used to evaluate effectiveness of side-channel analysis countermeasures on a variety of different FPGA platforms.

Implementation attack techniques and countermeasures are an active area of academic research. However, differences between device platforms, measurement equipment, and test setup make it difficult to compare results. Within the research community, there is great interest in performing implementation attacks in a way that allows fair comparison against one another. Comparison is useful for evaluating the relative strength of different attack methods, as well as determining the relative security level of different devices and implementation attack countermeasures. The DPA Contest [56] provides a framework for comparison of differential power analysis side-channel attacks. The most popular iteration of the DPA contest provides a fixed number of power traces collected from an FPGA implementation of AES and requires entrants to submit attack scripts that can derive the secret key of the de-

vice. Since all attacks target the same implementation and rely on the same measurements, fair comparison is straightforward. However, the applicability of the DPA contest is limited due to its reliance on a trusted third party to provide the side-channel measurements needed to mount an attack.

6.2.2 Standardization

At present, there is no standard technique or open source platform for evaluation of vulnerability to implementation attacks. This creates a significant disadvantage for both designers and users in terms of validating correct application of countermeasures and establishing a minimum acceptable level of implementation attack resistance. In response to this concern, NIST has decided to establish standards for implementation attack resistance in its forthcoming FIPS 140-3 computer security standard [47]. The FIPS 140-3 standard is currently undergoing a public comment period in which NIST has specifically requested comments on the appropriate role of implementation attack resistance in security level certification [44]. NIST has also sought input on development of standard techniques and tools for noninvasive implementation attack resistance validation through the Non-Invasive Attack Testing Workshop held in 2011. A standardized test environment and procedure is necessary to provide the quality and repeatability of implementation attack resistance evaluation required by the FIPS 140 standard.

Chapter 7

A Modular Test Environment for Implementation Attacks

In this chapter, a modular design approach is applied to improve the process of implementation attack vulnerability analysis. The proposed modular test environment is a flexible solution that can be used for any type of implementation attack on any cryptosystem, regardless of the platform. Use of this test environment improves the efficiency of the attack by allowing users to focus their efforts on critical tasks, design of new attacks or countermeasures, rather than construction of an appropriate test setup for data collection and analysis.

7.1 Test Environment Design

In this section, design of a modular testing environment for implementation attack vulnerability analysis and standardization is described. A key feature of the test environment design is the separation between the design under test (DUT) and the test script used to perform vulnerability analysis. This approach allows a clear separation between the tasks of system design and security testing, each of which requires separate expertise. The test environment is designed to ensure that test scripts can be written without specific knowledge of the implementation details of the DUT or the platform on which the DUT is implemented. Due to the lack of dependency between the DUT and the test script, the proposed test environment allows a single test script to be used to evaluate multiple DUTs in order to provides a fair security comparison of each DUT. The proposed design is open source and additional details on the design and functionality of the proposed test environment are available at `http://rijndael.ece.vt.edu/iameter`.

A block diagram of the proposed test environment is shown in Figure 7.1. The unshaded blocks are implemented as part of the test environment itself, while the shaded blocks are

user-designed modules. As shown, the user designs the test script and DUT. The attack equipment is also provided by the user and is used to observe or manipulate device behavior as required to perform the implementation attack. The test environment defines an interface for each user-designed module, which permits interaction within the test environment, while allowing each module to remain independent. The interface modules are designed to maximize portability and can be used to support virtually any DUT platform or attack equipment.

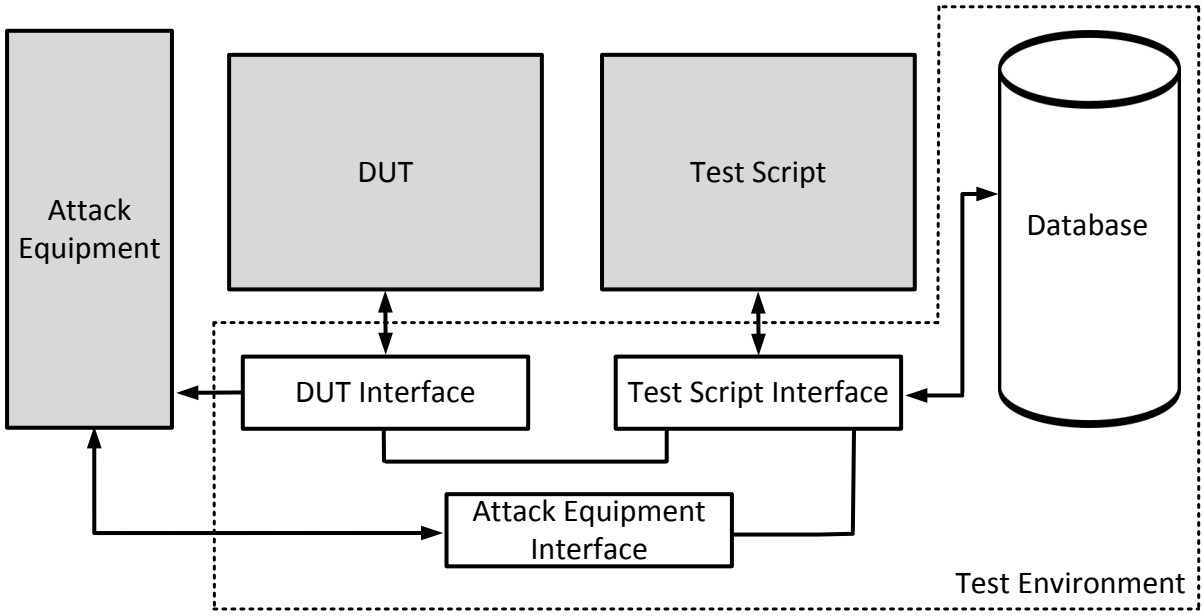


Figure 7.1: Modular test environment for implementation attacks: Block diagram

Advantages of the proposed test environment include portability and openness. Implementation attacks are complex and require specialized knowledge. The test environment allows test scripts to be designed independently of the DUT, which accommodates integration of implementation attack expertise into test scripts. This approach makes it possible for implementation attack experts with no knowledge of the DUT to create highly effective test scripts. The portability inherent in the test environment design enhances this feature by ensuring that test scripts, as well as the test environment itself, are independent of the platform on which the DUT is implemented. While hardware and software DUTs may have completely different implementation characteristics, both can be tested in the proposed environment using the same test script. In addition, the test environment is independent of the attack equipment, as well as the type of implementation attack tested.

An additional feature of the proposed test environment is its openness; the design is fully open-source. The open source design has a number of practical advantages including extensibility and low-cost, however, the primary benefits are more far-reaching. The proposed

test environment provides a uniform open platform that can be used in both industry and academic research and can facilitate cooperation and collaboration. For the past decade, the trend in security and cryptography has been to embrace openness, rather than obscurity, in the design and analysis of cryptographic algorithms. The open selection process used by NIST to decide the AES [48] and SHA-3 [45] standard algorithms exemplifies this trend. The proposed test environment can be used to increase openness by allowing establishment of open standards for implementation attack resistance and security validation. For example, a standard test script to determine the implementation attack security of an algorithm or protocol could be established and openly distributed.

7.1.1 Usage Model

The proposed modular test environment is intended to be used for evaluation of the implementation attack resistance of a design. Due to the broad scope of implementation attacks, implementation attack resistance should be integrated into the system design process as follows. The initial system requirements document should define both functional and security requirements for the design. Implementation attack security goals should be included through specification of relevant types of attacks and required resistance levels. The system designer then constructs the DUT, incorporating countermeasures intended to reduce the risk of the specified implementation attacks. Meanwhile, the test script designer, ideally an implementation attack expert or security standards body, independently creates a test script based on the requirements document to determine whether the DUT satisfies the defined implementation attack security requirements.

The test procedure used in the test script varies depending on the cryptographic algorithm or protocol implemented by the DUT, as well as the type of implementation attack performed. However, a general test sequence is presented as a guideline of a test procedure that is appropriate in most cases. The test script initialization routine should establish a connection with the DUT and attack equipment using their respective interfaces. If necessary, the initialization routine should also configure the attack equipment with the desired settings. The test script then enters a data collection loop. During each loop iteration, the test script sends a message to the DUT to start operation, observes or manipulates DUT operation using the attack equipment, and creates a trace based on the DUT input, output, and the observations or manipulations performed by attack equipment. The number of loop iterations depends on the required security level of the DUT, where a large number of iterations can be used to verify a high security level. Analysis of the traces to determine implementation attack vulnerability can either be performed in real-time during data collection or in post-processing after all traces have been collected. The primary advantage of real-time analysis is that test procedure can terminate early when the DUT reaches a failure condition before trace collection is complete.

Table 7.1: DUT interface API

Function	**Description**
init(addr_len, data_len)	Establish a connection between the DUT and the test script. Transfers size of each message field, as defined in the DUT, to the test script to ensure synchronization
close()	Close the connection between DUT and test script.
read(addr, data)	Reads a message from the DUT and returns it to the test script. The address returned is the address of the DUT port from which the data were received. Returns immediately with value of 0 if a complete message is not available from DUT.
write(addr, data)	Writes a message to the DUT. The parameters addr and data correspond to the address and data message fields respectively.

7.1.2 DUT Interface

The DUT interface is the module that interacts directly with the DUT and manages data transfer between the test script and the DUT. The primary goal of the DUT interface is to create an abstraction layer over the DUT itself that allows communication with the test script, regardless of the data transfer protocol used by the DUT. To achieve this, the DUT interface defines a message passing protocol that must be supported by both the DUT and the test script. To support the message passing protocol, the DUT must specify its number of input/output ports, as well as the size of each port. The message passing protocol defines a message as containing two fields: address and data. The address field gives the specific port associated with the message data. The size of the data field is equal to the size of each DUT port and address field size is equal to base-2 logarithm of the number of ports. The DUT interface API is shown in Table 7.1. The API allows the DUT interface to be platform-independent by ensuring that all aspects of data transfer between the DUT and test script below the abstraction level of the message passing protocol are confined to the implementation of the DUT interface itself. In addition, the functions defined in the API are general enough that they can apply to to any DUT and implementation attack scenario.

The DUT interface can also implement synchronization between the DUT and attack equipment. This is supported by an additional triggering output port in the DUT interface that connects directly to the attack equipment. Assertion of the triggering output can be used to notify the attack equipment of the status of sensitive operations performed by the DUT. In many cases, the trigger from the DUT interface asserts a start signal in the attack equipment to begin data collection or manipulation.

Table 7.2: Attack equipment interface API

Function	**Description**
init(ip_addr)	Establish connection between the test script and the attack equipment over the given IP address. Returns a 1 for success
close()	Close communication link between the test script and the attack equipment.
set_parameters(keyword_args)	Set the parameters of the attack equipment to the values determined by the keyword arguments.
write(cmd)	Write the command to the attack equipment and returns the number of bytes written.
read()	Read the response from the attack equipment, returns the response string.
capture(save_capture, file_name)	Retrieve the latest data captured from the attack equipment, appends it in the file determined by *file_name* if *save_capture* is not equal to 0.
save_settings(file_name)	Save settings of the attack equipment into a file determined by *file_name*.
recall_settings(file_name)	Configure the attack equipment with the parameters stored in a file.

7.1.3 Attack Equipment

Implementation attacks require specialized devices to observe or manipulate behavior of the DUT. These devices are referred to as the attack equipment and the test environment must include at least one such device. The specific attack equipment required varies based on the type of implementation attack under evaluation. The test environment defines an attack equipment interface in order to allow interaction between the test script and attack equipment. The attack equipment interface API is shown in Table 7.2. As shown, the interface allows the test script to access data measured from the DUT, send a configuration string to fix the equipment settings, and read or write equipment-specific commands. Connection between the test script and the attack equipment is established over IP, which has the benefit of compatibility with a wide variety of instrumentation and support for remote interaction.

Table 7.3: Database traces table

Column	**Description**
TRACEID	32-bit number uniquely identifying the trace.
ASSOCIATED	1024 character STRING. The string represents associated data for the trace, such as the input/ output values of the cipher DUT (plaintext, ciphertext, key).
CAMPAIGNID	Reference (32-bit) number to the CAMPAIGN record that describes the overall setup used to collect the trace.
DATA	20kb raw binary array. This represents the trace data, in binary format.

7.1.4 Database

A unique feature of the proposed test environment is the inclusion of a database to store data collected from the attack equipment. A trace is a set of data collected by the attack equipment during a single operation performed by the DUT. Storage of traces in the database allows easy management and retrieval of data collected over multiple iterations of testing. In addition, use of the database allows physical separation between trace collection and analysis. Once a set of traces has been collected and stored in the database, test scripts can access them via the database API and perform attacks or analysis remotely.

Each time an implementation attack is performed, hundreds or thousands of measurement traces of DUT behavior must be collected using the attack equipment. Each trace represents an execution of some operation by the DUT and is characterized by the input and output data of the DUT, as well as the observations or manipulations of the attack equipment. Implementation attacks are defined not only by the traces collected from the DUT, but also by the configuration settings of the test environment. The database is organized into two tables: the traces table, which stores measurement traces, and the experiments table, which stores data on test environment settings during trace collection. The columns of each database table are shown in Table 7.3 and Table 7.4, respectively. The information contained in these tables can be used for either of two distinct purposes: to analyze a set of traces for implementation attack vulnerability or to reproduce a set of traces by duplicating the DUT, platform, and test environment configuration used for data collection.

All columns of each database table can be easily accessed within a test script using the provided database API. For ease of the use, the database API provides an additional function that allows storage of multiple traces from one experiment with a single function call.

Table 7.4: Database experiment table

Column	**Description**
CAMPAIGNID	32-bit number uniquely identifying the experiment.
EQUIPSETTINGS	64 kb STRING. This field contains the complete attack equipment settings associated with the setup in this experiment.
DUTREPOURL	1024 character STRING. This represents a URL to a repository that contains the full DUT definition (source code, bit streams, binaries).
DUTREPOREV	32-bit integer. This represents the repository version of the DUT for this campaign.
CAMPAIGNCOMMENT	User-defined 1024 character string. This field contains a brief description of the experiment, the attack equipment used, the DUT platform used, the DUT, etc. Note that the DUT repository can provide additional detailed documentation, if needed.
SAMPLEBYTES	8-bit number that tells how many bytes per sample the capture contains for the campaign.

7.1.5 Test Script Interface

Test scripts execute in a Python scripting environment installed on a control PC and interact with the DUT, attack equipment, and database. The test script interface is defined as the control PC software required to allow communication from the Python scripting environment to the DUT, attack equipment, and database via their respective interfaces. The control PC software is accessed directly from the scripting environment via the test script and operates according to the specified APIs for the DUT interface, attack equipment interface, and database.

Chapter 8

Practical Demonstration of the Modular Test Environment

The functionality of the proposed modular test environment is demonstrated by using it to perform a power analysis side-channel attack. The test setup is typical for SCA and includes an oscilloscope and embedded microprocessor. The results demonstrate that the modular test environment can be easily applied to a practical implementation attack. Further applications of the test environment to various implementation attacks proposed in the literature are also discussed.

8.1 CPA on Embedded Software Implementation of AES

In this section, an application of the proposed test environment to assess the vulnerability of an embedded system to SCA using power analysis is presented. This example uses a software implementation of AES, without countermeasures against implementation attacks, on a MicroBlaze soft-core microprocessor configured on a Spartan-3 FPGA. A Tektronix DPO3034 oscilloscope is used to collect power traces from the device and each power trace represents the power consumption during one AES encryption. The CPA technique is used to analyze the power traces and determine the secret key used for AES encryption.

8.1.1 DUT

The DUT in this experiment is a standard software implementation of AES encryption. The DUT receives a key and plaintext from the test environment and responds with the encrypted ciphertext. The DUT is defined with a total of three input/output ports, one each for the

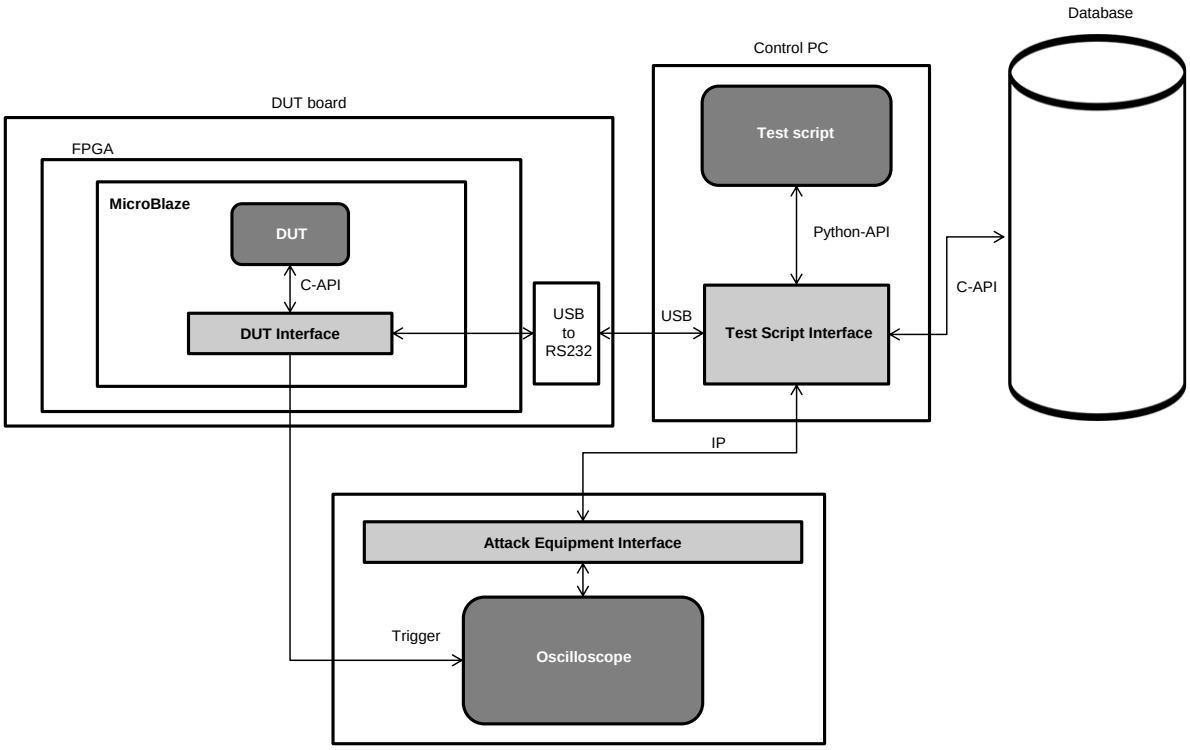


Figure 8.1: Test environment setup for CPA experiment: Block diagram

key, plaintext, and ciphertext, and sets the size of each port as sixteen bytes. This allows the complete key, plaintext, or ciphertext to be transferred using a single message.

The DUT is implemented on a Xilinx Spartan 3 XC3S500E FPGA platform configured with a MicroBlaze microprocessor [61]. The MicroBlaze is programmed with a software application that runs the AES encryption in an infinite loop. The DUT is connected to the test script interface via the control PC using an RS-232 serial connection.

The DUT interface handles all RS-232 data transfer operations using a send/receive buffer to implement the message passing protocol for the DUT and test script. This ensures that all communications are seen as complete messages by both the DUT and test script. The DUT interface also implements assertion of the trigger signal to the attack equipment at the start of each AES encryption. The trigger signal is used by the oscilloscope to begin collecting a power trace from the DUT. To reduce the impact of noise on power consumption measurements, each encryption with a single key and plaintext is repeated 32 times and DUT power consumption is averaged over the 32 traces.

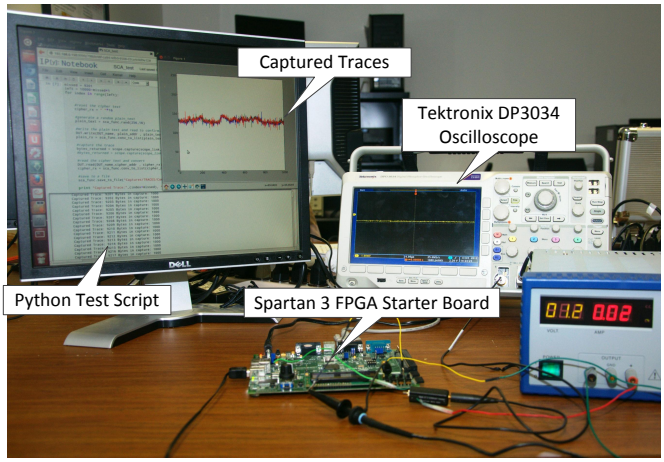


Figure 8.2: Test environment setup for CPA experiment: Photograph

8.1.2 Test Script

The complete test environment setup for this demonstration includes a control PC, DUT board, and oscilloscope. The control PC executes the test script and includes the test script interface for interaction with the other components. The DUT board is a Xilinx Spartan 3 starter kit FPGA board configured with the DUT, as well as the DUT interface. The oscilloscope is used as the attack equipment and measures power consumption of the DUT during AES encryption. A block diagram of the complete test setup is shown in Figure 8.1 and a photo of the physical setup is shown in Figure 8.2.

In this demonstration, the test environment collects traces from the DUT and stores them in the database. After completing trace collection, traces are read from the database and analyzed using CPA to extract the encryption key. This requires two separate test scripts; one for data collection and one for data analysis. The data collection test script provides a straightforward example of the interaction between all test environment components.

The data collection test script follows the general test procedure described in Section 7.1.1. The test script first initiates a connection with the DUT over USB and the oscilloscope over IP. Then, the script sets the parameters of the oscilloscope to capture traces in binary format averaging over 32 points and using one byte per sample and one thousand samples per trace. After this initialization process is complete, the test script sends a random encryption key to the DUT. This key remains constant for all encryptions performed during this experiment. The test script then enters the data collection loop, where it performs iterations of the following operations:

- Send random plaintext message to the DUT
- Read captured power trace from the oscilloscope

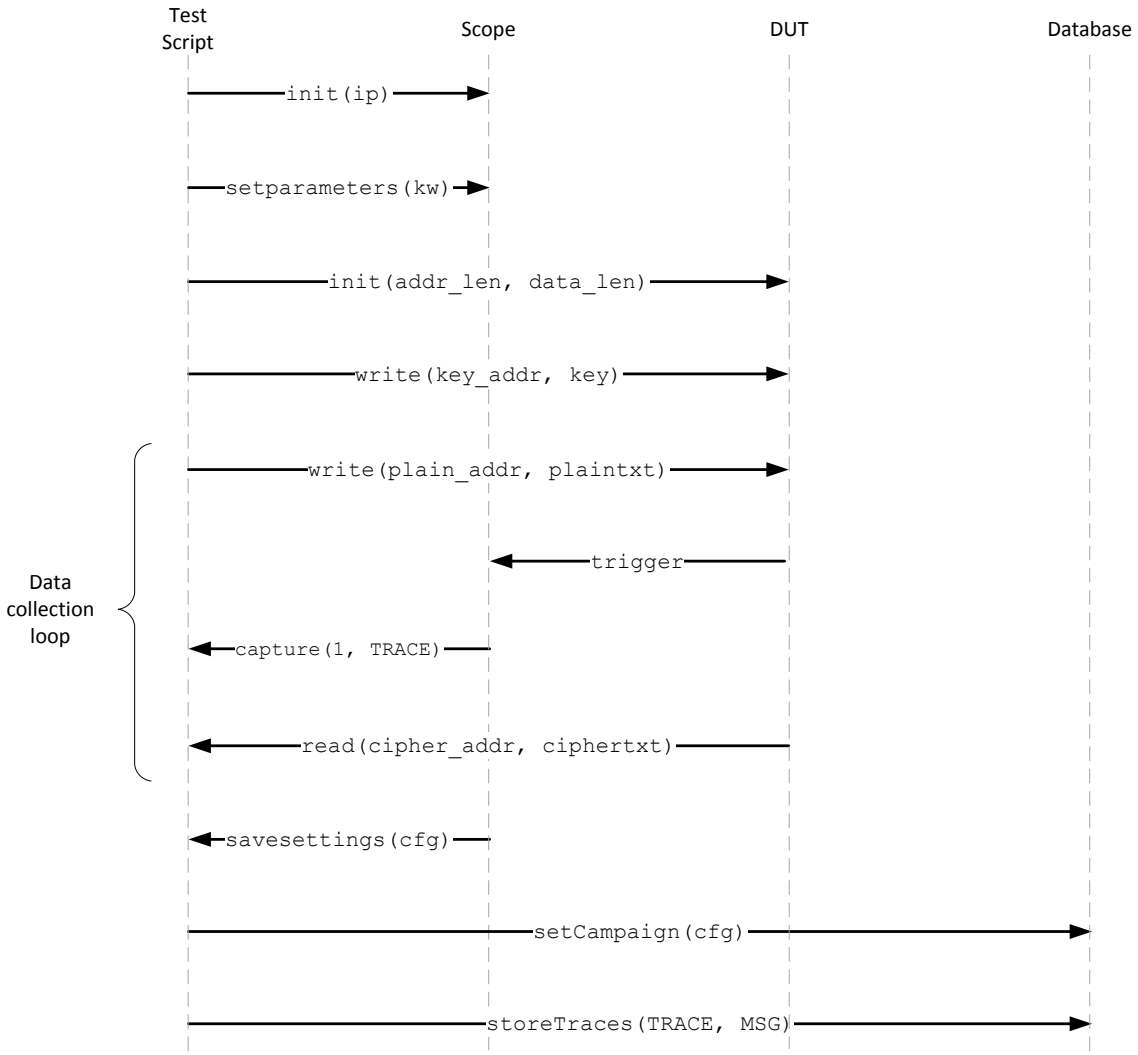


Figure 8.3: Test script sequence diagram for CPA experiment

- Append captured trace data to a file containing all previously collected traces
- Read ciphertext from the DUT
- Append plaintext and ciphertext to a file containing all previous messages

After completion of the data collection loop, the test script stores the collected trace data, along with general information about the experiment, in the SQL database. The traces are stored using the database API function that accepts multiple traces in a single buffer and separates them into a trace table entry corresponding to each measurement. This allows the test script to store all collected traces with a single call to the API by providing the file containing all collected traces, along with the file containing all plaintext and ciphertext pairs. The test script stores general data about the experiement and test environment setup, such as oscilloscope setting read directly from the scope and a link to the DUT source code, in the database experiment table.

Integration of the test environment modules, including DUT, attack equipment, and database, is achieved using the test script, which executes API function calls for each component. When an API function call is executed, the corresponding module interface is engaged to complete the required data transfer. The sequence diagram for the CPA data collection test script is shown in Figure 8.3. The sequence diagram illustrates the interactions between test environment components and gives the API functions used to complete each data transfer. The test script is independent of the operations performed by the DUT and attack equipment. This ensures that the test environment remains both flexible and portable; changing the DUT platform or symmetric key encryption algorithm implemented in the DUT does not impact the sequence diagram in Figure 8.3.

Python source code of the test script is shown in Listing 8.1. As shown, the test script uses the high-level API functions to perform the required operations and relies on the test script interface to implement low-level communication with the DUT and attack equipment interfaces.

```
1  import scope
2  import DUT
3  import sql
4  import sca_func
5
6  #SCRIPT PARAMETERS
7  TOTAL_TRACES = 3000
8  SAMPLES_PER_TRACE = 1000
9  BYTES_PER_SAMPLE = 1
10 traceFile = 'traces'
11 assocFile = 'msgFile.txt'
12
13 #INIT
14 #Initialize connection to oscilloscope
15 IP = '192.168.0.98'
```

```
16 s = scope.init(IP)
17 s.set_parameters(num_avg = 32, record_length=SAMPLES_PER_TRACE,
       bytes_per_point=BYTES_PER_SAMPLE)
18
19 #Initialize connection to DUT
20 DUT_name = "Digilent Spartan 3E"
21 DUT.init(DUT_name, addr_len, data_len)
22 #Send KEY to DUT
23 key = sca_func.rand(256, 16)
24 DUT.write(DUT_name, key_addr, key)
25
26 #DATA COLLECTION LOOP
27 index = 0
28 while i <= TOTAL_TRACES:
29     #Generate a random plain_text
30     plaintxt = sca_func.rand(256,16)
31     try:
32         #Write plaintext to DUT
33         DUT.write(DUT_name, plain_addr, plaintxt)
34         #Get the Captured Trace from Scope
35         buf = s.capture(save_capture=1, traceFile)
36         #Read ciphertext
37         DUT.read(DUT_name, cipher_addr, ciphertxt)
38         #Save to a file
39         sca_func.save_to_file(assocFile, (index), key, plaintxt,
               ciphertxt)
40     except IOError:
41        #Error in scope connection, reconnect
42         s = scope.init('192.168.0.98')
43     else:
44         index=index+1
45
46 #DATA STORAGE
47 db = sql.SQL()
48 s.save_settings('scopesettings.txt')
49
50 #Create experiment table entry in database
51 scopeSettings = open('scopesettings.txt').read()
52 comment = open('expcomment.txt').read()
53 db.setCampaign(scopeSettings, DUT_URL, DUT_REV, expComment,
       SAMPLES_PER_PT)
54
55 #Create traces table entries in database
56 db.storeTraces(traceFile, assocFile, SAMPLES_PER_TRACE, TOTAL_TRACES)
57
58 #Close connection to DUT and Oscilloscope
59 s.close()
60 DUT.close(DUT_name)
```

Listing 8.1: Data collection test script for CPA on AES

8.1.3 Results

The CPA attack targets the add round key and S-box lookup operations in the first round of AES encryption. During these operations, the plaintext is the known input to the encryption and the internal state of the algorithm is given by output of the S-box lookup for byte i as

$$state_i = \text{SBOX}(key_i \text{ XOR } plaintext_i).$$

Therefore, it is possible to compute the resulting internal state byte for each possible value of key byte i and estimate the power consumption of the operation as $\text{HW}(state_i)$ using the Hamming weight model. The correct value of the key byte is the one that shows highest correlation with the actual power consumption.

A simple analysis script is used to retrieve the power traces from the database and perform the CPA attack. The attack targets each key byte separately and uses a Hamming weight power model to compute the correlation between the measured power traces and the Hamming weight of the first round S-box output for each possible value of the key byte. For each key byte, the key value guess is the value showing the highest correlation with the power traces. The success of the attack is measured by the number of bytes for which the key value guess matches the actual encryption key used during data collection.

In this experiement, all key bytes were correctly guessed after analysis of 2000 power traces from the DUT, which indicates that the DUT is not secure against CPA side-channel attacks. This result demonstrates that the proposed test environment is well-suited for practical use in implementation attack resistance testing. An important feature of this demonstration is that the analysis script is completely independent of the data collection script and neither depends on the implementation details of the DUT or its platform.

8.2 Further Applications

Implementation attack strategies vary widely and new attacks are constantly being proposed. An important feature of the proposed test environment is its flexibility to allow use with any kind of implementation attack, secure device, and cryptographic algorithm. This section presents examples of well-known implementation attacks and describes how the proposed test environment can be used to evaluate device vulnerability to each attack.

8.2.1 Power and Electromagnetic Attack

Side-channel attacks using power consumption or electromagnetic radiation are the most common implementation attacks against embedded systems. These attacks are based on the observation that small fluctuations in device power consumption or electromagnetic radiation

during an encryption can reveal information about the internal state of the algorithm that can be used to derive the secret key. Vulnerability to power and electromagnetic attacks can be evaluated in terms of the number of measurement traces required to find the full key. In the previous section, use of the proposed test environment to perform CPA was demonstrated. Test setup for other forms of power and electromagnetic attacks is identical to the setup used for CPA; the difference in attack strategy is manifested in the analysis script, rather than the test setup.

8.2.2 Timing Attack

Timing attacks are a form of side-channel attacks that exploit variations in execution time of secure operations. Timing variations occur due to data-dependent software branching, as well as system architecture. During the AES selection process, analysis by Daemen and Rijmen concluded that Rijndael, later selected as AES, was not vulnerable to timing attacks [19]. However, Bernstein has since demonstrated a simple, yet effective timing attack against a server performing AES encryption [6]. Bernstein's attack exploits the timing dependence of table lookups, particularly due to the data cache used in general purpose CPUs. Table lookups resulting in a cache miss will take measurably longer than lookups resulting in a cache hit. Therefore, detailed analysis of encryption timing and its relationship with plaintext data values allows extraction of the complete encryption key. With improvements to this attack by Bonneau and Mironov, a server's key can be found with 2^{13} encryptions [9].

The AES cache timing attack requires precision timing measurements, a general purpose CPU with cached memory, and knowledge of plaintext and ciphertext. The proposed test environment can evaluate vulnerability to this attack as follows. The DUT used by both Bernstein [6] and Bonneau and Mironov [9] is a commercial CPU performing AES encryption using the OpenSSL library. Required attack equipment is a high-precision clock for measuring execution time of each encryption performed by the DUT. The test script has the DUT perform a number of encryptions and stores the plaintext, ciphertext, and execution time for each. This information can be used directly to perform analysis according to specific techniques described in the literature [6] [9].

8.2.3 Cold Boot Attack

Cold boot attacks, introduced by Halderman et al. [25], attack desktop computers using an approach similar to SCA. This attack allows observation of sensitive data in RAM immediately after shut down and reboot of a system. Due to the physical properties of RAM hardware, data remnants are preserved for a short time after power removal, which allows an attacker to extract sensitive data after an incomplete shutdown. The attack is successful because encryption keys are stored in RAM during execution of sensitive operations. Results in [25] demonstrate that this attack can be used to defeat widely used full disk encryption

schemes and suggest that any sensitive data in memory are vulnerable.

Test setup to perform a cold boot attack requires physical access to a desktop computer with RAM and software to dump RAM contents to a file for analysis. In some cases, cooling equipment may also be necessary to prevent RAM contents from fading before they can be recorded by the attack software. This attack can be incorporated into the proposed test environment by treating the RAM as the DUT with the desktop computer as the DUT platform. This attack requires multiple pieces of attack equipment including attack software that records RAM contents, power supply interrupter for the DUT, and cooling equipment. All communication with each piece of attack equipment is incorporated into the attack equipment interface in the test environment. In this setup, a trace is a file dump of all RAM contents at the time of power removal. To implement the cold boot attack, the test script would use the attack equipment to remove and quickly restore power supply from the DUT, then the test script would direct the attack equipment to execute the attack software and record the contents of the RAM.

8.2.4 Optical Fault Attack

Optical fault induction is a semi-invasive fault attack proposed by Skorobogatov and Anderson [54]. This attack allows manipulation of any individual SRAM bit using a laser. The practical significance of the attack is that it can allow the attacker to change the control flow of cryptographic operations by flipping selected bits in the on-chip instruction memory. This can make significant changes to computations performed during the cryptographic operation that allow recovery of the secret key. Although it requires depackaging the chip, the semi-invasive optical fault induction attack, unlike invasive attacks, which are typically very expensive to implement, does not require mechanical manipulation of the chip silicon and can be performed with low-cost off-the-shelf components.

Test setup for an optical fault induction attack requires a depackaged chip, laser, and knowledge of chip input and output. The proposed test environment can accommodate this attack by defining the depackaged chip as the DUT and the laser as the attack equipment. The attack equipment interface allows control of the exact chip location exposed to the laser and the DUT interface allows transmission of input and output data to and from the DUT. Without direct knowledge of the device design, optical fault induction attacks require some exploration to determine the impact of different SRAM cells on the cryptographic module under attack. The proposed test environment can also be used by incorporating a test script that induces faults in different SRAM cells and analyzes the impact on the final output of the device. Using this method, a complete attack can be mounted once the attacker determines the SRAM manipulations required to produce output that reveals the secret key.

Chapter 9

Conclusion

Cryptanalysis is an active field of research that aims to establish the practical security of cryptographic algorithms by evaluating their vulnerability to brute force, logical, and implementation attacks. A cryptanalytic system requires a complex design that must be highly optimized in order to provide an estimate of attack difficulty and the capabilities of a malicious attacker. Design space exploration is an essential component of system optimization that investigates the impact of various parameters on overall performance and identifies the specific parameter values that yield best results. Flexibility to support this experimentation is an important feature to allow easier optimization of a complex cryptanalytic design. It is also important that cryptanalytic systems minimize the design cost of system infrastructure, as this allows designers to focus their efforts on performance-critical components.

In this thesis, two distinct approaches are presented to improve the design methodology for cryptanalytic systems. First, Bluespec is proposed as an alternative to conventional HDLs to allow hardware designs at a higher abstraction level than RTL and facilitate the design space exploration necessary to optimize system architecture. HDL coding is very low-level compared to the complexity handled in a typical ECDLP architecture and its bottom-up design flow and low-level coding of control do not encourage design space exploration. An ECDLP implementation using Bluespec was presented and several factors, including the separation of interface from behavior, and the flexible specification of rule-based control, that seem to confirm the suitability of Bluespec for complex design tasks were discussed. In addition, the feasibility of quick design space exploration for a Bluespec-based design is demonstrated. Future work includes the evaluation of more complicated Pollard rho enhancements (such as negation maps [5], and/or tag tracing [16]) using Bluespec. Another open question is whether Bluespec is accessible as a parallel design language to software-oriented designers.

A test environment for implementation attacks has also been presented. The proposed test environment is constructed using a modular design process that separates the DUT from the implementation attack test procedure, allowing each to be designed independently. The

proposed design is focused on preserving portability and openness in the test environment and the modular design process ensures that the environment is applicable to any type of implementation attack, regardless of the DUT platform, implementation details, or cryptographic algorithm used. The proposed test environment is suitable for use in conjunction with implementation attack security standards to allow security validation that provides uniform, repeatable, and comparable results. The capabilities of the proposed design are demonstrated by presenting results for an actual CPA attack on AES and describing its application to other forms of implementation attacks. The proposed test environment improves the design process by providing a flexible infrastructure that is independent of device platform and type of implementation attack, eliminating the time-consuming process of constructing a unique set up for each implementation attack test scenario. This allows designers to focus their efforts on critical system components, namely design of the DUT and analysis test script. Future work includes practical application of the proposed test environment to other forms of implementation attacks, such as fault and probing attacks, and establishment of a library of standard data collection and analysis scripts to compare security of different implementations.

References

[1] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on RSA with CRT: Concrete results and practical countermeasures," in *Cryptographic Hardware and Embedded Systems (CHES), Workshop on*, ser. Lecture Notes in Computer Science, B. Kaliski, e. Ko, and C. Paar, Eds. Springer Berlin / Heidelberg, 2003, vol. 2523, pp. 81–95, 10.1007/3-540-36400-5_20. [Online]. Available: http://dx.doi.org/10.1007/3-540-36400-5_20

[2] F. Bao, R. Deng, Y. Han, A. Jeng, A. Narasimhalu, and T. Ngair, "Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults," in *Security Protocols, International Workshop on*, ser. Lecture Notes in Computer Science, B. Christianson, B. Crispo, M. Lomas, and M. Roe, Eds. Springer Berlin / Heidelberg, 1998, vol. 1361, pp. 115–124, 10.1007/BFb0028164. [Online]. Available: http://dx.doi.org/10.1007/BFb0028164

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370 –382, Feb. 2006. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2005.862424

[4] D. Bernstein, H.-C. Chen, C.-M. Cheng, T. Lange, R. Niederhagen, P. Schwabe, and B.-Y. Yang, "ECC2K-130 on Nvidia GPUs," in *Progress in Cryptology - INDOCRYPT 2010*, ser. Lecture Notes in Computer Science, G. Gong and K. Gupta, Eds. Springer Berlin / Heidelberg, 2010, vol. 6498, pp. 328–346. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17401-8_23

[5] D. Bernstein, T. Lange, and P. Schwabe, "On the correct use of the negation map in the Pollard rho method," in *Public Key Cryptography - PKC 2011*, ser. Lecture Notes in Computer Science, D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, Eds. Springer Berlin / Heidelberg, 2011, vol. 6571, pp. 128–146. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19379-8_8

[6] D. J. Bernstein, "Cache-timing attacks on AES," Tech. Rep., Apr. 14 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[7] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology CRYPTO '97*, ser. Lecture Notes in Computer Science, B. Kaliski, Ed. Springer Berlin / Heidelberg, 1997, vol. 1294, pp. 513–525, 10.1007/BFb0052259. [Online]. Available: http://dx.doi.org/10.1007/BFb0052259

[8] I. Blake, G. Seroussi, N. Smart, and J. W. S. Cassels, *Advances in Elliptic Curve Cryptography*, 2nd ed., ser. London Mathematical Society Lecture Note Series. New York, NY, USA: Cambridge University Press, May 2005.

[9] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems (CHES), Workshop on*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds. Springer Berlin / Heidelberg, 2006, vol. 4249, pp. 201–215, 10.1007/11894063_16. [Online]. Available: http://dx.doi.org/10.1007/11894063_16

[10] J. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe, "ECC2K-130 on Cell CPUs," in *Progress in Cryptology - AFRICACRYPT 2010*, ser. Lecture Notes in Computer Science, D. Bernstein and T. Lange, Eds. Springer Berlin / Heidelberg, 2010, vol. 6055, pp. 225–242. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12678-9_14

[11] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery, "On the security of 1024-bit RSA and 160-bit elliptic curve cryptography," IACR Cryptology ePrint Archive, Report 2009: 389, 2009, http://eprint.iacr.org/2009/389.

[12] J. Bos, M. Kaihara, T. Kleinjung, A. Lenstra, and P. Montgomery, "Solving a 112–bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction," *International Journal of Applied Cryptography*, vol. 2, no. 3, pp. 212–228, 2012. [Online]. Available: http://dx.doi.org/10.1504/IJACT.2012.045590

[13] R. P. Brent, "An improved Monte Carlo factorization algorithm," *BIT Numerical Mathematics*, vol. 20, pp. 176–184, 1980, 10.1007/BF01933190. [Online]. Available: http://dx.doi.org/10.1007/BF01933190

[14] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems (CHES), Workshop on*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds. Springer Berlin / Heidelberg, 2004, vol. 3156, pp. 135–152, 10.1007/978-3-540-28632-5_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28632-5_2

[15] Brightsight, "Unique tools from the security lab." [Online]. Available: http://www.brightsight.com/documents/marcom-materials/Brightsight_Tools.pdf

[16] J. Cheon, J. Hong, and M. Kim, "Speeding up the Pollard rho method on prime fields," in *Advances in Cryptology - ASIACRYPT 2008*, ser. Lecture Notes in Computer Science, J. Pieprzyk, Ed., vol. 5350. Springer Berlin / Heidelberg, 2008, pp. 471–488. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89255-7_29

[17] C. Clavier, "Side Channel Analysis for Reverse Engineering (SCARE) - an improved attack against a secret A3/A8 GSM algorithm," Cryptology ePrint Archive, Report 2004/049, 2004, http://eprint.iacr.org/.

[18] Cryptography Research, "DPA workstation." [Online]. Available: http://www.cryptography.com/technology/dpa-workstation.html

[19] J. Daemen and V. Rijmen, "Resistance against implementation attacks: a comparative study of the AES proposals," in *Second AES Candidate Conference (AES2)*. National Institute of Standards and Technology (NIST), Mar. 22-23 1999. [Online]. Available: http://csrc.nist.gov/archive/aes/round1/conf2/papers/daemen.pdf

[20] W. Diffie and M. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644 – 654, Nov 1976. [Online]. Available: http://dx.doi.org/10.1109/TIT.1976.1055638

[21] J. Fan, D. Bailey, L. Batina, T. Güneysu, C. Paar, and I. Verbauwhede, "Breaking elliptic curve cryptosystems using reconfigurable hardware," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE Computer Society, Aug. 31, 2010-Sept. 2, 2010, pp. 133–138. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/FPL.2010.34

[22] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Cryptographic Hardware and Embedded Systems (CHES), Workshop on*, ser. Lecture Notes in Computer Science, E. Oswald and P. Rohatgi, Eds. Springer Berlin / Heidelberg, 2008, vol. 5154, pp. 62–78. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85053-3_5

[23] T. Güneysu, C. Paar, and J. Pelzl, "Attacking elliptic curve cryptosystems with special-purpose hardware," in *Field programmable gate arrays (FPGA), 2007 ACM/SIGDA 15th international symposium on*. New York, NY, USA: ACM, 2007, pp. 207–215. [Online]. Available: http://dx.doi.org/10.1145/1216919.1216953

[24] T. Güneysu, C. Paar, and J. Pelzl, "Special-purpose hardware for solving the elliptic curve discrete logarithm problem," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 1, no. 2, pp. 8:1–8:21, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1145/1371579.1371580

[25] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys." *Communications of the ACM*, vol. 52, no. 5, pp. 91 – 98, 2009. [Online]. Available: http://dx.doi.org/10.1145/1506409.1506429

[26] H. Handschuh, P. Paillier, and J. Stern, "Probing attacks on tamper-resistant devices," in *Cryptographic Hardware and Embedded Systems (CHES), Workshop on*,

ser. Lecture Notes in Computer Science, e. Ko and C. Paar, Eds. Springer Berlin / Heidelberg, 1999, vol. 1717, pp. 727–727, 10.1007/3-540-48059-5_26. [Online]. Available: http://dx.doi.org/10.1007/3-540-48059-5_26

[27] D. R. Hankerson, S. A. Vanstone, and A. J. Menezes, *Guide to elliptic curve cryptography*. New York, NY, USA: Springer, 2004.

[28] L. Judge, M. Cantrell, C. Kendir, and P. Schaumont, "A modular testing environment for implementation attacks," in *Workshop on Redefining and Integrating Security Engineering (RISE) at ASE/IEEE International Conference on Cyber Security 12*, Washington DC, USA, Dec. 14-16 2012.

[29] L. Judge, S. Mane, and P. Schaumont, "A hardware accelerated ECDLP with high-performance modular multiplication," *International Journal of Reconfigurable Computing*, vol. 2012, 2012. [Online]. Available: http://www.hindawi.com/journals/ijrc/2012/439021/

[30] L. Judge and P. Schaumont, "A flexible hardware ECDLP engine in Bluespec," in *Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS), Workshop on*, Washington DC, USA, Mar. 17-18 2012. [Online]. Available: http://2012.sharcs.org/record.pdf

[31] A. Kerckhoffs, "La cryptographie militaire (part 1)," *Journal des sciences militaires*, vol. IX, pp. 5–38, Jan 1883.

[32] ——, "La cryptographie militaire (part 2)," *Journal des sciences militaires*, vol. IX, pp. 161–191, Feb 1883.

[33] J. Kim, K. Oh, D. Choi, and H. Kim, "SCARF: profile-based side channel analysis resistant framework," in *Security and Management (SAM'12), 2012 International Conference on*, Jul. 16-19 2012.

[34] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. pp. 203–209, Jan. 1987. [Online]. Available: http://www.jstor.org/stable/2007884

[35] P. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology (CRYPTO96)*, ser. Lecture Notes in Computer Science, N. Koblitz, Ed. Springer Berlin / Heidelberg, 1996, vol. 1109, pp. 104–113. [Online]. Available: http://dx.doi.org/10.1007/3-540-68697-5_9

[36] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology CRYPTO 99*, ser. Lecture Notes in Computer Science, M. Wiener, Ed. Springer Berlin / Heidelberg, 1999, vol. 1666, pp. 789–789, 10.1007/3-540-48405-1_25. [Online]. Available: http://dx.doi.org/10.1007/3-540-48405-1_25

[37] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, "Breaking ciphers with COPACOBANA a cost-optimized parallel code breaker," in *Cryptographic Hardware and Embedded Systems (CHES), Workshop on*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds. Springer Berlin / Heidelberg, 2006, vol. 4249, pp. 101–118. [Online]. Available: http://dx.doi.org/10.1007/11894063_9

[38] P. Majkowski, M. Rawski, T. Wojciechowski, Z. Kotulski, and M. Wojtynski, "Heterogenic distributed system for cryptanalysis of elliptic curve based cryptosystems," in *Systems Engineering, 2008 (ICSENG '08), 19th International Conference on*, Las Vegas, Nevada, USA, Aug. 19-21, 2008, pp. 300 –305. [Online]. Available: http://dx.doi.org/10.1109/ICSEng.2008.73

[39] S. Mane, L. Judge, and P. Schaumont, "An integrated prime-field ECDLP hardware accelerator with high-performance modular arithmetic units," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, P. M. Athanas, J. Becker, and R. Cumplido, Eds. Cancun, Quintana Roo, Mexico: IEEE Computer Society, Nov. 30-Dec. 2, 2011, pp. 198–203. [Online]. Available: http://dx.doi.org/10.1109/ReConFig.2011.12

[40] A. Menezes, P. Van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, ser. Discrete Mathematics and Its Applications. Boca Raton, FL: CRC Press, Oct. 1996.

[41] G. Meurice de Dormale, P. Bulens, and J.-J. Quisquater, "Collision search for elliptic curve discrete logarithm over GF(2^m) with FPGA," in *Cryptographic Hardware and Embedded Systems (CHES), Workshop on*, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds. Springer Berlin / Heidelberg, 2007, vol. 4727, pp. 378–393. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74735-2_26

[42] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology CRYPTO 85 Proceedings*, ser. Lecture Notes in Computer Science, H. Williams, Ed. Springer Berlin / Heidelberg, 1986, vol. 218, pp. 417–426. [Online]. Available: http://dx.doi.org/10.1007/3-540-39799-X_31

[43] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987. [Online]. Available: http://dx.doi.org/10.2307/2007888

[44] National Institute of Standards and Technology (NIST), "FIPS 140-3 (second draft) sections submitted for comments," Aug. 30 2012. [Online]. Available: http://csrc.nist.gov/groups/ST/FIPS140_3/documents/FIPS_140-3_sections_submitted_for_comments.pdf

[45] *Cryptographic Hash Algorithm Competition*, National Institute of Standards and Technology (NIST), 2007-2012. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

[46] *FIPS 186-3: Digital Signature Standard (DSS)*, National Institute of Standards and Technology (NIST), Jun. 2009. [Online]. Available: http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

[47] *Security Requirements for Cryptographic Modules (Revised Draft)*, National Institute of Standards and Technology (NIST), 2009. [Online]. Available: http://csrc.nist.gov/publications/PubsDrafts.html#FIPS-140--3

[48] J. Nechvatal, E. B. L. Bassham, M. Dworkin, J. Foti, and E. Roback, *Report on the Development of the Advanced Encryption Standard (AES)*, National Institute of Standards and Technology (NIST), Oct. 2 2000. [Online]. Available: http://csrc.nist.gov/archive/aes/round2/r2report.pdf

[49] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 129–146, 10.1007/978-1-4020-8588-8_8. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-8588-8_8

[50] J. M. Pollard, "Monte carlo methods of index computation (mod p)," *Mathematics of Computation*, vol. 32, no. 143, pp. 918–924, 1978. [Online]. Available: http://www.jstor.org/stable/2006496

[51] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (EMA): Measures and counter-measures for smart cards," in *Smart Card Programming and Security*, ser. Lecture Notes in Computer Science, I. Attali and T. Jensen, Eds. Springer Berlin / Heidelberg, 2001, vol. 2140, pp. 200–210, 10.1007/3-540-45418-7_17. [Online]. Available: http://dx.doi.org/10.1007/3-540-45418-7_17

[52] Riscure, "Inspector." [Online]. Available: http://www.riscure.com/tools/inspector

[53] J.-M. Schmidt and C. Kim, "A probing attack on AES," in *Information Security Applications*, ser. Lecture Notes in Computer Science, K.-I. Chung, K. Sohn, and M. Yung, Eds. Springer Berlin / Heidelberg, 2009, vol. 5379, pp. 256–265, 10.1007/978-3-642-00306-6_19. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00306-6_19

[54] S. Skorobogatov and R. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems (CHES), Workshop on*, ser. Lecture Notes in Computer Science, B. Kaliski, e. Ko, and C. Paar, Eds. Springer Berlin / Heidelberg, 2003, vol. 2523, pp. 31–48, 10.1007/3-540-36400-5_2. [Online]. Available: http://dx.doi.org/10.1007/3-540-36400-5_2

[55] *SEC 2: Recommended Elliptic Curve Domain Parameters*, Standards for Efficient Cryptography Group (SECG), Jan. 2010. [Online]. Available: http://www.secg.org/download/aid-784/sec2-v2.pdf

[56] *DPA contest*, Télécom ParisTech, 2008-2012. [Online]. Available: http://www.dpacontest.org

[57] E. Teske, "On random walks for Pollard's rho method," *Mathematics of Computation*, vol. 70, no. 234, pp. 809–825, Apr 2001. [Online]. Available: http://dx.doi.org/10.1090/S0025-5718-00-01213-8

[58] ——, "Speeding up Pollard's rho method for computing discrete logarithms," in *Algorithmic Number Theory*, ser. Lecture Notes in Computer Science, J. Buhler, Ed. Springer Berlin / Heidelberg, 1998, vol. 1423, pp. 541–554, 10.1007/BFb0054891. [Online]. Available: http://dx.doi.org/10.1007/BFb0054891

[59] P. C. van Oorschot and M. J. Wiener, "Parallel collision search with cryptanalytic applications," *Journal of Cryptology*, vol. 12, pp. 1–28, 1999. [Online]. Available: http://dx.doi.org/10.1007/PL00003816

[60] R. Velegalati and J.-P. Kaps, "Introducing FOBOS: Flexible Open-source BOard for Side-channel analysis," in *Constructive Side-Channel Analysis and Secure Design (COSADE), Third International Workshop on: Work in Progress Session*, May 3 2012.

[61] Xilinx, "MicroBlaze soft processor core." [Online]. Available: http://www.xilinx.com/tools/microblaze.htm

[62] *Virtex-5 FPGA XtremeDSP Design Considerations*, Xilinx, Inc., Jan. 2012, http://www.xilinx.com/support/documentation/user_guides/ug193.pdf.

Appendix A

Modmul.bsv : Bluespec System Verilog listing for Modular Multiplier

```
1  interface Multiplier;
2    method Action load(Bit#(18) f, Bit#(16) g);
3    method Bit#(34) result();
4  endinterface
5
6  (* synthesize *)
7  (* always_ready = load, result *)
8  module mkMultiplier(Multiplier);
9    Reg#(Bit#(34)) product <- mkReg(0);
10
11   method Action load(Bit#(18) f, Bit#(16) g);
12     product <= {0, f} * {0, g};
13   endmethod
14
15   method Bit#(34) result();
16     return product;
17   endmethod
18 endmodule
19
20 interface Accumulator;
21   method Action add(Bit#(20) f, Bit#(16) g);
22   method Action reset();
23   method Bit#(21) result();
24 endinterface
25
26 (* synthesize *)
```

```
27 (* always_ready = add, reset, result *)
28 module mkAccumulator(Accumulator);
29   Reg#(Bit#(21)) sum <- mkReg(0);
30
31   method Action add(Bit#(20) f, Bit#(16) g);
32     sum <= sum + {0, f} + {0, g};
33   endmethod
34
35   method Action reset();
36     sum <= 0;
37   endmethod
38
39   method Bit#(21) result();
40     return sum;
41   endmethod
42 endmodule
43
44 interface Adder;
45   method Action add(Bit#(21) f, Bit#(6) g);
46   method Bit#(22) result();
47 endinterface
48
49 (* synthesize *)
50 (* always_ready = add, result *)
51 module mkAdder(Adder);
52   Reg#(Bit#(22)) sum <- mkReg(0);
53
54   method Action add(Bit#(21) f, Bit#(6) g);
55     sum <= {0, f} + {0, g};
56   endmethod
57
58   method Bit#(22) result();
59     return sum;
60   endmethod
61 endmodule
62
63 interface ModMul;
64   method Action load(Bit#(128) f_in, Bit#(128)g_in);
65   method Maybe#(Bit#(128)) result();
66 endinterface
67
68 (* synthesize *)
```

```
69 (* always_ready = load *)
70 module mkModMul(ModMul);
71   Reg#(Bit#(144)) f <- mkReg(0);
72   Reg#(Bit#(128)) g <- mkReg(0);
73   Reg#(Bit#(4)) rcnt <- mkReg(15);
74   Reg#(Bit#(128)) prev_res <- mkReg(0);
75   Wire#(Bit#(34)) wrap_around <- mkWire();
76   Wire#(Bit#(34)) carry_wrap <- mkWire();
77   Wire#(Bit#(34)) mod_f <- mkWire();
78   Reg#(Maybe#(Bit#(128))) cout <- mkWire;
79   Multiplier m0 <- mkMultiplier();
80   Multiplier m1 <- mkMultiplier();
81   Multiplier m2 <- mkMultiplier();
82   Multiplier m3 <- mkMultiplier();
83   Multiplier m4 <- mkMultiplier();
84   Multiplier m5 <- mkMultiplier();
85   Multiplier m6 <- mkMultiplier();
86   Multiplier m7 <- mkMultiplier();
87   Accumulator a0 <- mkAccumulator();
88   Accumulator a1 <- mkAccumulator();
89   Accumulator a2 <- mkAccumulator();
90   Accumulator a3 <- mkAccumulator();
91   Accumulator a4 <- mkAccumulator();
92   Accumulator a5 <- mkAccumulator();
93   Accumulator a6 <- mkAccumulator();
94   Accumulator a7 <- mkAccumulator();
95   Adder c0 <- mkAdder();
96   Adder c1 <- mkAdder();
97   Adder c2 <- mkAdder();
98   Adder c3 <- mkAdder();
99   Adder c4 <- mkAdder();
100  Adder c5 <- mkAdder();
101  Adder c6 <- mkAdder();
102  Adder c7 <- mkAdder();
103
104  rule do_mult;
105    f <= {f[125:0], mod_f[17:0]};
106    g <= g >> 16;
107    m0.load(f[17:0], g[15:0]);
108    m1.load(f[35:18], g[15:0]);
109    m2.load(f[53:36], g[15:0]);
110    m3.load(f[71:54], g[15:0]);
```

```
111     m4.load(f[89:72], g[15:0]);
112     m5.load(f[107:90], g[15:0]);
113     m6.load(f[125:108], g[15:0]);
114     m7.load(f[143:126], g[15:0]);
115   endrule
116
117   rule wrap;
118     wrap_around <= 3 * {0, m7.result()[33:16]};
119     carry_wrap <= 3 * {0, c7.result()[21:16]};
120     mod_f <= 3 * {0, f[143:126]};
121   endrule
122
123   rule do_acc;
124     a0.add(wrap_around[19:0], m0.result()[15:0]);
125     a1.add({0, m0.result()[33:16]}, m1.result()[15:0]);
126     a2.add({0, m1.result()[33:16]}, m2.result()[15:0]);
127     a3.add({0, m2.result()[33:16]}, m3.result()[15:0]);
128     a4.add({0, m3.result()[33:16]}, m4.result()[15:0]);
129     a5.add({0, m4.result()[33:16]}, m5.result()[15:0]);
130     a6.add({0, m5.result()[33:16]}, m6.result()[15:0]);
131     a7.add({0, m6.result()[33:16]}, m7.result()[15:0]);
132   endrule
133
134   rule do_reduce;
135     c0.add(a0.result(), carry_wrap[5:0]);
136     c1.add(a1.result(), c0.result()[21:16]);
137     c2.add(a2.result(), c1.result()[21:16]);
138     c3.add(a3.result(), c2.result()[21:16]);
139     c4.add(a4.result(), c3.result()[21:16]);
140     c5.add(a5.result(), c4.result()[21:16]);
141     c6.add(a6.result(), c5.result()[21:16]);
142     c7.add(a7.result(), c6.result()[21:16]);
143   endrule
144
145   rule incr_counter (rcnt < 4'd15);
146     rcnt <= rcnt + 1;
147   endrule
148
149   rule assign_res;
150     if (rcnt == 4'd14)
151       cout <= tagged Valid ({c7.result()[15:0],
            c6.result()[15:0], c5.result()[15:0], c4.result()[15:0],
```

```
            c3.result()[15:0], c2.result()[15:0], c1.result()[15:0],
            c0.result()[15:0]});
152       else
153         cout <= tagged Invalid;
154     endrule
155
156     method Action load(Bit#(128) f_in, Bit#(128) g_in);
157       f <= {2'd0, f_in[127:112], 2'd0, f_in[111:96], 2'd0,
              f_in[95:80], 2'd0, f_in[79:64], 2'd0, f_in[63:48], 2'd0,
              f_in[47:32], 2'd0, f_in[31:16], 2'd0, f_in[15:0]};
158       g <= g_in;
159       a0.reset();
160       a1.reset();
161       a2.reset();
162       a3.reset();
163       a4.reset();
164       a5.reset();
165       a6.reset();
166       a7.reset();
167       rcnt <= 0;
168     endmethod
169
170     method Maybe#(Bit#(128)) result();
171       return cout;
172     endmethod
173 endmodule
```