EFFICIENT AND TAMPER-RESILIENT ARCHITECTURES FOR PAIRING BASED CRYPTOGRAPHY

by

Erdinc Ozturk

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy
in
Electrical and Computer Engineering
by

February 2009

APPROVED:

Prof. Berk Sunar, Major Advisor

Prof. Wenjing Lou

Prof. Xinming Huang

Prof. Wayne Burleson

Prof. Fred Looft, Department Head

Abstract

Identity based cryptography was first proposed by Shamir [36] in 1984. Rather than deriving a public key from private information, which would be the case in traditional public key encryption schemes, in identity based schemes a user's identity plays the role of the public key. This reduces the amount of computations required for authentication, and simplifies key-management. Efficient and strong implementations of identity based schemes are based around easily computable bilinear mappings of two points on an elliptic curve onto a multiplicative subgroup of a field, also called pairing.

The idea of utilizing the identity of the user simplifies the public key infrastructure. However, since pairing computations are expensive for both area and timing, the proposed identity based cryptosystem are hard to implement. In order to be able to efficiently utilize the idea of identity based cryptography, there is a strong need for an efficient pairing implementations.

Pairing computations could be realized in multiple fields. Since the main building block and the bottleneck of the algorithm is multiplication, we focused our research on building a fast and small arithmetic core that can work on multiple fields. This would allow a single piece of hardware to realize a wide spectrum of cryptographic algorithms, including pairings, with minimal amount of software coding. We present a novel unified core design which is extended to realize Montgomery multiplication in the fields $GF(2^n)$, $GF(3^m)$, and $GF(p)$. Our unified design supports RSA and elliptic curve schemes, as well as identity based encryption which requires a pairing computation on an elliptic curve. The architecture is pipelined and is highly scalable. The unified core utilizes the redundant signed digit representation to reduce the critical path delay. While the carry-save representation used in classical unified architectures is only good for addition and multiplication operations, the redundant signed digit representation also facilitates efficient computation of comparison and subtraction operations besides addition and multiplication. Thus, there is no need for transformation between the redundant and non-redundant representations of field elements, which would be required in classical unified architectures to realize the subtraction and comparison operations. We also quantify the benefits of unified architectures in terms of area and critical path delay. We provide detailed implementation results. The metric shows that the new unified architecture provides an improvement over a hypothetical non-unified architecture of at least 24.88% while the improvement over a classical unified architecture is at least 32.07%.

Until recently there has been no work covering the security of pairing based cryptographic hardware in the presence of side-channel attacks, despite their apparent suitability for identity-aware personal security devices, such as smart cards. We present a novel non-linear error coding framework which incorporates strong adversarial fault detection capabilities into identity based encryption schemes built using Tate pairing computations. The presented algorithms provide quantifiable resilience in a well defined strong attacker model. Given the emergence of fault attacks as a serious threat to pairing based cryptography, the proposed technique solves a key problem when incorporated into software and hardware implementations. In this dissertation, we also present an efficient accelerator for computing the Tate Pairing in characteristic 3, based on the Modified Duursma Lee algorithm.

Acknowledgements

First, I would like to thank my advisor and mentor Prof. Berk Sunar for his guidance and support and tolerance throughout the years. He was always available and he always made the effort to make sure that I am on the right path. It has been a privilege working with him and throughout my research I never doubted his guidance.

I am grateful to my dissertation committee members, Prof. Wayne Burleson, Prof. Wenjing Lou and Prof. Xinming Huang for the time and effort they spent on my dissertation. Their valuable suggestions and inputs have improved this dissertation.

I would particularly like to thank my lab mates Gunnar Gaubatz, Ghaith Hammouri and Jens Peter Kaps. They worked with me on many papers. I am also grateful to my other lab mates Kahraman D. Akdemir, Deniz Karakoyunlu and Selcuk Baktir. Kahraman and I were not only lab mates, we went to the same college, same grad school, shared the same lab and lived in the same house, I am truly grateful for all the help and support he gave me throughout the years. I would also like to thank my close friend Ipek for all her support, the times spent in AK would not be as bearable without her.

Nothing would work in the department without the amazing work of Robert Brown, the manager of computational facilities. He was always fast, accurate and thorough whenever we had a problem. I also would like to thank the administrative assistants Cathy Emmerton, Brenda McDonald, and Colleen Sweeney. Their constant supply of sweets in the office helped us feel better.

Most importantly, I would like to show my gratitude towards my family, my parents

Dursun and Gonul Ozturk, and my brothers Sinan and Celal Ozturk. They were always there for me, they always supported me and inspired me.

Lastly, I would like to thank Intel Corporation for supporting my research for over four years and the National Science Foundation for partially funding this research through the Career award ANI-0133297.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently there has been an increase in research activity on pairing based cryptography such as identity based cryptosystems [8]. Identity based cryptography was first proposed by Shamir [36] in 1984. Rather than deriving a public key from private information, which would be the case in traditional public key encryption schemes, in identity based schemes a user's identity plays the role of the public key. This reduces the amount of computations required for authentication, and simplifies key-management.

Elliptic curve and RSA (or Diffie-Hellman) schemes are typically implemented over $GF(p)$ or $GF(2^n)$ and over Z_n (or $GF(p)$). Numerous architectures were proposed to support arithmetic for elliptic curve cryptography and RSA-like schemes [34, 3]. Unified architectures for the fields $GF(p)$ and $GF(2^n)$ were also proposed [34, 17, 43, 33, 38, 35, 1]. However, the emergence of pairing based cryptography has attracted a significant level of interest in arithmetic in $GF(3^m)$. Hardware architectures for arithmetic in characteristic three have appeared in [28], [20], and [7].

Pairing based cryptography may utilize all of the three kinds of mathematical structures. Moreover, ECC and RSA schemes are typically implemented over prime or binary fields and integer rings, respectively. Thus, it would be highly desirable to have a single piece of unified hardware that supports arithmetic in all three kinds of domains simultaneously. To the best of our knowledge, such an architecture is still lacking.

While a unified architecture is highly desirable, the scalability and efficiency of the hardware is important. Here we use the notion of scalability as introduced in [37]. The design should scale without the redesign of the architecture, by simply increasing the number of processing units. The scalability feature along with the unified approach would allow the architecture to support a wide spectrum of operating points ranging from low-end and low-power devices to high-end server platforms. For efficiency reasons we design our architecture around a carry-free architecture. Furthermore, the scalable nature of the design allows pipelining techniques to be used to further improve efficiency. Our architecture supports the basic arithmetic operations (i.e., addition, multiplication and inversion) in the arithmetic extension fields $GF(p)$[1], $GF(2^n)$ and $GF(3^m)$. All operations are carried out in the residue space defined by the Montgomery multiplication algorithm [24].

Efficient implementations of identity based schemes are based around easily computable bilinear mappings of two points on an elliptic curve onto a multiplicative subgroup of a field, also called pairing. One mapping that seems to be of particular interest is the

[1]Since we do not make use of the field properties in our design, the architecture supports also arithmetic in integer rings Z_n and hence supports RSA.

modified Tate pairing by Duursma and Lee, not in the least due to the improved algorithms due to Kwon and Baretto et al.

Until recently there has been no work covering the security of pairing based cryptographic hardware in the presence of side-channel attacks, despite their apparent suitability for identity-aware personal security devices, such as smart cards. Indeed, Page and Vercauteren [29] for the first time investigate a fault attack on the modified Tate pairing in the context of the Baek-Zheng threshold decryption scheme. It allows the attacker to recover the private point value of the decryption servers with relatively little effort, which leads to the defacto compromise of the scheme.

1.1 Contributions

Contributions of this work are outlined as follows:

- In order to support Elliptic Curve Cryptography, factoring based cryptographic systems, public key primitives and recently proposed identity based schemes, we propose a new and more efficient unified multiplier that operates in three fields, namely $GF(p)$, $GF(2^n)$, and $GF(3^m)$. To the best of our knowledge, this is the first attempt to combine the arithmetic of these three, cryptographically important, finite fields in a single datapath.

- We present a metric to quantitatively demonstrate the advantages of the proposed unified multiplier over the classical unified multiplier that supports arithmetic only in $GF(p)$ and $GF(2^n)$. The unified architectures proposed so far [34, 17, 35, 43], lacked quantitative analysis of the advantage of using a unified approach. It has only been reported that unified architecture result in negligible overhead in area and critical path delay (CPD). In this work, we quantified the gain in terms of the area $\times$ CPD metric, which showed that the benefits of the new unified architecture far exceed that of the classical unified architecture.

- We utilize a different carry-free arithmetic that allows efficient comparison and subtraction operations in $GF(p)$-mode. The classical unified architectures [34, 17, 35, 43] utilize the carry-save representation in order to eliminate the carry propagation in $GF(p)$ mode. It is not easy to perform subtraction and comparison operations in the carry-save representation, where field elements are expressed as the sum of two integers. For instance, [43] transforms the elements of $GF(p)$ that are in carry-save form to non-redundant form by adding the number to itself repeatedly in order to perform comparison and subtraction operations necessary to realize other field operations such as multiplicative inversion. For our carry-free arithmetic, the field elements are represented as the difference of two field elements, instead of sum. This representation facilitates efficient subtraction and comparison operations. Consequently, all arithmetic operations in cryptographic computations can be performed without the need of transformations between redundant and non-redundant forms.

- We computed execution times of basic operations for three prominent public key cryptography algorithms: ECC scalar point multiplication, RSA exponentiation, and Tate pairing computations. The results show that Tate pairing computations used in identity based cryptosystems can be performed by the proposed unified architecture in a comparably efficient manner.

- We apply a specially parameterized variant of the original construction for robust codes from [18], as well as the construction from [14] to a modified Tate pairing implementation. Instead of only protecting the loop counter of the algorithm, we also protect the entire data path to avoid potential attacks on the base points $(\mathcal{P}, \mathcal{Q})$. The resulting architecture achieves a high degree of robustness against even highly motivated attackers, and at the same time only requires a moderate overhead in area and delay.

  In addition, a contribution of lesser importance is the introduction of scalable Montgomery algorithm for ternary extension field, $GF(3^m)$. Although it is a straightforward adaptation of the algorithm presented in [37] to ternary extension fields, it is the first attempt to formulate such an algorithm.

1.2 Motivation

The attack described in [29] is just the first one reported, therefore we must expect that new types of attacks will be discovered in the future. The current attack is focused mainly on the loop counter of the modified Duursma-Lee algorithm, but in principle there may be many more parts of the system that are vulnerable to a fault attack. To provide the highest level of assurance even under adversarial conditions, we need to protect the entire system with a robust error detection mechanism.

A common approach to prevent errors in cryptographic hardware involves the use of existing redundancy, for example by decrypting an encrypted result and comparing to the input. In other cases simple concepts from classic linear coding theory (parity prediction, Hamming codes) are applied to standard circuits, for low-cost protection against basic faults. For instance, a number techniques involving parity checking codes [31, 32] and a more general technique based on linear codes were proposed in [13].

With the advent of more advanced attack techniques and more accurate error and attacker models, these simple techniques may prove to be inadequate. Effective hardening of systems against sophisticated and malicious fault attacks requires strong error detection under worst case assumptions rather than average case. A general property of linear codes is that the sum of two codewords is again a codeword [41]. An error pattern with the same value as that of a codeword can thus never be detected. Another problematic aspect that arises out of this property is the code's behavior with regard to certain error patterns with Hamming weight a multiple of the codes minimum distance, e.g. burst errors. These errors are often not detectable and may occur with high probability.

In [18] Karpovsky and Taubin introduced a novel family of non-linear systematic error detecting codes, which provides robustness according to the minimax criterion, i.e. it minimizes the maximum probability of missing an error over all non-zero error patterns.

In addition to the probability of missing an error, the nonlinear encoding also makes the event of missing an error highly data dependent. Without a priori knowledge of the data it is therefore practically infeasible to induce an undetected error.

While the adoption of robust codes has been successful for symmetric cryptosystems like the AES block-cipher [22], robust implementations of public key cryptography has been challenging due to the seemingly incompatible arithmetic of the encoding procedure and public key operations. In [14] the authors for the first time gave a general construction for robust arithmetic codes versatile enough to be applied to any fixed width data-path for digit serial general purpose arithmetic. Using these codes it is possible to protect any type of integer ring or prime field arithmetic, e.g. RSA, Diffie-Hellman, Elliptic Curves over $GF(p)$ against active adversaries. While the codes achieve a high degree of robustness, they also impose a tremendous computational overhead, which may be too costly for practical implementations.

1.3 Dissertation Outline

We started our research with the basic building blocks for pairing based cryptosystems. Multiplication is the bottleneck of the cryptographic applications so we focused our research on efficient multiplier architectures. Since efficient pairing implementations work on fields other than prime and binary fields, we aimed at building an arithmetic core that would work on three important fields: prime binary and ternary. This would make the embedding of pairing algorithms into the cryptographic accelerators that utilize many cryptographic algorithms.

In Chapter 2, we are presenting brief and necessary background information that is used throughout our research. In Section 2.1, we explain the building blocks of arithmetic in characteristic three. We give detailed information for hardware implementations of ternary field arithmetic in this section. In Section 2.2, Tate Pairing algorithm and its modified version are explained. In Section 2.3 we introduce the traditional RSD representation and our notational conventions. We utilized the RSD representation to make better use of the carry-save architecture for our purposes.

Then in Chapter 3 our unified arithmetic core design is explained. We first explain the implementation of our core implementation. Then, we give detailed explanations of how to utilize our arithmetic core for basic operations such as addition, subtraction and comparison.

Chapter 4 presents the Montgomery multiplication algorithms for the three fields. For this architecture, we utilized our unified arithmetic core. In Section 4.1 we introduce the Montgomery multiplier design, and describe relevant system level architectural details such as pipelining and architectural scaling. We then present the complexity analysis and implementation results in Section 4.4. We provide timing estimates for particular schemes based on the number of processing units and give a comparative analysis in Section 4.4.1. In Section 4.5, we give a detailed analysis of power consumption of the proposed design and compare it with other architectures. Section 4.6 provides a discussion on the side channel attacks and gives a detailed analysis of dynamic power consumption of the discussed architectures.

Lastly, in Chapter 5, we give a detailed explanation and analysis of the proposed tamper-resilient architectures. In Section 5.1, the lightweight and tamper-resilient error detection scheme built over the extension field is explained in detail, with hardware and software implementation analysis. Section 5.2 gives a detailed analysis of the lightweight and robust error detection scheme built over the base field.

Chapter 2

Background

2.1 Arithmetic in Characteristic Three

In this section, we present hardware architectures for addition, subtraction, multiplication and cubing in $GF(3^m)$. Characteristic three arithmetic is slightly more complicated than characteristic two arithmetic since coefficients can take three values: 0, 1 and 2. Hence, two bits are needed to represent each digit in $GF(3)$. There are two common representations: i) 0, 1, 2 = 00, 01, 10 and ii) 0, 1, 2 = 00, 01 10, 11. The advantage of the latter representation is that ”check if zero operation” is implemented by only checking the most significant bit of the digit since both alternatives for representing digit 0 have 0 in the most significant position. The disadvantage, however, is that negation is performed by subtracting the digit from zero, which can be done by using the addition circuit again in one clock cycle. The negation, on the other hand, in the former representation is performed by just swapping the most and the least significant bits, which is almost free in hardware implementations. Since negation operation is used very often especially in performing $GF(3^{6m})$ multiplication, the former representation is more advantageous in our case. For arithmetic operations, m-bit elements are expressed as 2m-bit arrays as follows

$$A = (a_m^H - 1, a_m^L - 1, ..., a_1^H, a_1^L, a_0^H, a_0^L)$$

2.1.1 Addition and Subtraction

Addition and subtraction is performed component-wise by using the Boolean expression in [15], i.e.

$$\begin{aligned} C_i &= A_i + B_i, \text{for } i = 0, 1, ..., m-1 \text{and} \\ t &= (A_i^L \vee B_i^H) \oplus (A_i^H \vee B_i^L) \\ C_i^H &= (A_i^L \vee B_i^L) \oplus t \\ C_i^L &= (A_i^H \vee B_i^H) \oplus t \end{aligned}$$

where $\vee$ and $\oplus$ stands for logical OR and XOR operations, respectively. In the representation, negation and multiplication of $GF(3)$ elements by 2 are equivalent operations and performed by swapping the most and least significant bits of the digit representing the element. Therefore, subtraction in $GF(3^m)$ is equally efficient as the addition in the same field and thus the same adder block is used for both operations. If subtraction is needed, bits in digits of subtrahend are individually swapped and connected to the adder block. Since this is achieved by only wiring, no additional hardware resource is used.

2.1.2 Cubing

For the Modified Duursma-Lee algorithm, we need cubing operation in $GF(3^{6m})$ and it is possible to build a parallel architecture by using $GF(3^m)$ cubing blocks as explained in the next section. Our aim is to build an optimum cubing circuit in $GF(3^m)$. Cubing is a linear operation in characteristic three and we adopt the technique presented in [6].

For characteristic three, frobenius map is written as follows:

$$A^3 \equiv (\sum_{i=0}^{m-1} a_i x^i)^3 \pmod{p(x)} = \sum_{i=0}^{m-1} a_i x^{3i} \pmod{p(x)}$$

This formula can be represented as follows:

$$\begin{aligned} A^3 &\equiv (\sum_{i=0}^{3(m-1)} a_{i/3} x^i)^3 \pmod{p(x)} \equiv T + U + V \pmod{p(x)} \\ &\equiv ((\sum_{i=0}^{m-1} a_{i/3} x^i) + (\sum_{i=m}^{2m-1} a_{i/3} x^i) + (\sum_{i=2m}^{3(m-1)} a_{i/3} x^i)) \pmod{p(x)} \end{aligned}$$

Here the degrees of the terms U and V are bigger than m and need to be reduced. For $p(x) = x^m + p_t x^t + p_0$ and $t < m/3$, the terms can be represented as follows as also showed in [6]:

$$\begin{aligned} U &= \sum_{i=m}^{2m-1} a_{i/3} x^i \pmod{p(x)} = \sum_{i=m}^{2m-1} a_{i/3} x^{i-m}(-p_t x^t - p_0) \pmod{p(x)} \\ V &= \sum_{i=2m}^{3(m-1)} a_{i/3} x^i \pmod{p(x)} = \sum_{i=2m}^{3(m-1)} a_{i/3} x^{i-2m}(a^{2t} - p_t p_0 a^t + 1) \pmod{p(x)} \end{aligned}$$

Reduction is basically done by additions. For irreducible polynomial $p(x) = x^m + p_t x^t + p_0$, each x^m and x^{2m} are replaced with $(-p_t x^t - p_0)$ and $(a^{2t} - p_t p_0 a^t + 1)^1$,

[1]Note that $x^{2m} = (x^m)^2 = (-p_t x_t - p_0)^2 = a^{2t} - p_t p_0 a^t + 1$ in $GF(3)$.

respectively. However, there remain the terms with degrees equal to or bigger than m after the first reduction step. This problem can be solved by performing reduction one more time. The result of the first reduction can be stored in a register and the second reduction can be done in the next clock cycle. This naturally increases the maximum operating frequency of the block. However since the cubing circuit is not in the critical path, the second reduction step is implemented in the same clock cycle as the first reduction step.

We optimize the reduction for the well known polynomial $p(x) = x^{97} + x^{16} + 2$ and calculate the terms to be added in order to achieve reduction in the same clock cycle. This optimization for a specific polynomial results in a very efficient implementation. We used 111 $GF(3)$ adders to complete the cubing operation. And critical path of the system consists of three serially connected $GF(3)$ adders.

2.1.3 Multiplication

The multiplier architecture presented in this section was implemented by Giray Komurcu.

Multiplication is the most important operation for pairing implementations due to its complexity. Since the modified Duursma Lee algorithm requires $GF(3^{6m})$ multiplications, we need 18 $GF(3^m)$ multipliers in parallel, as explained in the next section. Therefore, designing an efficient multiplier architecture is the key for an efficient accelerator.

Hardware architectures proposed in the literature for $GF(3^m)$ multiplication can be treated in three major classes: parallel, serial and digit multipliers. Firstly, parallel multipliers multiply two $GF(3^m)$ elements in one clock cycle. Although parallel multiplier sustain a high throughput, they consume prohibitively large amount of area and reduce the maximum clock frequency due to very long critical path. Since area and time complexity are very critical parameters for the practical usage of pairings, parallel multipliers are not appropriate on constrained devices.

Secondly, serial multipliers process a single coefficient of the multiplier at each clock cycle. These types of multipliers require m clock cycles for each $GF(3^m)$ multiplication, while their area consumption and critical path delay are relatively small compared to other types of multipliers.

Finally, digit multipliers are very similar to serial multipliers but they process n coefficients of the multiplier at each clock cycle rather than a single coefficient. Consequently, the operation is completed in m/n cycles. The area consumption is more than the serial multipliers and increases with the digit size. Since the area critical path delay also increases with the digit size, the choice of n is important due to area and time concerns.

We prefer to use use serial multipliers in our implementation, which incur increased number of clock cycles, while providing a better solution in terms of area and frequency. Serial multipliers can also be treated in two classes: i) least-significant-element-first (LSE) and ii) most-significant-element-first (MSE). Although there is not much difference between the two types we implement the LSE Multiplier.

As illustrated in Algorithm 1, the reduction is performed in an interleaved fashion.

For interleaved reduction, we subtract $a_m(p_{m-1}x^{m-1} + \ldots + p_1x + p_0x)$ from the partial

Algorithm 1 LSE Multiplier [6]

```
Require: $A = \sum_{i=0}^{m-1} a_i a^i, B = \sum_{i=0}^{m-1} b_i a^i$, where $a_i, b_i \in GF(p)$
Ensure: $C \equiv A \cdot B = \sum_{i=0}^{m-1} c_i a^i$, where $c_i \in GF(p)$
1: $C \leftarrow 0$
2: for $i = 0$ to $m - 1$ do
3:     $C \leftarrow b_i A + C$
4:     $A \leftarrow Aa \pmod{p(a)}$
5: end for
6: return $(C)$
```

result C whenever $a_m \neq 0$ since $x^m = -p_{m-1}x^{m-1} - \ldots - p_1 x - p_0$.

Two LSE multipliers are designed to examine the effect of fixed versus generic polynomials on time and space complexities. In the generic design, modulus polynomial is given as input to the block. The advantage of the generic design is that it can be used with any polynomial in characteristic three. This is an important flexibility for systems that may use more than one polynomials. In case of fixed polynomials, the coefficients of the polynomial can be hard coded into the multiplier unit resulting in reduction of design complexity. For the fixed irreducible polynomial of $x^{97} + x^{16} + 2$, used in many pairing based cryptographic systems in literature, only two $GF(3)$ additions are needed in each iteration of interleaved reduction.

The proposed $GF(3^m)$ LSE multiplier architecture is shown in Figure 2.1.

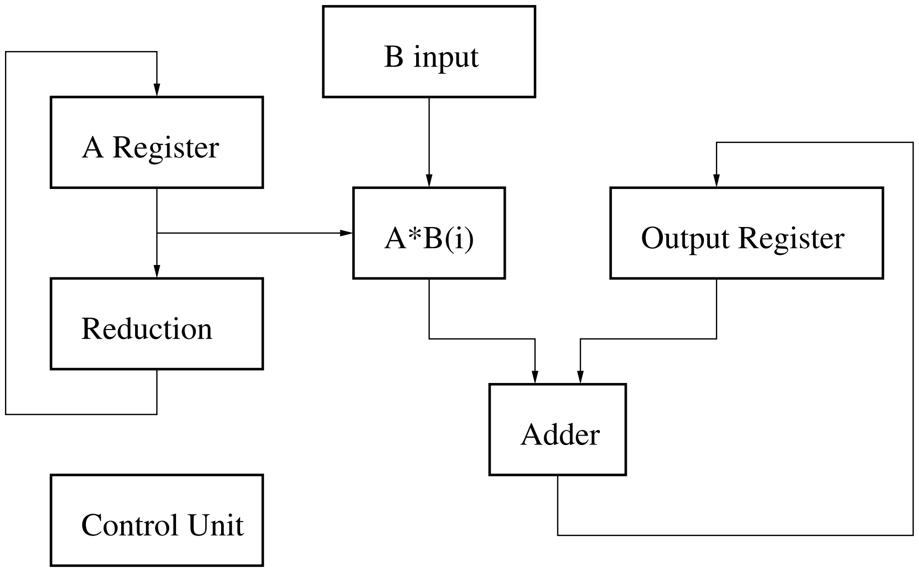


Figure 2.1: LSE multiplier architecture over $GF(3^m)$

2.2 Tate Pairing

The original Tate pairing is defined as follows.

Definition 1 *Let E be an elliptic curve over a finite field $\mathbb{F}_q$. Let l be a positive integer which is a coprime to q. Generally l is a prime and $l|\#E(\mathbb{F}_q)$. Let k be a positive integer such that $l|(q^k-1)$. Finally, let $G = E(\mathbb{F}_q^k)$. Then the Tate Pairing is a mapping:*

$$\langle .,. \rangle : G\,[l] \times G/lG \rightarrow \mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^l \tag{2.1}$$

For further information on Tate pairing, we refer the reader to [12, 10].

2.2.1 Modified Tate Pairing

Duursma and Lee proposed an efficient way of Tate pairing calculations in characteristic three [10]. They introduced the algorithm known as the Duursma-Lee Tate Pairing Algorithm on a family of supersingular hyperelliptic curves. Kwon [23] and Barreto et al. [4] improved the performance of Duursma-Lee Algorithm for Tate Pairing. Their method for computing the Tate Pairing in characteristic three, the Kwon-BGOS algorithm, is shown in Algorithm 1 [16].

Following from [4, 5], they constructed their algorithm on the supersingular elliptic curve $E(F_q) : y^2 = x^3 - x \pm 1$ which is defined over a Galois field $GF(3^m)$. Let $q = 3^m$ where m is generally a prime and $\mathcal{P} = (x_1, y_1)$ and $\mathcal{Q} = (x_2, y_2)$ are points of order l: $\mathcal{P}, \mathcal{Q} \in E_{\pm}[l](GF(3^m))$. Then the Kwon-BGOS modified Tate Pairing on the elliptic curve E is defined as the mapping

$$\langle .,. \rangle : E_{\pm}(GF(3^m))[l] \times E_{\pm}(GF(3^m))[l] \rightarrow GF(3^{6m}) \tag{2.2}$$

which is a function on two points $\mathcal{P}$ and $\mathcal{Q}$ given as

$$\hat{e}(\mathcal{P}, \mathcal{Q}) = e_l(\mathcal{P}, \phi(\mathcal{Q}))^{\epsilon} = \tau \in GF(3^{6m}) \tag{2.3}$$

where $\epsilon = (3^{6m} - 1)/l$ and ϕ is the distortion map defined as $\phi(\mathcal{Q}) = \phi(\rho - x_q, \sigma y_q)$. It should be noted that σ and ρ are the zeros of $\sigma^2 + 1$ and $\rho^3 - \rho \mp 1$, respectively and thus satisfy $\sigma^2 + 1 = \rho^3 - \rho \mp 1 = 0 \in GF(3^{6m})$.

For the purpose of simplicity and to better emphasize the calculations that will be introduced later in this chapter, we utilize the curve $E(\mathbb{F}_q) : y^2 = x^3 - x + 1$.

Algorithm 1 - The Kwon-BGOS algorithm [23]

Input: points $\mathcal{P} = (x_1, y_1)$,

$\mathcal{Q} = (x_2, y_2) \in E_{\pm}[l](GF(3^m))$

Output: $f_{\mathcal{P}}(\phi(\mathcal{Q})) \in F^*_{q^6}/(F^*_{q^3})^l$

```
Step   Operation                          Comments
1:     $f := 1$
2:     $x_2 := x_2^3$                     $GF(3^m)$ arithmetic
3:     $y_2 := y_2^3$                     $GF(3^m)$ arithmetic
4:     $d := \pm m \pmod 3$               $GF(3)$ arithmetic
5:     for $i$ from 1 to $m$
6:       $x_1 := x_1^9$                   $GF(3^m)$ arithmetic
7:       $y_1 := y_1^9$                   $GF(3^m)$ arithmetic
8:       $\mu := x_1 + x_2 + d$           $GF(3^m)$ arithmetic
9:       $\lambda := y_1 y_2 \sigma - \mu^2$   $GF(3^m)$ arithmetic
10:      $g := \lambda - \mu\rho - \rho^2$
11:      $f := f^3$                       $GF(3^{6m})$ arithmetic
12:      $f := f \cdot g$                 $GF(3^{6m})$ arithmetic
13:      $y_2 := -y_2$                    $GF(3^m)$ arithmetic
14:      $d := d \mp 1 \pmod 3$
15:    return $f^{q^3-1}$
```

Figure 2.2: Tate Pairing in Characteristic Three

2.3 Redundant Signed Digit (RSD) Arithmetic

Although carry-free arithmetic decreases the propagation delay in addition operations, its use for modular subtraction operations introduces significant problems. When two's complement representation is used for subtraction, the carry overflow must be ignored. If there is no carry overflow, the result is negative. Since there can be hidden carry overflow with carry-free representation, it is hard to be sure that the result is positive or negative. It requires additional operations and additional hardware, which increases both latency and area. The RSD representation was introduced by Avizienis [2] in an effort to overcome this difficulty.

Arithmetic in the RSD representation is quite similar to carry-free arithmetic. An integer is still represented by two positive numbers, however the non-redundant form of the representation is the difference between these two numbers, not the sum. If the number X is represented by x^p and x^n then $X = x^p - x^n$.

One advantage of using the RSD representation is that it eliminates the need for two's complement representation to handle negative numbers. It is thus much easier to do both addition and subtraction operations without worrying about the carry and borrow chain. Furthermore, the subtraction operation does not require taking two's complement of the subtrahend. It is a more natural representation if both addition and subtraction operations need to be supported. This is indeed the case in the Montgomery multiplication and inversion algorithms. Also, comparison of two integers is much easier with RSD representation. After subtracting one integer from the other one, which is a simple addition operation, a conventional comparator can be utilized.

2.3.1 Number Representations

As mentioned earlier, the integer X is represented by two integers, x^p and x^n, and $X = x^p - x^n$. For the RSD representation, we reserve the notation (x^p, x^n) to represent the number X. The RSD representation for extension fields is described as follows:

1. Prime field $GF(p)$: Elements of the prime field $GF(p)$ may be represented as integers in binary form. In the binary RSD representation its digits can have three different values: 1, 0 and -1. These three digit values are represented as

$$\begin{aligned} 1 &\rightarrow (1,0) \\ 0 &\rightarrow (0,0) \\ -1 &\rightarrow (0,1) \end{aligned}$$

2. Binary extension field $GF(2^n)$: Elements of the field $GF(2^n)$ may be considered as polynomials with coefficients from $GF(2)$. This allows one to represent $GF(2^n)$ elements by simply ordering its coefficients into a binary string. Since there is no carry chain in $GF(2)$ arithmetic, a digit can have the values 1 or 0. These values are represented as:

$$\begin{aligned} 1 &\rightarrow (1,0) \\ 0 &\rightarrow (0,0) \end{aligned}$$

3. Ternary extension field $GF(3^m)$: Elements of the extension field $GF(3^m)$, may be considered as polynomials over $GF(3)$. The coefficients can take the values -2, -1, 0, 1, and 2. However, since there is no carry propagation in $GF(3^m)$ polynomial arithmetic, the digit values -2 and 2 are congruent to 1 and -1, respectively. The RSD representations for possible coefficient values are given as

$$\begin{aligned} 2 &\rightarrow (0,1) \\ 1 &\rightarrow (1,0) \\ 0 &\rightarrow (0,0) \\ -1 &\rightarrow (0,1) \\ -2 &\rightarrow (1,0) \end{aligned}$$

2.4 Robust Codes

A class of non-linear systematic error detecting codes, so-called “robust codes”, were proposed by Karpovsky and Taubin [18]. They can achieve optimality according to the minimax criterion, that is, minimizing over all (n, k) codes the maxima of the fraction of undetectable errors $Q(e)$ for $e \neq 0$. The following definition from [18] rigorously defines a particular class of non-binary codes.

Definition 2 *Let V be a linear p-ary (n, k) code ($p \geq 3$ is a prime) with $n < 2k$ and check matrix $H = [P|I]$ with rank$(P) = r = n - k$. Then $C_V = \{(x, w)|x \in GF(p^k), w = (Px)^2 \in GF(p^r)\}$.*

To quantify the performance of C_V we need a metric. The error masking probability for a given non-zero error $e = (e_x, e_w)$ may be quantified as

$$Q(e) = \frac{|\{x|(x + e_x, w + e_w) \in C_V\}|}{|C_V|} \ .$$

Note that we call the code C_V robust, if it minimizes maxima of $Q(e)$ over all possible non-zero errors. Reference [18] provides the following theorem which quantifies the error detection performance of the nonlinear code C_V.

Theorem 1 *For C_V the set $E = \{e|Q(e) = 1\}$ of undetected errors is a $(k-r)$-dimensional subspace of V, $p^k - p^{k-r}$ errors are detected with probability 1 and remaining $p^n - p^k$ errors are detected with probability $1 - p^{-r}$.*

These codes achieve total robustness for the case $r = k$, when the subspace of undetectable errors collapses to the zero codeword and all non-zero error patterns can be detected with a probability of either $1 - p^{-r}$ or 1. The reduced overhead for the case $r < k$ is obtained at the expense of the loss of total robustness. Nonetheless, some important properties of robust codes are retained: Contrary to linear encoding schemes, the probability of missing an error is largely data-dependent. This has one very important consequence: an active adversary trying to induce an undetected error in the data would need to

- know the value of the data a priori in order to compute an undetectable error pattern
- induce a fault with sufficient spatial and temporal accuracy such as to successfully re-create the matching error in the device.

In a linear scheme, any error pattern that is a codeword itself will lead to a successful compromise, regardless of the value of the targeted data vector. Although there also exist some error patterns in the robust scheme which will escape detection, their number is significantly smaller than in linear schemes. For robust coding schemes, the number of such (undetected) error vectors can be made exponentially small by linearly increasing the number of redundancy bits. Hence, an attacker will have virtually no chance of inserting an undetectable error vector, unless they read the target data vector first, then compute an appropriate error pattern, and precisely insert the computed pattern with high spatial and temporal resolution.

Chapter 3

Our Unified Arithmetic Unit

Parts of this chapter were presented in [26].

We first build a unified arithmetic core for the basic arithmetic operations (i.e., addition, subtraction and comparison). The core is unified so that it can perform the arithmetic operations of three extension fields: $GF(p)$, $GF(2^n)$ and $GF(3^m)$. Since the elements of the three different fields are represented using a very similar data structure, the algorithms for basic arithmetic operations in these fields are structurally identical. We use this fact to our advantage to realize a unified arithmetic core.

3.1 The Architecture

The conventional 1-bit full adder assumes positive weights for all of its three binary inputs and two outputs. However, full adders can be generalized to have both positive and negative-weight inputs and outputs. This allows us to construct an adder design with both inputs and outputs in RSD form, since we can have negative weight numbers as inputs. In our core design, we used two forms of the generalized full adders as shown in Figure 3.1: one negative weight input (GFA-1) and two negative-weight inputs (GFA-2). Note that GFA-0 is identical to a common full adder design.

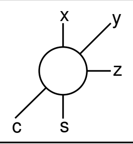
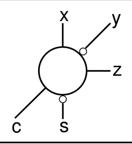
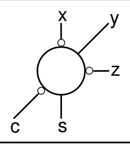
Logic symbol			
Type	GFA−0	GFA−1	GFA−2
Function	x+y+z = 2c+s	x−y+z=2c−s	−x+y−z=−2c+s

Figure 3.1: Generalized full adders

The logic behaviors of a common full adder and two generalized full adders are shown in Figure 3.2. As visible from the truth table, $GFA-1$ and $GFA-2$ have the same logical characteristics. The only difference is the order of the inputs and outputs. The same hardware is used for both types of generalized full adders. However, it should be noted that the decoding of the outputs are different. For $GFA-1$, the result is decoded as $2c-s$. For $GFA-2$, the result is decoded as $-2c+s$.

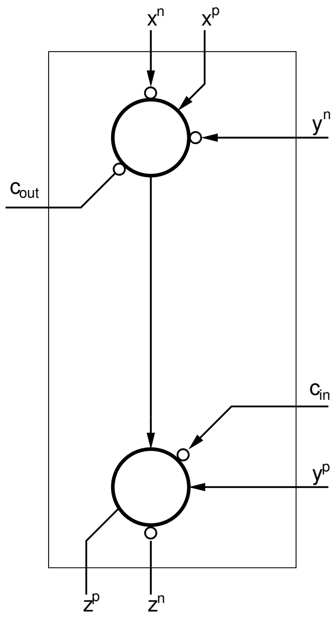
A single digit unified adder unit is constructed using two of the generalized full adders as shown in Figure 3.3(b). The unified adder unit has two digits in RSD representation as inputs and one digit in RSD representation as output. The unified digit adder unit also has carry input and output, which are only used for arithmetic in $GF(p)$. In total, the unit has 5 bits input and 3 bits output.

			GFA–0		GFA–1		GFA–2	
x	y	z	s	c	s	c	s	c
0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	1	1
0	1	0	1	0	1	0	1	0
0	1	1	0	1	0	0	0	0
1	0	0	1	0	1	1	1	1
1	0	1	0	1	0	1	0	1
1	1	0	0	1	0	0	0	0
1	1	1	1	1	1	1	1	1

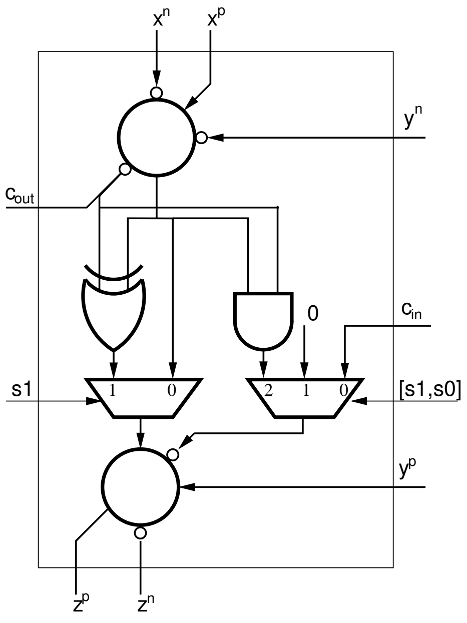
Figure 3.2: Logic tables of the three generalized full adders

We start by designing the hardware for the prime fields ($GF(p)$) first. Two generalized full adders connected in the configuration shown in Figure 3.3(a) is sufficient to handle the digit arithmetic of $GF(p)$ elements. To make the adder architecture work for $GF(2^n)$ arithmetic, we inhibit the carry chain. Also, since the digits can only have the values $(0, 0)$ and $(1, 0)$, the negative weight inputs of the adder are set to logic 0.

Modifying the adder design to make it also work for $GF(3^m)$ is more difficult since the hardware works for base two and we need to support base three. The carry-free structure of the $GF(3^m)$ arithmetic operations makes our task easier. When carrying out arithmetic operations in $GF(3^m)$, the outputs of the adders have to be decoded. Since the generalized full adder works in binary form, the output is also in binary. We need to convert this output to base 3 before entering the data into the second generalized full adder. An XOR gate and an AND gate is sufficient for this conversion as shown in Figure 3.3(b). There is also need for multiplexers, where the select inputs of the multiplexers determine the field in which



(a) Single RSD adder unit.



(b) Unified RSD adder unit.

Figure 3.3: RSD adder unit with both inputs and outputs in RSD form

the adder is operating. The carry bits are only used when the circuit functions in $GF(p)$ mode. In Figure 3.3(b), $s1$ and $s0$ are the select inputs of the multiplexers. The modes of the hardware are:

$$[s1, s0] = 0, 0 \quad \rightarrow \quad GF(p)$$

$$[s1, s0] = 0, 1 \quad \rightarrow \quad GF(2^n)$$

$$[s1, s0] = 1, 0 \quad \rightarrow \quad GF(3^m)$$

Now we need to cascade n single digit RSD units in order to build an n-digit RSD adder. Figure 3.4 shows the backbone of the structure. There are n 1-digit RSD adders and one GFA-1 adder to handle the last carry bit, which is omitted in $GF(2^n)$ and $GF(3^m)$.

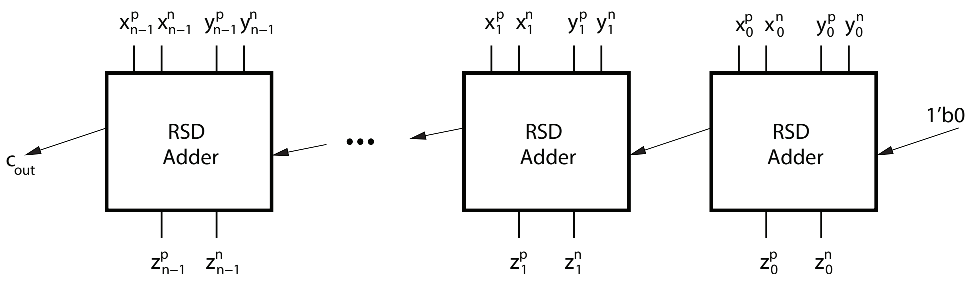

x^p_{n-1} x^n_{n-1} y^p_{n-1} y^n_{n-1}
x^p_1 x^n_1 y^p_1 y^n_1
x^p_0 x^n_0 y^p_0 y^n_0
RSD Adder
RSD Adder
RSD Adder
c_{out}
1'b0
z^p_{n-1} z^n_{n-1}
z^p_1 z^n_1
z^p_0 z^n_0

Figure 3.4: RSD adder

In figure 3.4, it can be seen that the carry bits of each RSD adder unit is propagated to the consecutive adder unit. It should be noted here that since the carry output of each adder unit is an output from the first GFA, this carry bit propagates **only** to the second GFA of the consecutive adder unit, as can be seen in Figures 3.3(b) and 3.3(b).

3.2 Arithmetic Operations

3.2.1 Addition

The addition operation is implemented as shown in Figure 3.4. The negative and positive parts of the numbers are entered accordingly and the select inputs of the multiplexers are set for desired field operations. There are also two control inputs to the adder for selecting the field, sel_2 and sel_3, which are not shown in Figure 3.4. These inputs are decoded accordingly and they determine the select inputs of the multiplexers. It should be noted that in Figure 3.4, carry propagation occurs only between neighboring cells.

3.2.2 Subtraction

Subtraction operation is identical to the addition operation. The only difference is that the positive and the negative parts of the numbers in RSD form are swapped before the operation. Swapping the positive and negative parts negates the number:

$$\begin{aligned} X &= (x^p, x^n) = x^p - x^n \\ Y &= (y^p, y^n) = y^p - y^n \\ X - Y &= (x^p, x^n) - (y^p, y^n) = (x^p, x^n) + (y^n, y^p) \end{aligned}$$

3.2.3 Comparison

To compare two numbers given in the RSD representation, first one must be subtracted from the second one. After subtraction, the positive and negative components of the result are compared. This can be realized using a conventional comparator design. If the positive part is larger, the first number is greater than the second one. If the negative part is larger, the second number is greater than the first one. If both parts are equal, then the numbers being compared are equal.

The comparison operation has 2 components: RSD adder and comparator. There is already RSD adders in the design and one of them could be utilized for comparison. Also, a single RSD adder can be instantiated for comparison reasons only, without a significant area overhead.

Furthermore, a conventional comparator is used for comparing the positive and negative parts of the resultant of the subtraction operation. We designed this comparator using Verilog and synthesized with Synopsys Design Compiler with 0.13μm ASIC library. For synthesis we used two target frequencies: 500 MHz and the maximum frequency for the circuit. For the maximum frequency, we setup the synopsys tools to push the limits of the critical path and optimize as much as possible for timing. The results are shown in Table 3.1.

word	500 MHz		Max. Freq.	
length	Area	CPD(ns)	Area	CPD(ns)
8	47	0.72	70	0.39
16	95	0.80	161	0.42
32	191	1.24	391	0.49
64	451	1.35	756	0.55

Table 3.1: Implementation results of comparator design with different word sizes.

We also implemented a single RSD adder to utilize for comparison. Synthesis results showed that the minimum CPD of a single RSD adder is 0.66 ns. This shows that the critical path of an adder and a comparator connected back to back will not be more than the overall circuit, even for the 64-bit case. Thus, the word comparison operation can be performed in a single clock cycle.

It should be noted that most of the field arithmetic operations require the equality comparison of 2 numbers. Hence, a much simpler comparator could be utilized for comparison operations.

Chapter 4

Unified Montgomery Multiplier Implementation

In this chapter we explain the multiplier design which implements Algorithms 2 and 3 in a single architecture. We do not go into the detail of the global control logic path since its function can be inferred easily from the algorithms.

4.1 Montgomery Multiplication

4.1.1 General Algorithm

The Montgomery multiplication algorithm [24] is an efficient method for performing modular multiplication with an odd modulus. The algorithm replaces costly division operations with simple shifts, which are particularly suitable for implementations on general-purpose computers.

Given two integers A and B, and the odd modulus M, the Montgomery multiplication algorithm computes $Z = \text{MonMul}(A, B) = A \cdot B \cdot R^{-1} \bmod M$, given $A, B < M$ and R such that gcd$(R, M) = 1$. Even though the algorithm works for any R which is relatively prime to M, it is more useful when $R = 2^n$ where $n = \lceil \log_2(M) \rceil$. Since R is chosen to be a power of 2, the Montgomery algorithm performs divisions by a power of 2, which is basically shift operations in digital computers. The Montgomery multiplication algorithm for binary extension fields $GF(2^n)$ are first introduced in [21]. We describe the Montgomery multiplication algorithm for ternary extension fields $GF(3^m)$ in the subsequent sections.

The proposed adder design is used to build a Montgomery multiplier architecture. Since we want our hardware to support arithmetic in three different fields, we identify similarities between the arithmetic algorithms and integrate them together into a single hardware implementation.

4.1.2 Montgomery Multiplication Algorithm for $GF(p)$ and $GF(2^n)$

The use of a fixed precision word alleviates the broadcast problem in the circuit implementation. Furthermore, a word-oriented algorithm allows design of a scalable unit. For a modulus of n-bit precision, and a word size of w-bits, $e = \lceil (n + 1)/w \rceil$ words are required for storing field elements. Note that an extra bit is used for the variables holding the partial sum in the Montgomery algorithm for $GF(p)$, since the partial sums can reach $(n + 1)$-bit precision. The algorithm we used ([37]) scans the multiplicand operand B word-by-word, and the multiplier operand A bit-by-bit. The vectors used in multiplication operations are expressed as

$$\begin{aligned} B &= (B^{(e-1)}, ..., B^{(1)}, B^{(0)}), \\ A &= (a_{n-1}, ..., a_1, a_0), \\ p &= (p^{(e-1)}, ..., p^{(1)}, p^{(0)}), \end{aligned}$$

where the words are marked with superscripts and the bits are marked with subscripts. For example, the ith bit of the kth word of B is represented as $B_i^{(k)}$. A particular range of bits in a vector B from position i to j where $j > i$ is represented as $B_{j..i}$. $(x|y)$ represents the concatenation of two bit sequences. Finally, 0^n stands for an all-zero vector of n bits. The algorithm is shown in Algorithm 2.

We use the RSD form for every vector in the multiplication algorithm, so each bit expressed in this algorithm is represented by two bits in the hardware, positive and negative parts of the numbers. As an example: $T_0^0 = (T_{0,p}^0, T_{0,n}^0)$.

The $GF(2^n)$ version of the algorithm is structurally identical with only a few minor differences. First of all, the operands and temporary variable T are represented as polynomials in the algorithm. The modulus is also a polynomial, $P(x)$. As a result of the polynomial arithmetic, the addition symbols, i.e. '+' represent carry-free addition or bit-wise XOR operation. Since polynomial addition is a carry-free operation, *Carry* is ignored in Steps $3, 5, 7$ and 9. Also, Step 13 is not operated.

4.1.3 Radix-3 Montgomery Multiplication Algorithm for $GF(3^m)$

Montgomery multiplication algorithms for $GF(p)$ and $GF(2^n)$ are similar to each other because they are both implemented in radix-2. Since the Montgomery multiplication algorithm for $GF(3^m)$ is implemented in radix-3, the algorithm needs to be modified. We already explained the differences for the addition part in RSD representation and we showed that radix-2 and radix-3 representations can be both implemented on a single hardware.

We will use polynomial basis representation for $GF(3^m)$. For a modulus size of m and a word size of w, $e = \lceil (m+1)/w \rceil$ words are required. Since there is no carry computation in $GF(3^m)$ arithmetic, there will be no need for any extra digits used other than those used for the variable polynomials. Every coefficient of the operands and the modulus is represented by two bits in the hardware, one for the positive part and one for the negative part, since the coefficients are in RSD representation. The algorithm scans the words of operand $B(x)$, and the coefficients of operand $A(x)$. In radix-3 representation, the polynomials used in multiplication operation are expressed as

$$\begin{aligned}
B(x) &= (b^{(e-1)} \cdot x^{(e-1)\cdot w} + ... + b^{(1)} \cdot x^{w} + b^{(0)}), \\
A(x) &= (a_{n-1} \cdot x^{n-1} + ... + a_1 \cdot x + a_0), \\
p(x) &= (p^{(e-1)} \cdot x^{(e-1)\cdot w} + ... + p^{(1)} \cdot x^{w} + p^{(0)}),
\end{aligned}$$

where the words are marked with superscripts and the coefficients are marked with subscripts. For example, the i-th coefficient of the k-th word of $B(x)$ is represented as $B_i^{(k)}$. The algorithm is shown in Algorithm 3.

4.2 Pipeline Organization

The presented Montgomery multiplication algorithms have the same loop structure: outer and inner loops with the variables i and j, respectively. Each processor unit (PU)[1] is responsible for one step of the outer loop with the variable i. Each PU receives the a_i digit as input. Also, every PU receives $B^{(j)}$, $p^{(j)}$ and $T^{(j)}$ as inputs, according to the inner loop variable j. The pipelined organization is shown in Figure 4.1.

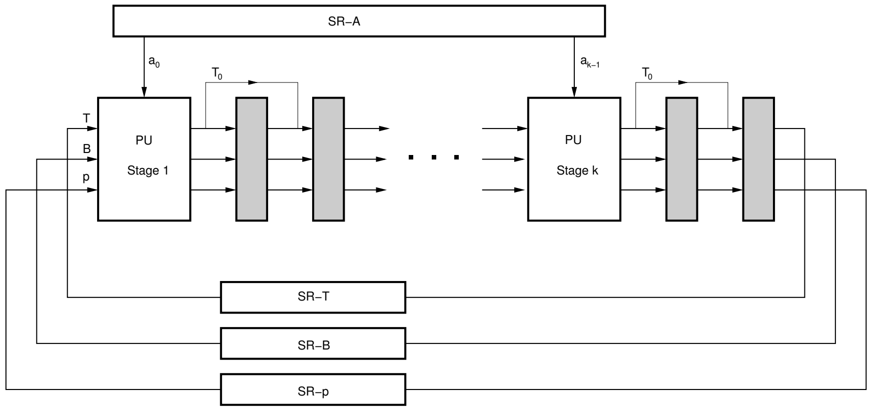


Figure 4.1: Pipeline organization for the Montgomery Multiplier

An important aspect of the pipeline is the organization of the registers. The digits a_i of the multiplier A are given serially to the PUs, and are used only for one iteration of the outer loop. So they can be discarded immediately after use. Therefore, a simple shift register with a load input will be sufficient. Also, rather than storing the multiplier A in a register, we can have a serial input for every digit and we store only the necessary a_i digit inside a register, only when it is needed. This will reduce the area and power consumption

[1]We will define the internal structure of the PU in the following section

of the architecture. The registers for the modulus p and multiplicand B can also be shift registers.

The multiplication starts with the first PU by processing the first iteration of the outer loop of the algorithm. As can be seen from Algorithm 2, the data required for the second iteration will be ready after 2 clock cycles. Therefore, the second PU has to be delayed from the first PU by 2 clock cycles. This is realized by using two stages of registers in between. Also, these registers are handling the shift operations for the partial sum (Step 8 of Algorithm 2) as shown in Figure 4.1.

When the first PU finishes the operations of an iteration step of the outer loop, it starts working on the next available iteration loop, and the second PU will be done in 2 clock cycles and start working on the next available iteration. The same computation pattern is repeated for the entire pipeline organization.

If there are sufficiently many PUs, the first PU will be done with the first iteration of the loop when the last PU operates on the last iteration of the same loop. There will be no pipeline stall and no need for intermediate shift registers to hold the data. The pipeline can continue working without stalling. This condition is satisfied if the number of PUs is at least half of the number of words of the operand. However, if there are not sufficiently many PUs, which means that a pipeline stall occurs, the modulus and multiplicand words generated at the last stage of the pipeline have to be stored in registers.

The shift registers $SR - T$, $SR - p$ and $SR - B$ to hold these values when there is pipeline stall. The length of these shift registers is of crucial importance and is determined

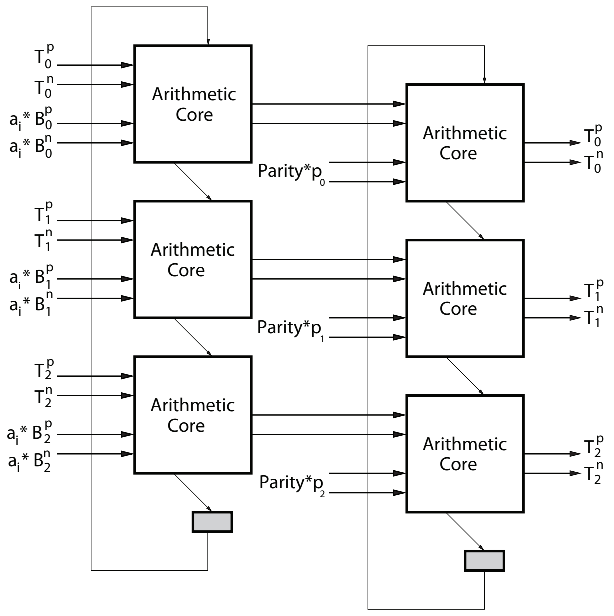


Figure 4.2: Processing Unit (PU) with w = 3.

by the number of pipeline stages k and the number of words e in the modulus. The width of the shift registers is equal to w, the word size. The length of these registers can be given as

$$L = \begin{cases} e - 2 \cdot (k-1) & : \text{ if } e \geq 2k \\ 0 & : \text{ otherwise.} \end{cases}$$

4.3 Processing Unit

The processing unit consists of two layers of adder blocks or unified arithmetic cores (cf. Section 3). The arithmetic core is capable of performing addition and subtraction operations in the fields $GF(p)$, $GF(2^n)$ and $GF(3^m)$. The block diagram of a processing unit with word size w=3 is shown in Figure 4.2.

As can be seen in the figure, a PU is responsible for performing the operation

$$a_i \cdot B^{(j)} + T^{(j)} \pm p^{(j)} \ .$$

This step is common for all the three fields, so this part of the PU is a very simple combination of the unified arithmetic cores. The inputs to these adders come from decoders designed to handle arithmetic in three different fields.

We need a simple logic for multiplying a single digit a_i of the multiplier A with a word $B^{(j)}$ of the multiplicand B to realized the first part $a_i \cdot B^{(j)}$ of the operation. Since a_i can only have the values $(0,0)$, $(1,0)$ or $(0,1)$, the result of $a_i \cdot B^{(j)}$ can be 0, $B^{(j)}$ or $-1 \cdot B^{(j)}$ respectively. Negating an integer is realized by simply swapping the positive and negative bits of its digits. A simple special encoder would be sufficient for this. We need another logic circuit to determine the parity in each iteration of the outer loop. We check the right-most digit of the modulus, i.e. $p_0^{(0)}$ and the right-most digit of the operation $T^{(0)} = a_0 \cdot B^{(0)} + T^{(0)}$, $T_0^{(0)}$ and determine the parity:

$$Parity = \begin{cases} (0,0) & : \text{ if } \ T_0^{(0)} = (0,0) \\ (0,1) & : \text{ if } \ p_0^{(0)} = T_0^{(0)} \\ (1,0) & : \text{ otherwise} \end{cases}$$

This is very similar to the encoder logic we used earlier. One difference is that since the parity is computed only once for every iteration step, it needs to be stored in a register after being computed by the PU.

4.4 Complexity Analysis

As mentioned earlier, if the number of PUs is at least half of the number of words in the operand, the pipeline will not stall and every PU will continuously operate. For multiplication, the total computation time, latency (clock cycles), is given as

$$Latency = \begin{cases} 2(m-1)+e & \text{if } \ e \geq 2k \\ (\lceil \frac{m}{k} \rceil)e + 2((m-1) \bmod k) & \text{otherwise} \end{cases} \tag{4.1}$$

The graphs given in Figure 4.3 illustrate how the latency of Montgomery Multiplication changes for various operand lengths and for a variable number of PUs.

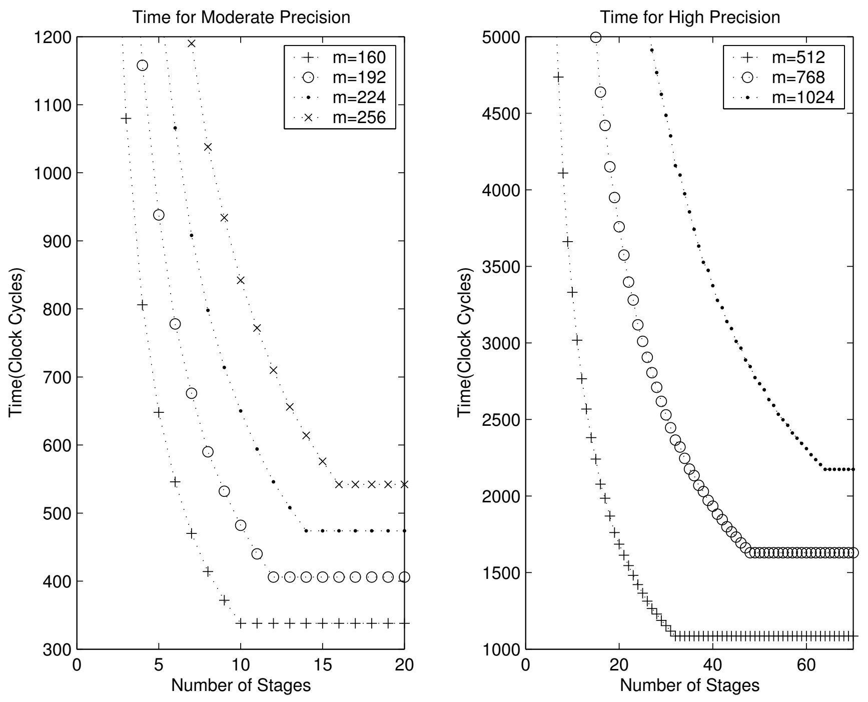


Figure 4.3: Computation time of Montgomery Multiplication for various number of PUs and operand lengths.

Table 4.1 shows the estimates for the number of clock cycles required for realizing ECC scalar point multiplication, RSA exponentiation, and Tate pairing computations with the modified Duursma-Lee algorithm. We pick a word size of 8-digits. For the implementation of ECC with 160-bits we assume mixed coordinates and the NAF representation are used to realize the scalar point multiplication operation. For point doubling we use Jacobian coordinates and for point addition we use affine+Jacobian coordinates. For RSA we assume a full 1024-bit exponent and use the square multiply algorithm. The Tate pairing computation is realized using the modified Duursma-Lee algorithm [19] over the field $GF(3^{6\times 97})$ (The original Duursma-Lee algorithm was proposed in [11]).

Note that the chosen lengths provide similar levels of security. We are not getting into the details of the clock cycle computations for the ECC and RSA cases since the computations are trivial. For the Tate pairing case we note that the modified Duursma-Lee algorithm [19] iterates 97 times and works by performing the operations in the field $GF(3^{97})$. In each iteration 20 multiplications and 10 cubing operations are carried out in the field $GF(3^{97})$. Each cube computation may be realized via two multiplications bringing the total number of multiplications to 40 per iteration of the main loop of the modified Duursma-Lee algorithm. Including the additional 4 multiplications performed in the initialization of the algorithm the total number of multiplications are found as $40 \cdot 97 + 4 = 3884$. In the 4 PU case the latency of one multiplication is found using Equation 4.1 as 312 clock cycles. Hence, the total paring computation requires $3884 \cdot 312 = 1211808$ cycles. For the 8 PU case the latency of one multiplication operation is found as 205 clock cycles leading to a total number of 796220 clock cycles.

Number of	**160-bit ECC**	**1024-bit RSA**	**Tate Pairing**
			$GF(3^{97})$
PUs	**(clock cycles)**	**(clock cycles)**	**(clock cycles)**
4	1507728	50340864	1211808
8	772524	25187328	796220
16	630708	12628992	796220
32	630708	6386688	796220

Table 4.1: The execution times for ECC scalar multiplication, RSA Exponentiation and Modified Duursma-Lee algorithms

Number of	**160-bit ECC**	**1024-bit RSA**	**Tate Pairing**
			$GF(3^{97})$
PUs	**(msec)**	**(msec)**	**(msec)**
4	3.015	100.681	2.424
8	1.545	50.374	1.592
16	1.261	25.258	1.592
32	1.261	12.773	1.592

Table 4.2: Execution times at frequency $f = 500$ MHz (Section 4.4.1)

4.4.1 Results and Comparison

In this section, we provide implementation results of the proposed unified architecture to demonstrate its advantage over classical architectures. We also include the implementation results of unified Montgomery multiplier circuit that operates in three finite fields. In addition, we present a qualitative comparison of the proposed architecture with previously defined architectures.

PU Architecture

The presented architecture was developed into Verilog modules and synthesized using the Synopsys Design Compiler tool. In the synthesis we used the TSMC 0.13 μm ASIC library and assumed a word size of 8 bits. The maximum operating frequency of the design was found as 800 MHz. However, the synthesize tool will try to optimize the circuit for timing if we set the target frequency at 800 MHz. Thus, for the rest of this section, we assume a target frequency of 500 MHz for synthesis results. The timing results at 500 MHz for three prominent public key operations are given in Table 4.2. We note that if the pipeline does not stall, as the number of PUs increases the register space will increase. Otherwise, the register space will stay constant with increasing number of PUs.

For proof of concept, we built and synthesized different PUs working on different fields. First category of implementations are those working on a single field only. The implementations, denoted as A_1, A_2, and A_3, are those working in fields $GF(p)$-only, $GF(2^n)$-only, and $GF(3^m)$-only, respectively. In the second category, there are two unified architectures. The implementation, denoted as A_4, is a unified architecture working in both fields $GF(p)$ and $GF(2^n)$. And finally, the implementation A_5 is the unified architecture working in all three fields, namely $GF(p)$, $GF(2^n)$, and $GF(3^m)$. All five architectures are implemented for three different word sizes, 8, 16, and 32 and the implementation results of these architectures are summarized in Table 4.3.

From Table 4.3, the cost of unified architectures compared to $GF(p)$-only implementation can be captured as overhead both in the area and critical path delay (CPD). However,

word length	A_1		A_2		A_3		A_4		A_5	
	Area	CPD (ns)	Area	CPD (ns)	Area	CPD (ns)	Area	CPD (ns)	Area	CPD (ns)
8	516	1.91	91	0.77	656	1.92	576	1.87	795	1.91
16	963	1.90	168	0.79	1257	1.92	1034	1.90	1556	1.92
32	1980	1.89	329	0.84	2534	1.92	2132	1.90	3013	1.92

Table 4.3: Implementation results of a PU with different word sizes.

the figures in Table 4.3 hardly give an idea about the advantage of the unified architectures. Apparently, the advantage of the unified architectures are saving in the area without too much adverse effect on the critical path delay. In order to measure the advantage of the unified architecture, we used (Area $\times$ CPD) as the metric. We first investigated the first unified architecture A_4 that has a single datapath for $GF(p)$ and $GF(2^n)$ and compared it against the implementation results of a hypothetical architecture, denoted as $A_1 + A_2$, that has two separate datapaths for $GF(p)$ and $GF(2^n)$. For the hypothetical architecture A_1+A_2, the area is the sum of areas of A_1 and A_2 architectures while the critical path delay is the maximum CPD of these two architectures. The implementation results are summarized in Table 4.4. The improvement of the architecture is found to be about 7%-8.5% in terms of the Area $\times$ CPD metric.

Similarly, we also investigated the advantage of the unified architecture, A_5 over a hypothetical architecture, $A_1+A_2+A_3$, that has three separate datapaths for the fields $GF(p)$, $GF(2^n)$, and $GF(3^m)$. The results, summarized in Table 4.5, show that the advantage of using the unified architecture, A_5 is at least 34.83% in terms of the metric (Area $\times$ CPD).

word length	Area				CPD			
	A_1	A_2	$A_1 + A_2$	A_4	A_1	A_2	$A_1 + A_2$	A_4
8	516	91	607	576	1.91	0.77	1.91	1.87
16	963	168	1131	1034	1.90	0.79	1.90	1.90
32	1980	329	2309	2132	1.89	0.84	1.89	1.90

word length	Area $\times$ CPD		improvement %
	$A_1 + A_2$	A_4	
8	1159	1077	7.07
16	2149	1965	8.56
32	4364	4051	7.17

Table 4.4: Advantage of the unified architecture A_4, for $GF(p)$ and $GF(2^n)$

The improvement figures in Table 4.5 clearly demonstrate that the unified architecture A_5 provides far superior performance compared to the classical unified architectures working for only the fields $GF(p)$ and $GF(2^n)$.

word length	Area			CPD		
	A_3	$A_1 + A_2 + A_3$	A_5	A_3	$A_1 + A_2 + A_3$	A_5
8	656	1263	795	1.92	1.92	1.91
16	1257	2388	1556	1.92	1.92	1.92
32	2534	4843	3013	1.92	1.92	1.92

word length	Area $\times$ CPD		improvement %
	$A_1 + A_2 + A_3$	A_5	
8	2425	1518	37.40
16	4585	2988	34.83
32	9299	5785	37.78

Table 4.5: Advantage of unified architecture A_5, for $GF(p)$, $GF(2^n)$, and $GF(3^m)$.

In order to see more clearly what one can gain with the new unified architecture A_5 over the classical one, A_4, we also compared the two unified architectures in terms of the

Area $\times$ CPD metric. The results summarized in Table 4.6 highlights the advantage of the new unified architecture over the classical one, which is at least 32%.

word	Area		CPD		Area $\times$ CPD		improvement
length	$A_4 + A_3$	A_5	$A_4 + A_3$	A_5	$A_4 + A_3$	A_5	$\%$
8	1232	795	1.92	1.91	2365	1518	35.81
16	2291	1556	1.92	1.92	4399	2988	32.07
32	4666	3013	1.92	1.92	8959	5785	35.43

Table 4.6: Advantage of the new unified architecture A_5 over the classical unified architecture A_4

From Tables 4.3, 4.4, 4.5 and 4.6, we can conclude the scalability property of the unified architecture and the pipelining organizations. The correlation between the word size of a single PU and area numbers is linear. Also, increasing the word size does not alter the critical path delay of the entire circuit. This scalability property is also true for the number of PUs used in the architecture. Tables 4.1 and 4.2 shows that the number of PUs is linear with the latency numbers before the saturation of the number of PUs.

Montgomery Multiplier Architecture

The Montgomery multiplier architecture presented in Section 4 was developed into Verilog modules and synthesized using the Synopsys Design Compiler. In the synthesis we used the TSMC 0.13 μm ASIC library and assumed a word size of 8 bits. The maximum operating frequency of the multiplier architecture was found as 800 MHz. This shows that the PU constitutes the critical path of the entire design. The synthesis results showed that the area of the multiplier for 4 PUs and 8 PUs was 11,512 and 15,361 two-input NAND equivalent gates, respectively. We note that as the number of PUs increases, the register space will increase if the pipeline does not stall. Otherwise, the register space will stay constant with increasing number of PUs.

Similarly, we also investigated the advantage of the unified Montgomery multiplier architecture over a hypothetical architecture that has three separate datapaths for the fields $GF(p)$, $GF(2^n)$, and $GF(3^m)$. The results, summarized in Table 4.7, show that the advantage of using the unified architecture is at least about 25% in terms of the metric (Area $\times$ CPD). The improvement figures in Table 4.7 clearly demonstrate that the unified multiplier architecture provides far superior performance compared to the classical unified architectures working for only the fields $GF(p)$ and $GF(2^n)$.

# of PUs	Area		CPD		Area $\times$ CPD		improvement $\%$
	Separate Paths	Unified	Separate Paths	Unified	Separate Paths	Unified	
4	10644	8372	2	1.91	21288	15991	24.88
8	15672	12128	2	1.91	31344	23164	26.10

Table 4.7: Synthesis results for Montgomery multiplier architectures, with unified and separate datapaths.

For our architecture, the final results are in the RSD form. After the field operations are completed, the results need to be converted back to the more conventional form before being sent to the adversary. For example, if we are using our multiplier in a Diffie-Hellman protocol, we need to perform an exponentiation operation first. During the exponentiation operation, the intermediate results will stay in the RSD form. After completing the exponentiation operation, the final result has to be converted back to the desired form, depending on the protocol. This conversion can be performed serially utilizing an 8-bit ripple carry adder. Since this is done only once, the latency overhead it produces is negligible, we could even use a bit-serial adder. However, we built an 8-bit ripple carry adder using Verilog and synthesized it with Synopsys Design Compiler, with $0.13\mu m$ library with a target frequency of 500 MHz. Synthesis results showed that the critical path of this adder is 1.34 ns, which is in the range of our multiplier circuit. The area of this adder is 66 gates equivalent. Thus, a word-serial addition operation can be performed without a significant area or latency overhead.

Comparison with Previous Unified Architectures

In this section, we compare the new architecture against the previously proposed unified architectures in [1, 17, 33, 34, 35, 38, 43] to put it in a perspective in relation to other unified architectures. The architecture in [34] is the first and perhaps the most basic unified architecture, whose simplified processing unit (PU) for three bits is shown in Figure 4.4. It basically consists of two layers of dual-field adders (that add with or without carry) and assumes that all inputs are in non-redundant form. It keeps temporary result in redundant form and therefore the final result is produced in redundant form as well. Consequently, the result must be converted back to non-redundant form if further computation is needed, which is the case with all public key cryptography algorithms. For instance, a scalar point multiplication in ECC with moderate security level (e.g. 160-bit) requires hundreds of

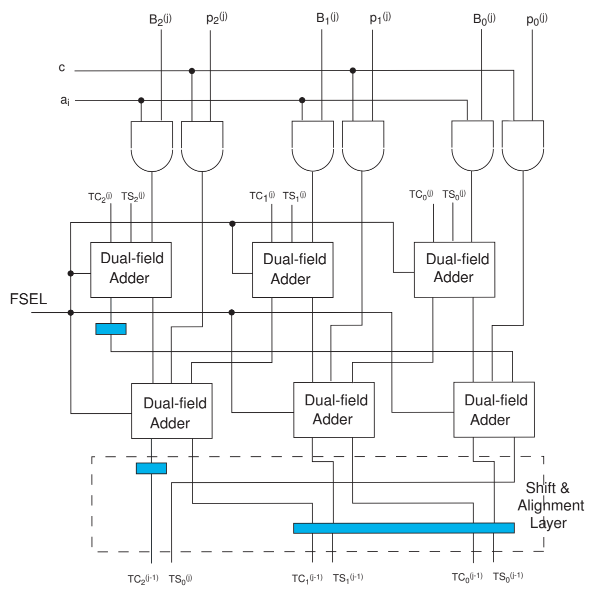


Figure 4.4: Processing Unit (PU) of the Original Unified Architecture with w = 3.

multiplications[2], which results in as many conversion operations.

The redundant representation used in previous unified architectures is the carry-save form, where an integer is represented as the sum of two other integers. The disadvantages of carry-save form are that i) two integers in carry-save form cannot be compared, and ii) subtraction is costly. Therefore, the partial results during the computations of cryptographic operations (i.e. elliptic curve scalar point multiplication, RSA exponentiation, etc.) must be converted back to the non-redundant form after every multiplication operation. The cost of the back transformation is two-fold: i) area for converter circuit and ii) time overhead (clock cycles) for reverse transformation. At the expense of extra overhead in time, the need for an extra inverter circuit can be eliminated as suggested in [43], where conversion

[2]More than a thousand multiplications are required for the same security level if the projective coordinates are used.

is achieved by repeated carry-save addition.

In summary, all previously proposed unified architectures are designed to efficiently perform a single field multiplication operation. They offer different properties to be appealing from various perspectives. The original unified architecture [34] utilizes single-radix, where the multiplier is scanned one bit at a time. [1, 38] proposes unified multipliers that scan the multiplier two or three bit at a time in order to reduce the cycle count without too much adverse effect on the critical path delay. The multiplier in [35] scans higher number of multiplier bits in $GF(2^n)$ mode than in $GF(p)$ mode in order to speedup the $GF(2^n)$ multiplication. The multipliers in [17, 43] are not scalable (i.e. work for a fixed precision) while the architecture in [43] is suitable for performing other field operations with the aid of conversion between redundant and non-redundant representations. Finally, [33] introduces a word-level (i.e. r-bit $\times$ r-bit) unified multiplier to be used in a ECC processor. An extensive comparison of all unified architectures and the proposed one is summarized in Table 4.8.

The proposed unified architecture is currently a single-radix implementation. However, it can easily be modified to work in higher radix or dual radices by applying the design techniques in [1, 38, 35]. There is support for other arithmetic operations such as comparison and subtraction in $GF(p)$-mode due to the new redundant signed representation. This support also exist in [43] at the expense of conversion operations from redundant to non-redundant representation.

Arc.	*GF*(3) support	Scalable	Conversion Necessary?	High Radix Possible?	Dual Radix Possible?	Support for Comparison &Subtraction
[1]	No	Yes	Yes	High-radix	No	No
[17]	No	No	Yes	No	No	No
[33]	No	No	Yes	No	No	No
[34]	No	Yes	Yes	Extensible	Extensible	No
[35]	No	Yes	Yes	High-radix	Dual-radix	No
[38]	No	Yes	Yes	High-radix	No	No
[43]	No	No	Yes	No	No	Yes
proposed	Yes	Yes	No	Extensible	Extensible	Yes

Table 4.8: Comparison of unified architectures(Arc.)

4.5 Power Consumption

The presented architecture was developed into Verilog modules and synthesized using the Synopsys Design Compiler tool. In the synthesis we used the TSMC 0.13 μm ASIC library. We assumed a target frequency of 500 MHz for synthesis results.

For proof of concept, we built and synthesized different PUs working on different fields. The implementations, denoted as A_1, A_2, and A_3, are those working in fields $GF(p)$-only, $GF(2^n)$-only, and $GF(3^m)$-only, respectively. In the second category, there are two unified architectures. The implementation, denoted as A_4, is a unified architecture working in both fields $GF(p)$ and $GF(2^n)$. Finally, the implementation A_5 is the unified architecture working in all three fields, namely $GF(p)$, $GF(2^n)$, and $GF(3^m)$. All five architectures are implemented for three different word sizes, 8, 16, and 32 and the implementation results of these architectures are summarized in Table 4.9.

From Table 4.9, the cost of unified architectures compared to $GF(p)$-only implemen-

word length	A_1		A_2		A_3	
	Average (μW)	Leakage (μW)	Average (μW)	Leakage (μW)	Average (μW)	Leakage (μW)
8	294.1275	6.8174	95.2853	2.3031	291.1586	7.7956
16	575.0258	13.1837	183.3014	4.4212	569.4142	15.2590
32	1131.6	25.8050	367.1780	8.6914	1122	30.0756

word length	A_4		A_5	
	Average (μW)	Leakage (μW)	Average (μW)	Leakage (μW)
8	336.9356	7.9878	393.1408	10.1903
16	657.6506	15.4158	767.4490	19.6494
32	1285.3	30.4312	1516.3	38.4589

Table 4.9: Power consumption of a PU with different word sizes.

tation can be captured as overhead in power consumption. We first investigated the first unified architecture A_4 that has a single datapath for $GF(p)$ and $GF(2^n)$ and compared it against the implementation results of a hypothetical architecture, denoted as $A_1 + A_2$, that has two separate datapaths for $GF(p)$ and $GF(2^n)$. For the hypothetical architecture A_1+A_2, the total power consumption is the sum of that of the A_1 and A_2 architectures. The implementation results are summarized in Table 4.10 and Table 4.11. The improvement of the architecture is found to be about 12%-14%.

word length	Average Power(μW)				
	A_1	A_2	$A_1 + A_2$	A_4	Improvement
8	294.1275	95.2853	389.4128	336.9356	13.47%
16	575.0258	183.3014	758.3272	657.6506	13.27%
32	1131.6	367.1780	1498.778	1285.3	14.24%

Table 4.10: Advantage of the unified architecture A_4, for $GF(p)$ and $GF(2^n)$ for average power consumption

word	Leakage Power(μW)				
length	A_1	A_2	$A_1 + A_2$	A_4	Improvement
8	6.8174	2.3031	9.1205	7.9878	12.41%
16	13.1837	4.4212	17.6049	15.4158	12.43%
32	25.8050	8.6914	34.4964	30.4312	11.78%

Table 4.11: Advantage of the unified architecture A_4, for $GF(p)$ and $GF(2^n)$ for leakage power

Similarly, we also investigated the advantage of the unified architecture, A_5 over a hypothetical architecture, $A_1 + A_2 + A_3$, that has three separate datapaths for the fields $GF(p)$, $GF(2^n)$, and $GF(3^m)$. The results, summarized in Table 4.12 and Table 4.13, show that the advantage of using the unified architecture, A_5 is around 40% in terms of power consumption. The improvement figures in Table 4.12 and Table 4.13 clearly demonstrate that the unified architecture A_5 provides far superior performance compared to the classical unified architectures working for only the fields $GF(p)$ and $GF(2^n)$.

word	Average Power(μW)		
length	$A_1 + A_2 + A_3$	A_5	Improvement
8	680.5714	393.1408	42.23%
16	1327.7414	767.4490	42.19%
32	2620.778	1516.3	42.14%

Table 4.12: Advantage of unified architecture A_5, for $GF(p)$, $GF(2^n)$, and $GF(3^m)$ for average power consumption.

In order to see more clearly what one can gain with the new unified architecture A_5 over the classical one, A_4, we also compared the two unified architectures in terms of power consumption. The results summarized in Table 4.14 highlights the advantage of the new unified architecture over the classical one, which is at least 35%.

word length	Leakage Power(μW)		
	$A_1 + A_2 + A_3$	A_5	Improvement
8	16.9161	10.1903	39.76%
16	32.8639	19.6494	40.21%
32	64.572	38.4589	40.44%

Table 4.13: Advantage of unified architecture A_5, for $GF(p)$, $GF(2^n)$, and $GF(3^m)$ for leakage power.

word length	Average Power(μW)			Leakage Power(μW)		
	$A_4 + A_3$	A_5	Improvement	$A_4 + A_3$	A_5	Improvement
8	628.0942	393.1408	37.41%	15.7834	10.1903	35.44%
16	1227.0648	767.4490	37.43%	30.6748	19.6494	35.94%
32	2407.3	1516.3	37.01%	60.5068	38.4589	36.44%

Table 4.14: Advantage of the new unified architecture A_5 over the classical unified architecture A_4

4.6 A Note on Side-Channel Attacks

In this section we would like to briefly comment on the side-channel characteristics of the proposed RSD multiplier as it is crucial to prevent information leakage through so-called side-channels (i.e. execution time, power consumption, EM and temperature profiles etc.) in cryptographic applications. We would like to note that most of the side-channel countermeasures are typically applied at either the algorithm or the circuit levels. For instance, an effective DPA counter-measure implemented at the algorithm layer is randomized exponentiation [9]. On the other hand, at the circuit level masking techniques may be applied [25]. At even lower levels, so-called power balanced cell libraries [39, 40, 30] which provide IC primitives that (ideally) have power consumption which is independent of the input bits, may be utilized. Any one of these techniques can be used alongside with the proposed multiplier. For instance, the presented architecture may be re-synthesized using a power balanced library at the cost of growing the area by roughly 2-3 times. On the other hand, a similar increase in area would be expected if the (non-unified) multiplier units are separately re-synthesized with the same cell library.

As far as the side-channel performance of individual components at the arithmetic level are concerned we could identify very little work in the literature. In [42] Walter and Samyde demonstrated a direct correlation between the Hamming weights of the operands, and the power traces obtained during their multiplication. The authors conclude that it would be possible to gain useful side-channel information from a parallel multiplier built using Wallace trees. The processing element used in the multiplier proposed in this chapter utilizes a redundant representation which will significantly reduce (if not eliminate) the correlation between the power traces from the Hamming weight of the operands. We can clearly claim that the proposed multiplier will be more resilient from this perspective than more traditional multipliers to side-channel attacks. Furthermore, the same reference ([42]) considers pipelining to be an effective counter-measure to power attacks as multiple words of the operands are processed together. This will make the task of discerning operand bits from power traces more difficult. The proposed architecture, therefore, has an additional level of protection against side-channel attacks due to its highly pipelined design.

We modeled the proposed PU design in Verilog with a word size of 4 bits. We used the Synopsys tools Design Compiler and Power Compiler for synthesizing our designs and Modelsim for simulation. Our target was the TSMC 0.13 μm ASIC library, which is characterized for power. We assumed a target frequency of 500 MHz for dynamic power analysis. We analyzed all five designs A_{1-5} using the power analysis design flow. For designs A_4 and A_5, we analyzed the PU architecture for all the supported modes. Figures 4.5, 4.6 and 4.7 show the dynamic power profiles of the A_1, A_2 and A_3 implementations, respectively. Figures 4.8 and 4.9 show the dynamic power profiles of the A_4 design with $GF(p)$ and $GF(2)$ modes, respectively. Figures 4.10, 4.11, 4.12 show the dynamic power profiles of the A_5 implementation with $GF(p)$, $GF(2)$ and $GF(3)$ modes, respectively.

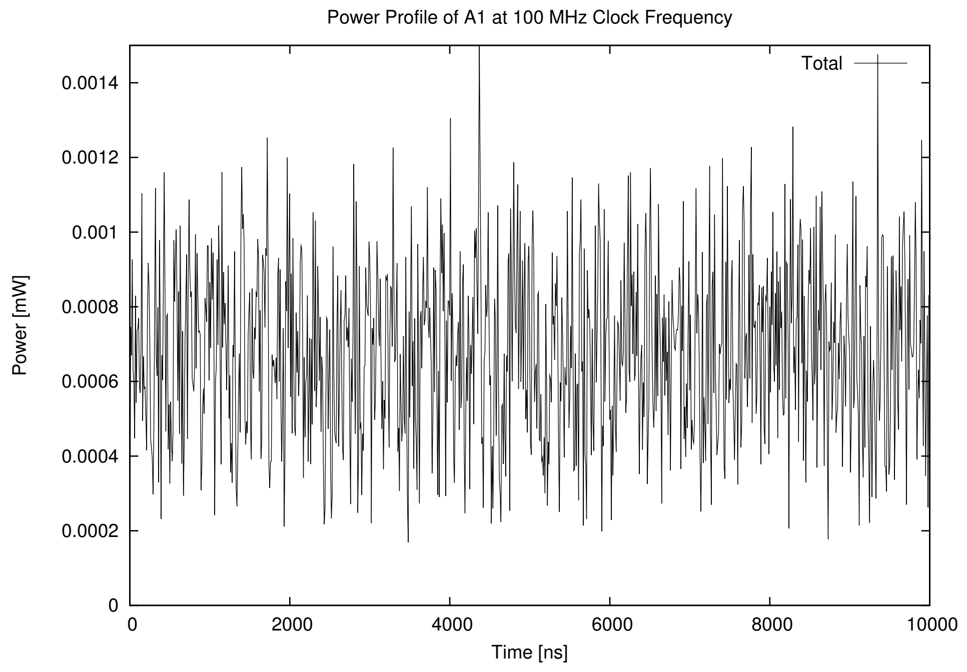


Figure 4.5: Power profile of the design A1 at 100MHz clock frequency

From these figures we can conclude that the unified architectures consume less power than that of the single $GF(p)$ and $GF(3)$ architectures. For $GF(2)$ operations, it is more feasible to utilize a single $GF(2)$ arithmetic unit. Since we are targeting architectures that

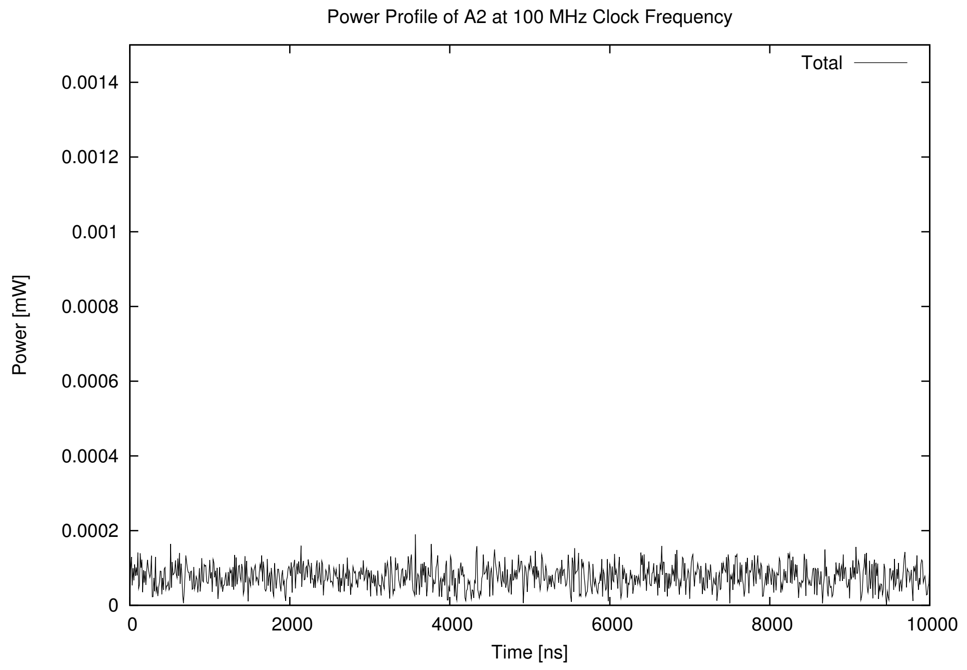


Figure 4.6: Power profile of the design A2 at 100MHz clock frequency

will operate on all the 3 fields, such as cryptographic accelerators with pairing support, it is much more feasible to use a unified architecture than single field arithmetic units. Also, we speculate that since the proposed unified architecture shows a more homogeneous power profile for all the 3 modes, it is promising to be more resistant against power attacks.

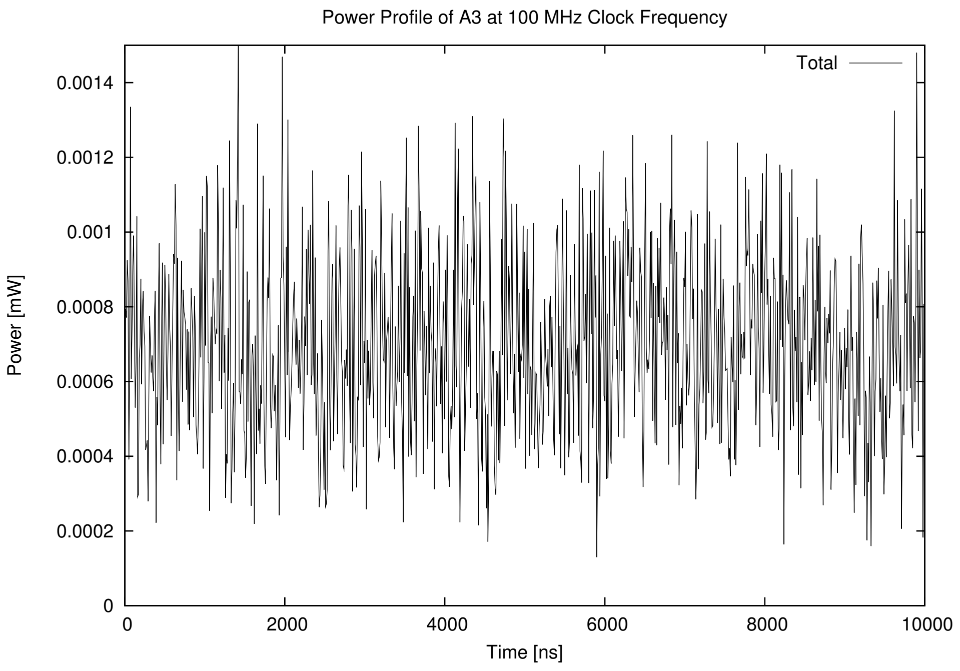


Figure 4.7: Power profile of the design A3 at 100MHz clock frequency

Algorithm 2 Montgomery multiplication algorithm for $GF(p)$

Require: $A, B \in GF(p)$ and p

Ensure: $C = A \cdot B \cdot 2^{-n} \in GF(p)$, where $n = \lceil \log_2 p \rceil$

```
1:  $T := 0^n$
2:  for $i$ from $0$ to $n-1$ do
3:      $(Carry|T^{(0)}) := a_i \cdot B^{(0)} + T^{(0)}$
4:      $Parity := T_0^{(0)}$
5:      $(Carry|T^{(0)}) := Parity \cdot p^{(0)} + (Carry|T^{(0)})$
6:      for $j$ from $1$ to $e-1$ do
7:          $(Carry|T^{(j)}) := a_i \cdot B^{(j)} + T^{(j)} + Parity \cdot p^{(j)} + Carry$
8:          $T^{(j-1)} := (T_0^{(j)}|T_{w-1..1}^{(j-1)})$
9:      end for
10:     $T^{e-1} := (Carry|T_{w-1..1}^{(e-1)})$
11: end for
12: $C := T$
13: if $C > p$ then $C := C - p$
14: return $C$
```

Algorithm 3 Montgomery multiplication algorithm for $GF(3^m)$

```
Require: $A(x), B(x) \in GF(3^m)$ and $p(x)$
Ensure: $C(x) = A(x) \cdot B(x) \cdot 3^{-m} \in GF(3^m)$, where $m$ is the degree of $p(x)$
 1: $T(x) := 0$
 2: for $i$ from 0 to $m-1$ do
 3:     $T^{(0)} := a_i \cdot B^{(0)} + T^{(0)}$
 4:     if $T_0^{(0)} = p_0^{(0)}$ then
 5:         $T^{(0)} := T^0 - p^{(0)}$
 6:         for $j$ from 1 to $e-1$ do
 7:             $T^{(j)} := a_i \cdot B^{(j)} + T^{(j)} - p^{(j)}$
 8:             $T^{(j-1)} := (T_0^{(j)} | T_{w-1..1}^{(j-1)})$
 9:         end for
10:     else
11:         $T^{(0)} := T^0 + p^{(0)}$
12:         for $j$ from 1 to $e-1$ do
13:             $T^{(j)} := a_i \cdot B^{(j)} + T^{(j)} + p^{(j)}$
14:             $T^{(j-1)} := (T_0^{(j)} | T_{w-1..1}^{(j-1)})$
15:         end for
16:     end if
17:     $T^{e-1} := ((0,0) | T_{w-1..1}^{(e-1)})$
18: end for
19: return $T(x)$
```

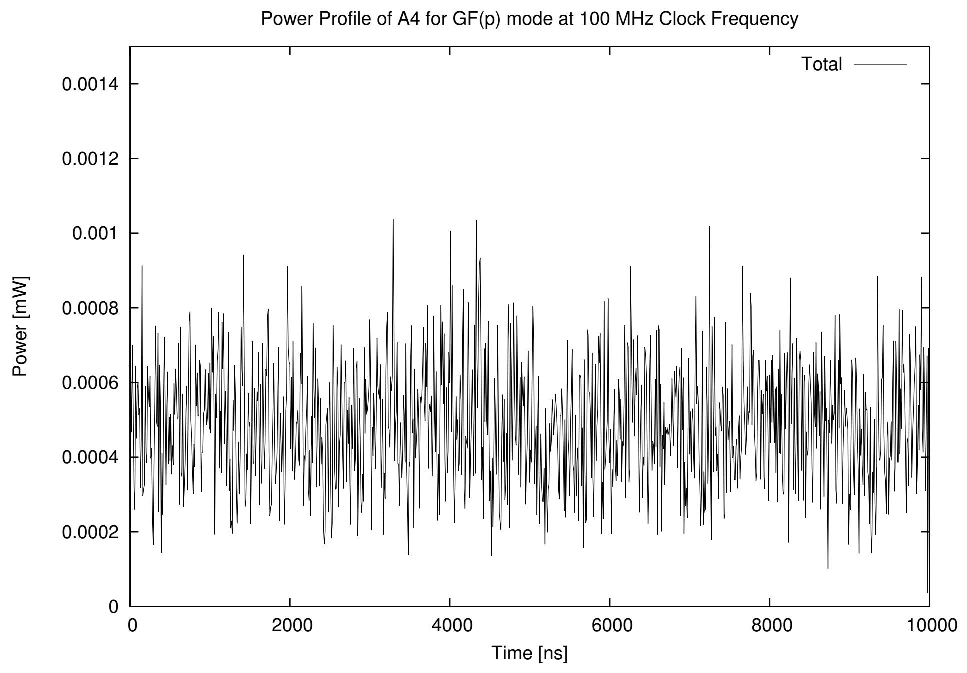


Figure 4.8: Power profile of the design A4 for $GF(p)$ mode at 100MHz clock frequency

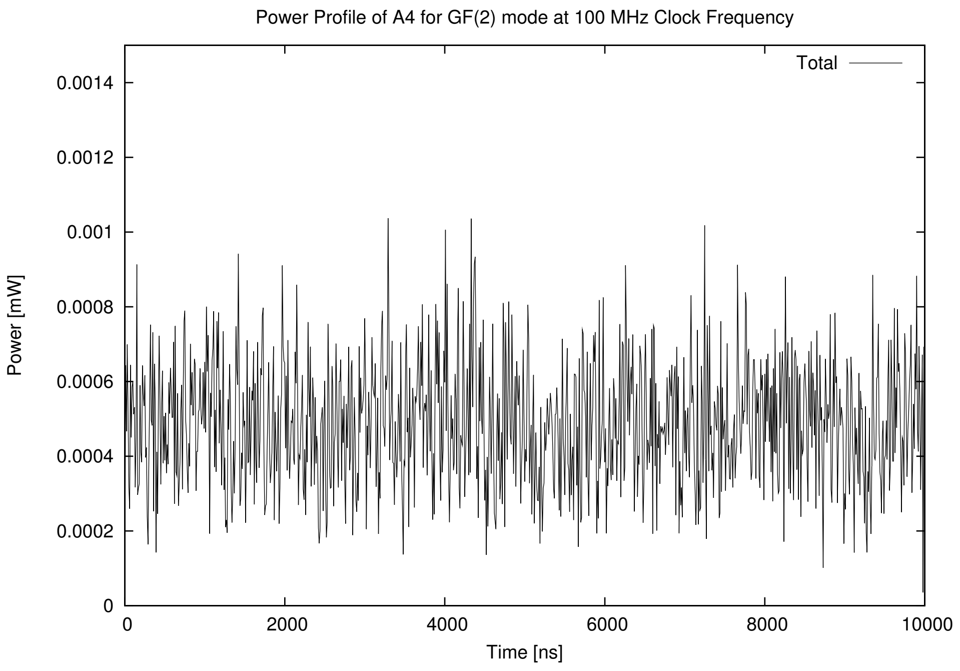


Figure 4.9: Power profile of the design A4 for $GF(2)$ mode at 100MHz clock frequency

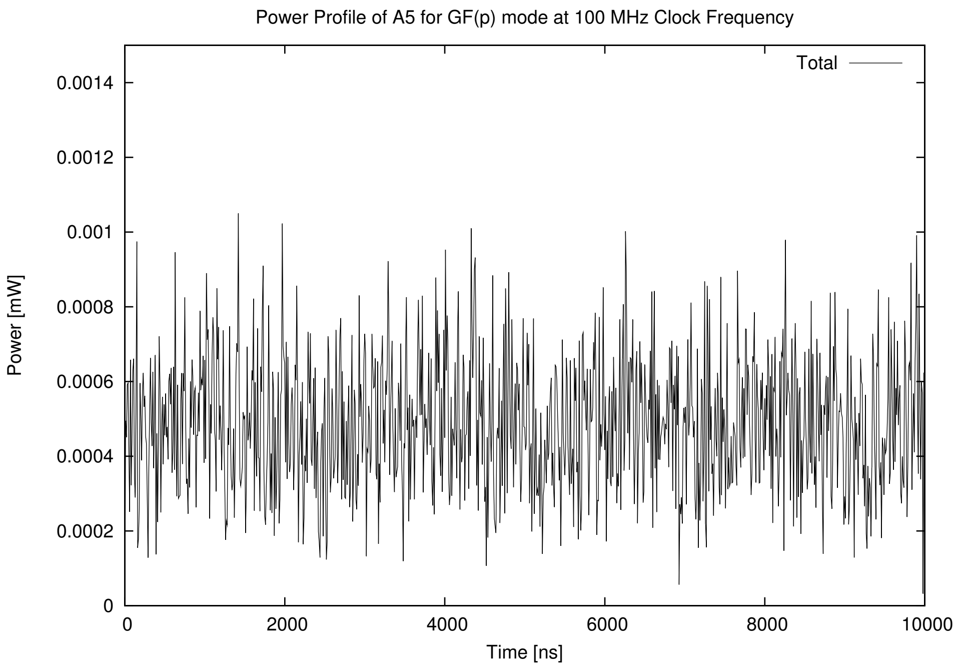


Figure 4.10: Power profile of the design A5 for $GF(p)$ mode at 100MHz clock frequency

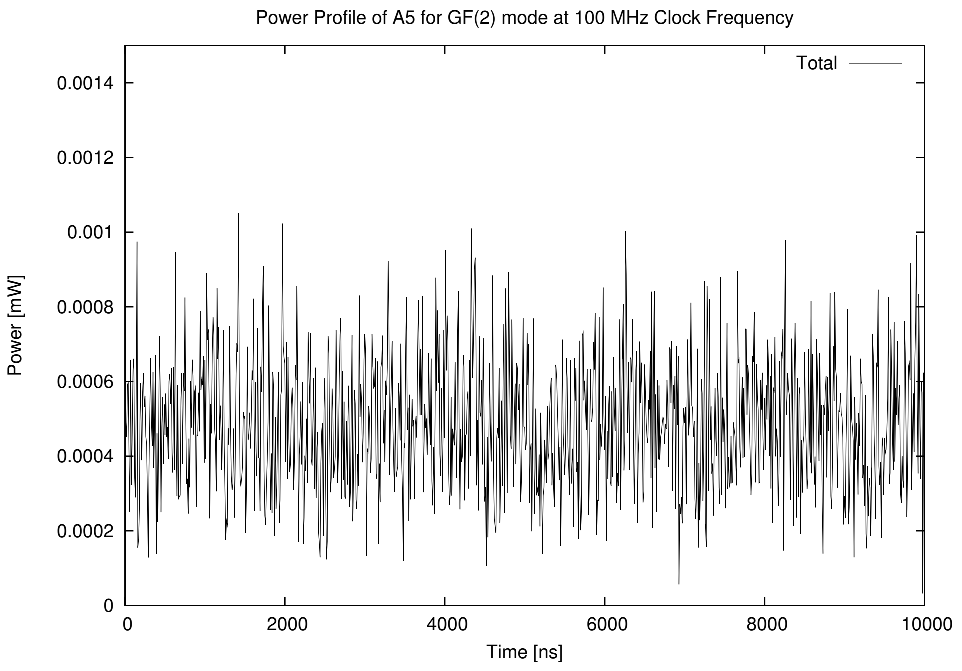


Figure 4.11: Power profile of the design A5 for $GF(2)$ mode at 100MHz clock frequency

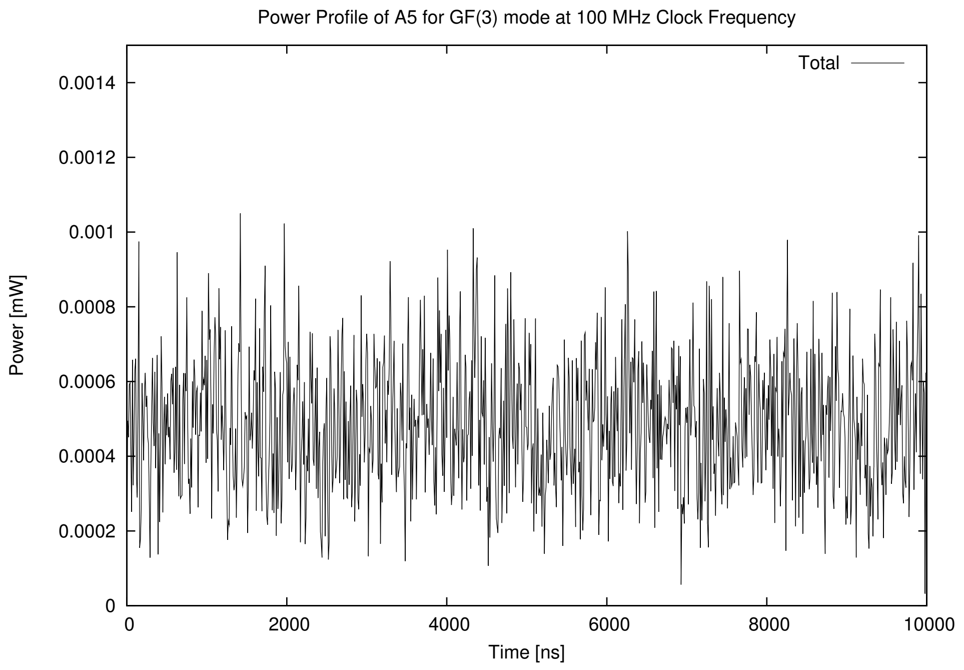


Figure 4.12: Power profile of the design A5 for $GF(3)$ mode at 100MHz clock frequency

Chapter 5

Tamper-Resilient Tate Pairing Architectures

Parts of this chapter were presented in [27]. Gunnar Gaubatz helped with the application of robust codes to Tate Pairing.

Our goal is to use strong error detecting codes to build tamper-resilient algorithms and architectures for Tate pairing computations. Here we specifically focus on the Kwon-BGOS algorithm which computes a pairing over characteristic 3, although it is fairly straightforward to generalize our techniques to support other pairing algorithms defined over different characteristics. The error model will apply with minor variations. For instance, for the Duursma-Lee Tate pairing algorithm, one would have to extend the existing units with a cubic-root computation circuit.

Our objective is to protect the arithmetic operations used in a Tate pairing computation against a sufficiently large class of error patterns, while keeping the overhead in performance low. As mentioned in the previous section, full robustness can only be achieved with $r = k$, resulting in a duplication of the operand size and likely more than 100% overhead. Furthermore, as we will show in the next section, only very specific choices of the sub-matrix P allow us to define arithmetic operations in such a way that the check-symbol of the result can be predicted efficiently based on the input operands' check-symbols.

An alternative method for tamper resilient multi-precision arithmetic was presented in [14], which achieves total robustness by encoding single digits separately. Such a high degree of protection, however, can only be obtained by accepting a substantial amount of overhead on the predictor and error detection networks. Depending on the anticipated threat level, such rigor may not be required. As long as the probability of error detection is sufficiently high to render malicious fault insertions infeasible, a lighter-weight scheme will work as well.

For our purposes we need a linear code over $GF(q)$ with rank$(P) = n - k$, which we will transform into an error resilient code. The error correcting properties and ease of decoding are irrelevant in this setting. For our purposes it suffices to select a simple linear code which will allow us to build resilient versions of the arithmetic operations (as described in the next section) in an efficient manner. We therefore pick a simple parity code of length $n = k + 1$. Note that by choosing $r = 1$ the $r \times n$ error check matrix $H = [P|I_r]$ becomes simply $H = [1\ 1\ 1 \ldots 1\ 1]$. Hence, for a vector a representing an element of $GF(q^k)$, the matrix-vector product Pa may be computed by simply summing the coefficients of $a = (a_0, a_1, \ldots, a_{k-1})$, i.e. $Pa = \sum_{i=0}^{k-1} a_i$. Thus the resilient encoded form of $a \in GF(q^k)$ is simply

$$\left(a, (\sum_{i=0}^{k-1} a_i)^2\right)$$

The Duursma-Lee Algorithm and The Kwon-BGOS Tate Pairing algorithm (Fig.2.2) both include arithmetic operation in $GF(3^m)$ and $GF(3^{6m})$. Our approach could be applied to either the base field $GF(3^m)$ or the extension field $GF(3^{6m})$. If the proposed approach is applied to the base field, each $GF(3^m)$ operation utilized to calculate the $GF(3^{6m})$ multiplication and cubing operations in Algorithm 1 will have an error detection scheme, which will cause a huge overhead on the latency of the algorithm, as well as the area utilization. However, if we apply our approach to the extension field, there will be only one error detection scheme for each $GF(3^{6m})$ operation. This will reduce the overhead of our approach significantly. Thus, we decided to build our scheme on the extension field $GF(3^{6m})$.

The Duursma-Lee Algorithm [10] includes one $GF(3^{6m})$ multiplication operation, while Kwon-BGOS algorithm includes one multiplication and one cubing operations in $GF(3^{6m})$. Regardless of the complexity and efficiency of both algorithms, in order to show proof of concept and examine our approach in a broader spectrum of operations, we chose to examine the Kwon-BGOS algorithm (Fig.2.2) and build a tamper resilient version of this algorithm.

5.1 Lightweight and Tamper-Resilient Error Detection Built Over the Extension Field

For our purposes we need a linear code over $GF(3^{6m})$. Following the notation from Section 2.4, we have $q = 3^m$ and $k = 6$, and thus $GF(q)$ will become $GF(3^m)$, while $GF(q^k)$ will be $GF(3^{6m})$. Let $f \in GF(3^{6m})$ represent an operand used in the Kwon-BGOS Tate pairing algorithm scheme. Let $w = (Pf)^2 \in GF(3^m)$. For proof of concept, we built our Modified Tate Pairing scheme using a parity code for $P = [1\ 1\ 1 \ldots\ 1]$. We assumed that the introduced error e_f is in $GF(3^{6m})$ and e_w is in $GF(3^m)$.

Based on the original definition of robust codes by Karpovsky and Taubin [18], we derive a slightly modified construction which allows us to accommodate the specific arithmetic needs of the pairing computation. Specifically, the original robust codes were defined over $\mathrm{GF}(p)$, with $p > 2$ a prime, while we extend the definition for codes over field extensions $\mathrm{GF}(q)$, where $q = p^m$. The resulting construction is no longer robust in the sense defined by Karpovsky. However, it will have significantly lower overhead.

As in the original paper, we use a simple, but appropriate additive error model, i.e. when an error $e = (e_f, e_w)$ is introduced to an operand f, the operand becomes $(f + e_f, w + e_w)$. Thus, if

$$w + e_w \neq (P(f + e_f))^2 \tag{5.1}$$

then our error detection scheme will work and the error detection network will flag an error.

Definition 3 *Let V' be a linear q-ary parity code ($q = p^m$, $p > 2$ is a prime) with $n = k+1$*

and check matrix $H = [P|I]$ with rank$(P) = 1$. Then $C_{V'} = \{(f, w)|f \in GF(q^k), w = (Pf)^2 \in GF(q)\}$.

We capture the performance of this specific non-linear error detection code with the following theorem.

Theorem 2 *A non-zero error $e = (e_f, e_w)$ on a code word $(f, w) \in C_{V'}$ will not be detected if and only if it satisfies the error masking equation*

$$(Pf)^2 + e_w = (P(f + e_f))^2 \,. \tag{5.2}$$

For $C_{V'}$ the set of undetectable errors ($\{e|Q(e) = 1\}$) is a $(k-1)$-dimensional subspace of V', $q^k - q^{k-1}$ errors are detected with probability 1, and the remaining $q^{k+1} - q^k$ errors are detected with probability $1 - q^{-1}$.

Proof 1 *From* (5.1)*, we have*

$$2(Pf)(Pe_f) + (Pe_f)^2 = e_w \tag{5.3}$$

There are three cases that will satisfy equation (5.3)*:*

1. *$Pe_f = e_w = 0$: For this condition, Equation* (5.3) *is satisfied for all f. Since e_f is in $GF(q^k)$:*

$$Pe_f = \Sigma_{i=0}^{k-1} e_{f_i} = 0 \tag{5.4}$$

The number of e_f satisfying Equation (5.4) *is q^{k-1}. Also, there is only one e_w satisfying $e_w = 0$. Thus, the total number of errors $e = (e_f, e_w)$ satisfying this condition is q^{k-1}.*

2. *$Pe_f = 0$ and $e_w \neq 0$: Equation* (5.3) *is not satisfied for any f. The number of e_f satisfying $Pe_f = 0$ was calculated to be q^{k-1}. In addition to this, we need to calculate the number of errors e_w satisfying $e_w \neq 0$. Since $e_w \in GF(q)$, total number of e_w for $e_w \neq 0$ is $q - 1$. Thus, the total number of errors $e = (e_f, e_w)$ satisfying this condition is $q^{k-1} \cdot (q-1) = q^k - q^{k-1}$.*

3. *$Pe_f \neq 0$: for any $e = (e_f, e_w)$ there exists a unique Pf satisfying Equation* (5.3)*. Let f be randomly selected and let e_f be satisfying the condition $Pe_f \neq 0$ The number of errors e_f satisfying this condition is $q^k - q^{k-1}$. The total number of errors e_w is q. Thus, the total number of errors $e = (e_f, e_w)$ is $(q^k - q^{k-1}) \cdot q = q^{k+1} - q^k$. The probability that a randomly selected f and an error e_f such that $Pe_f \neq 0$ will not be satisfying Equation* (5.3) *is $1 - q^{-1}$.*

Example 1 *For the modified Tate pairing algorithm we have $k = 6$ and $q = 3^m$. Thus the number of undetected errors is 3^{5m}, the number of reliably detected (with probability* 1*) errors is $3^{6m} - 3^{5m}$, and $3^{7m} - 3^{6m}$ errors are detected with probability $1 - 3^{-m}$.*

This probability is incorrectly calculated in [27]. While it provides some level of protection, the technique is not robust as defined by Karpovsky et al [18].

Even though this method is sufficient against weak adversaries, a more advanced attacker can still easily bypass this error detection mechanism and manipulate the device under attack. Here we should note that a weak attacker can only introduce random errors to the circuit. The weak attack model is easy to carry out. An attacker can change the stored values in the circuit with conventional methods: temperature variation and laser heating induced attacks, clock glitching, etc. This type of attack requires little control. It means that an attacker can only introduce random errors to the circuit and then observe the output. On the other hand, a strong attacker can insert any additive error pattern of his choosing. Thus, a strong attacker can identify and introduce errors that are not detectable by the circuit and deduce information from the arithmetic operations carried out in the circuit.

As seen from Algorithm 1, the implementation of efficient arithmetic for $GF(3^{6m})$ and $GF(3^m)$ is essential in Tate Pairing algorithm of Kwon-BGOS. These field operations determine the efficiency and the complexity of the overall Tate Pairing operation. Thus, in order to build an efficient, yet tamper-resilient architecture for Tate Pairing, it is crucial to implement resilient arithmetic primitives that are light-weight for $GF(3^{6m})$ as well as $GF(3^m)$. Sections 5.1.1 and 5.2.1 will provide more detail on the performance analysis of the field operations in $GF(3^{6m})$ and $GF(3^m)$, respectively.

5.1.1 $GF(3^{6m})$ Operations

The elements of $GF(3^{6m})$ are represented in the basis $\{1, \sigma, \rho, \sigma\rho, \rho^2, \sigma\rho^2\}$. Let $a \in GF(3^{6m})$ and

$\{\zeta^0, \zeta^1, \zeta^2, \zeta^3, \zeta^4, \zeta^5\}$ = $\{1, \sigma, \rho, \sigma\rho, \rho^2, \sigma\rho^2\}$. Then,

$$a = \Sigma_{k=0}^{5} a_k \zeta^k = a_0 + a_1\sigma + a_2\rho + a_3\sigma\rho + a_4\rho^2 + a_5\sigma\rho^2$$

where $a_k \in GF(3^m)$. It was noted previously that σ and ρ are the zeros of $\sigma^2 + 1$ and $\rho^3 - \rho \mp 1$, respectively and thus satisfy $\sigma^2 + 1 = \rho^3 - \rho \mp 1 = 0 \in GF(3^{6m})$.

Following this notation, multiplication and cubing operations in $GF(3^{6m})$ can be performed utilizing $GF(3^m)$ arithmetic. In this section, we will present explicit formulations of $GF(3^{6m})$ arithmetic. The equations presented could be further optimized for hardware implementation. However, our aim is to explain only the mathematical background of these operations. Also, we present the mathematical background for our error detection strategy as it is applied to $GF(3^{6m})$ operations.

Multiplication

Step 12 of Algorithm 1 is a multiplication operation in $GF(3^{6m})$. The two elements, f and $g \in GF(3^{6m})$ are multiplied. We can break this multiplication operation down into arithmetic in characteristic three as follows:

$$\begin{aligned}
f &= f_0 + f_1 \cdot \sigma + f_2 \cdot \rho + f_3 \cdot \sigma\rho + f_4 \cdot \rho^2 + f_5 \cdot \sigma\rho^2 \\
g &= g_0 + g_1 \cdot \sigma + g_2 \cdot \rho - \rho^2 \;\; (g_3 = g_5 = 0, g_4 = -1) \\
r &= f \cdot g \\
&= (f_0g_0 - f_1g_1 + f_5g_2 - f_2) + \\
&\quad (f_1g_0 - f_0g_1 + f_4g_2 - f_3) \cdot \sigma + \\
&\quad (f_2g_0 - f_3g_1 + f_0g_2 + f_5g_2 - f_2 - f_5) \cdot \rho + \\
&\quad (f_3g_0 - f_2g_1 + f_1g_2 + f_4g_2 - f_3 - f_4) \cdot \sigma\rho + \\
&\quad (f_5g_0 - f_4g_1 + f_2g_2 - f_0 - f_5) \cdot \rho^2 + \\
&\quad (f_4g_0 - f_5g_1 + f_3g_2 - f_1 - f_4) \cdot \sigma\rho^2
\end{aligned}$$

We pick a simple parity code and apply the approach explained in Section 5.1:

$$\begin{aligned}
w_f &= (f_0 + f_1 + f_2 + f_3 + f_4 + f_5)^2 \\
w_g &= (g_0 + g_1 + g_2 - 1)^2 \\
w_r &= ((f_0g_0 - f_1g_1 + f_5g_2 - f_2) \\
&\quad + (f_1g_0 - f_0g_1 + f_4g_2 - f_3) \\
&\quad + (f_2g_0 - f_3g_1 + f_0g_2 + f_5g_2 - f_2 - f_5) \\
&\quad + (f_3g_0 - f_2g_1 + f_1g_2 + f_4g_2 - f_3 - f_4) \\
&\quad + (f_5g_0 - f_4g_1 + f_2g_2 - f_0 - f_5) \\
&\quad + (f_4g_0 - f_5g_1 + f_3g_2 - f_1 - f_4))^2 \\
&= ((f_0 + f_1 + f_2 + f_3 + f_4 + f_5) \cdot g_0 \\
&\quad + (f_0 - f_1 + f_2 - f_3 - f_4 + f_5) \cdot g_1 \\
&\quad + (f_0 + f_1 + f_2 + f_3 - f_4 - f_5) \cdot g_2 \\
&\quad + (-f_0 - f_1 + f_2 + f_3 + f_4 + f_5))^2 \\
&= (\sqrt{w_f}\sqrt{w_g} + f_1g_1 + f_3g_1 + f_4g_1 + f_4g_2 + f_5g_2 - f_2 - f_3 - f_4 - f_5)^2 \\
&= w_fw_g + (f_1g_1 + f_3g_1 + f_4g_1 + f_4g_2 + f_5g_2 - f_2 - f_3 - f_4 - f_5)^2 \\
&\quad +2 \cdot (f_1g_1 + f_3g_1 + f_4g_1 + f_4g_2 + f_5g_2 - f_2 - f_3 - f_4 - f_5) \cdot \sqrt{w_f}\sqrt{w_g} \\
&= w_fw_g + T_1^2 + T_2
\end{aligned}$$

where

$$\begin{aligned} T_1 &= f_1g_1 + f_3g_1 + f_4g_1 + f_4g_2 + f_5g_2 \\ &\quad -f_2 - f_3 - f_4 - f_5 \\ T_2 &= 2 \cdot (f_1g_1 + f_3g_1 + f_4g_1 + f_4g_2 + f_5g_2 \\ &\quad -f_2 - f_3 - f_4 - f_5) \cdot \sqrt{w_f}\sqrt{w_g} \end{aligned}$$

As an outcome of these equations, we can clearly state that the check-symbol of the result of the multiplication, w_r, can be extracted from the check-symbols of the inputs, w_f and w_g. This extraction logic includes 1 multiplication in $GF(3^m)$ for $w_f w_g$, 2 multiplications, 4 additions and 4 subtractions for T_2 and 1 square operation for T_1^2. The calculation of w_g costs 1 square and 2 addition operations. Also, the calculation of w_r costs 1 square operation. Since f is the result of another $GF(3^{6m})$ operation, the calculation of w_f is not included in the overhead analysis, because it is calculated as w_r of another operation. Thus, the overall overhead for this approach is 3 multiplication, 3 square, 6 addition and 4 subtraction operations. The standard implementation of Algorithm 1 utilizes 18 multiplications in the base field $GF(3^m)$ plus a number of addition/subtraction operations [19]. Since the operational complexity of addition/subtraction operations are small compared to multiplication operations in Algorithm 1, we will only calculate the overhead of the operations in terms of multiplication and cubing operations in $GF(3^m)$ for the rest of the chapter.

Cubing

Step 11 of Algorithm 1 is a cubing operation in $GF(3^{6m})$. This cubing operating is carried out with 6 $GF(3^m)$ cubing operations. As shown in step 11, the element $f \in GF(3^{6m})$ is cubed. The equations for the cubing operation are as follows:

$$\begin{aligned}
f &= f_0 + f_1 \cdot \sigma + f_2 \cdot \rho + f_3 \cdot \sigma\rho + f_4 \cdot \rho^2 \\
&\quad + f_5 \cdot \sigma\rho^2 \\
f^3 &= f_0^3 + f_1^3 \cdot \sigma^3 + f_2^3 \cdot \rho^3 + f_3^3 \cdot \sigma^3\rho^3 + f_4^3 \cdot \rho^6 \\
&\quad + f_5^3 \cdot \sigma^3\rho^6 \\
&= \left(f_0^3 + f_2^3 + f_4^3\right) - \left(f_1^3 + f_3^3 + f_5^3\right) \cdot \sigma + \left(f_2^3 - f_4^3\right) \cdot \rho \\
&\quad + \left(f_5^3 - f_3^3\right) \cdot \sigma\rho + f_4^3 \cdot \rho^2 - f_5^3 \cdot \sigma\rho^2
\end{aligned}$$

We pick a simple parity code and apply the approach explained in Section 5.1:

$$\begin{aligned}
w_f &= \left(f_0 + f_1 + f_2 + f_3 + f_4 + f_5\right)^2 \\
w_{f^3} &= \left(f_0^3 - f_1^3 - f_2^3 + f_3^3 + f_4^3 - f_5^3\right)^2 \\
&= \left(\left(f_0^3 + f_1^3 + f_2^3 + f_3^3 + f_4^3 + f_5^3\right) + \left(f_1^3 + f_2^3 + f_5^3\right)\right)^2 \\
&= w_f^3 + \left(f_1^3 + f_2^3 + f_5^3\right)^2 + 2 \cdot \left(f_0^3 + f_1^3 + f_2^3 + f_3^3 + f_4^3 + f_5^3\right)\left(f_1^3 + f_2^3 + f_5^3\right) \\
&= w_f^3 + T_3^2 + T_4
\end{aligned}$$

where

$$\begin{aligned} T_3 &= \left(f_1^3 + f_2^3 + f_5^3\right) \\ T_4 &= 2 \cdot \left(f_0^3 + f_1^3 + f_2^3 + f_3^3 + f_4^3 + f_5^3\right) \cdot \\ &\quad \left(f_1^3 + f_2^3 + f_5^3\right) \end{aligned}$$

According to these equations, we can state that the resulting check-symbol of the cubing operation, w_{f^3}, can be extracted from the check-symbol of the input, w_f. This extraction logic includes 1 cubing in $GF(3^m)$ for w_f^3, 1 multiplication for T_4 and 1 square operation for T_3^2. Also, the calculation of w_{f^3} costs 1 square operation. Again, since f is a result of another $GF(3^{6m})$ operation, the calculation of w_f is not included in the overhead analysis, because it is calculated as w_r of another operation. Thus, the overall overhead for this approach is 1 cubing, 1 multiplication and 2 square operations.

5.1.2 Performance Analysis

In this section we will summarize the complexity analysis of Algorithm 2.2 and the overhead caused by using the tamper-resilient arithmetic.

Mathematical Analysis

$\mathbf{GF(3^{6m})}$ operations. The $GF(3^{6m})$ multiplication operation in the Kwon-BGOS Tate Pairing algorithm is carried out with 18 $GF(3^m)$ multiplications, while the cubing operation in $GF(3^{6m})$ is carried out with 6 $GF(3^m)$ cubing operations [19]. Table 5.1 shows the

	# $GF(3^m)$ operations	
$GF(3^{6m})$ operations	Standard Implement.	Resilience Overhead
Mult.	18 muls	3 muls, 3 square
Cube	6 cube	1 cube, 1 mul, 2 square

Table 5.1: Complexity of $GF(3^{6m})$ operations for standard and resilient implementations.

	# $GF(3^m)$ operations	
Step	Standard Implement.	Resilience Overhead
2	1 cube	1 square, 1 cube
3	1 cube	1 square, 1 cube
6	2 cube	2 square, 2 cube
7	2 cube	2 square, 2 cube
8	1 add	3 square, 1 mul
9	1 mul	1 square, 1 mul
	1 square	1 square

Table 5.2: Number of additional $GF(3^m)$ multiplication operations required for implementing resilient $GF(3^m)$ operations.

complexity of $GF(3^{6m})$ operations for standard implementation, plus the additional overhead required for resilient implementation. It should be noted that these numbers are not dependent on the field size m, since we built our scheme on the extension field.

$\mathbf{GF(3^m)}$ operations Table 5.2 shows the detailed number of $GF(3^m)$ operations required for the remaining steps of the algorithm. Also, the table illustrates the additional number of operations required for the resilient approach.

$GF(3^m)$ operations	# of operations utilized	
	Standard Implement.	Resilience Overhead
Cube	6	1
Mult.	18	9

Table 5.3: Total number of $GF(3^m)$ multiplication, cubing and square operations required for Algorithm 1.

Hardware and Software Implementations

The implementation of Algorithm 2.2 requires one multiplication and one cubing operation in $GF(3^{6m})$. Following from Table 5.1, multiplication is realized with 18 base field multiplications and the cubing operation is realized with 6 base field cubing operations. The overhead caused by our approach can also be calculated from these two tables. The total overhead for a $GF(3^{6m})$ multiplication is 3 multiplications and 3 squaring operations carried out in the base field $GF(3^m)$. For the $GF(3^{6m})$ cubing operation, the total overhead is 1 cubing, 1 multiplication and 2 squaring operations in $GF(3^m)$. Table 5.3 gives the total number of $GF(3^m)$ multiplication and cubing operations required to realize Algorithm 2.2 (assuming that the complexity of a squaring operation is roughly equal to the complexity of a multiplication operation) as well as the overhead caused.

If the extension field arithmetic of Algorithm 1 is implemented in hardware, there are various area/speed trade-offs one can make to tailor the implementation to meet the overall requirements of the application. For instance, to achieve maximum speed performance, one can utilize 27 multipliers and 7 cubing units (operating in $GF(3^m)$) realizing the entire loop iteration in parallel within one clock cycle. On the other extreme, we can take a serial approach by implementing only one multiplier and one cubing unit in hardware. The serial approach will require many more iterations (clock cycles), however, the area is significantly reduced. If the number of multipliers employed to realize the resilient approach is more than half of the number of multipliers employed for the standard implementation, then the resilience approach will not cause any overhead in latency. However, it will cause an area overhead of about 50%.

In software implementations all of the operations mentioned need to be calculated serially. Since the number of operands that needs to be stored in registers is constant, there is no opportunity for trade-off between area and speed. The total (latency) overhead incurred in implementing our approach in software applications is therefore approximately 50%.

Note that while 50% overhead may seem excessive from a performance point of view, the overhead is significantly less than that of comparable techniques, given the level of protection it provides. For instance, the technique proposed in [14] protects an arbitrary data-path with common integer addition and multiplication units (e.g. Montgomery multiplier) with more than 200% overhead. Also, the technique proposed for symmetric cryptosystems like the AES block-cipher [22] requires an area overhead of more than 100 %.

The overhead that is associated with our scheme may seem excessive at first glance, especially when compared to some low-cost linear schemes [31, 32, 13]. It is important, however, to take into account the error model on which a particular method is based. Simple parity-based methods often only detect single bit-errors, but even in more advanced linear schemes the error detection capabilities are limited. Generally speaking, any error pattern that is a valid codeword can never be detected, since a linear combination of codewords results in another codeword. In our error model we assume that the device is under attack by a weak adversary with limited resources. As such, the error detection scheme needs to be resilient enough to detect random codeword error patterns with high probability. Such strong error detection capability comes at a moderate cost due to the non-linear encoding.

Note that, at the current level of protection in the worst case errors are detected with probability $1 - 3^{-m}$. To give a concrete example, a common implementation choice is $m = 93$, where the detection probability becomes lower bounded by $1 - 3^{-93} \approx 1 - 2^{-147}$. In addition, we may decide to sacrifice a little bit from the security level and choose a much smaller number of check digits. For instance, we may choose the parity matrix to map to a subfield of $GF(3^{93})$, e.g. $GF(3^{31})$ which would yield a security level of $1 - 3^{-31} \approx 1 - 2^{-49}$. Such a security level would suffice in practice due to the fact that the circuit would be disabled even after the first fault is detected and the attacker would have to acquire a new unit with the same settings (e.g. a smart card) to continue with the attack. From a performance viewpoint, the operations on the check bits would be implemented over a much smaller field (with 3-times fewer digits), and therefore the complexity of the operations, and hence the overhead, would be reduced significantly. For this, however, the resilient components would have to be redesigned and we leave this along with an actual implementation as future work.

5.2 Lightweight and Robust Error Detection Built Over the Base Field

Based on the original definition of robust codes by Karpovsky and Taubin [18], we derive a slightly modified construction which allows us to accommodate the specific arithmetic needs of the pairing computation, while maintaining robustness properties. Specifically, the original robust codes were defined over $\mathrm{GF}(p)$, with $p > 2$ a prime, while we extend the definition for codes over field extensions $\mathrm{GF}(q)$, where $q = p^m$.

As in the original paper, we use a simple, but appropriate additive error model, i.e. when an error $e = (e_x, e_w)$ is introduced to an operand x, the operand becomes $(x + e_x, w + e_w)$. Thus, if

$$w + e_w \neq (P(x + e_x))^2 \tag{5.5}$$

then our error detection scheme will work and the error detection network will flag an error.

Definition 4 *Let V' be a linear q-ary parity code ($q = p^m$, $p > 2$ is a prime) with $n = k+1$ and check matrix $H = [P|I]$ with rank$(P) = 1$. Then $C_{V'} = \{(f, w)|f \in GF(q^k), w = (Pf)^2 \in GF(q)\}$.*

We capture the performance of this specific non-linear error detection code with the following theorem.

Theorem 3 *A non-zero error $e = (e_x, e_w)$ on a code word $(x, w) \in C_{V'}$ will not be de-*

tected if and only if it satisfies the error masking equation

$$(Px)^2 + e_w = (P(x + e_x))^2. \tag{5.6}$$

For $C_{V'}$ the set of undetectable errors ($\{e|Q(e) = 1\}$) is a $(k-1)$-dimensional subspace of V', $q^k - q^{k-1}$ errors are detected with probability 1, and the remaining $q^{k+1} - q^k$ errors are detected with probability $1 - q^{-1}$.

Proof 2 *From* (5.5)*, we have*

$$2(Px)(Pe_x) + (Pe_x)^2 = e_w \tag{5.7}$$

There are three cases that will satisfy equation (5.7)*:*

1. $Pe_x = e_w = 0$*: For this condition, Equation* (5.7) *is satisfied for all x. Since e_x is in $GF(q^k)$:*

   $$Pe_x = e_x = 0 \tag{5.8}$$

   The number of e_f satisfying Equation (5.8) *is 1. Also, there is only one e_w satisfying $e_w = 0$. Thus, the total number of errors $e = (e_f, e_w)$ satisfying this condition is 1.*

2. $Pe_x = 0$ *and* $e_w \neq 0$*: Equation* (5.7) *is not satisfied for any x. The number of e_x satisfying $Pe_x = 0$ is 1. In addition to this, we need to calculate the number of errors e_w satisfying $e_w \neq 0$. Since $e_w \in GF(q)$, total number of e_w for $e_w \neq 0$ is $q - 1$. Thus, the total number of errors $e = (e_x, e_w)$ satisfying this condition is $q^{k-1} \cdot (q-1) = q^k - q^{k-1}$. In our case, $q^k - q^{k-1} = 3^m - 1$.*

3. $Pe_x \neq 0$*: for any* $e = (e_x, e_w)$ *there exists a unique* Px *satisfying Equation* (5.7)*. Let* x *be randomly selected and let* e_x *be satisfying the condition* $Pe_x \neq 0$ *The number of errors* e_x *satisfying this condition is* $q^k - q^{k-1}$*. The total number of errors* e_w *is* q*. Thus, the total number of errors* $e = (e_x, e_w)$ *is* $(q^k - q^{k-1}) \cdot q = q^{k+1} - q^k$*. The probability that a randomly selected* x *and an error* e_x *such that* $Pe_x \neq 0$ *will not be satisfying Equation* (5.7) *is* $1 - q^{-1}$*. For our case,* $q^{k+1} - q^k = 3^{2m} - 3^m$ *and* $1 - q^{-1} = 1 - 3^{-m}$.

As seen from Algorithm 1, the implementation of efficient arithmetic for $GF(3^{6m})$ and $GF(3^m)$ is essential in Tate Pairing algorithm of Kwon-BGOS. These field operations determine the efficiency and the complexity of the overall Tate Pairing operation. Thus, in order to build an efficient, yet fault-tolerant architecture for Tate Pairing, it is crucial to implement robust arithmetic primitives that are light-weight for $GF(3^m)$ Section 5.2.1 will provide more detail on the performance analysis of our robust field operations in $GF(3^m)$.

5.2.1 Robust $GF(3^m)$ Operations

In order to apply robust arithmetic to all operations of Algorithm 2.2, we extended the approach taken for $GF(3^{6m})$ operations to also cover $GF(3^m)$ operations. To do so, we simply regarded an element in $GF(3^m)$ to be an element in $GF(3^{6m})$ with all of the coefficients being 0, except the coefficient of ζ^0. The parity of an element is simply itself, and the computed checksum is its square.

Cubing

Steps 2, 3, 6 and 7 of Algorithm 2.2 include cubing operations in $GF(3^m)$. The input for this cubing operation is x and the output is $r = x^3$. Application of the robust approach from Section 5.2 yields: $w_x = x^2$ and $w_r = r^2 = (x^3)^2$. Thus, we can extract w_r from w_x by simply taking the cube of w_x. This approach adds an overhead of 1 square and 1 cubing operations in $GF(3^m)$.

Addition

Step 8 of Algorithm 2.2 is addition operation in $GF(3^m)$. The inputs for this operation are x_1 and x_2 and the output is $r = x_1 + x_2$. If we apply our approach we get: $w_{x_1} = x_1^2$, $w_{x_2} = x_2^2$ and $w_r = (x_1 + x_2)^2 = x_1^2 + 2x_1x_2 + x_2^2$. Thus, we can extract w_r from w_{x_1} and w_{x_2} by simply adding these two values to $2x_1x_2$. The overhead for this is 3 squares and 1 multiplication operation in $GF(3^m)$.

Multiplication

Step 9 of Algorithm 2.2 includes multiplication operations in $GF(3^m)$. The inputs for this multiplication operation are y_1 and y_2 and the output is $r = y_1y_2$. This yields: $w_{y_1} = y_1^2$, $w_{y_2} = y_2^2$ and $w_r = (y_1y_2)^2 = y_1^2y_2^2$. Thus, we can extract w_r from w_{y_1} and w_{y_2} by simply multiplying these two values, with an overhead of 3 square and 1 multiplication operations in $GF(3^m)$.

Squaring

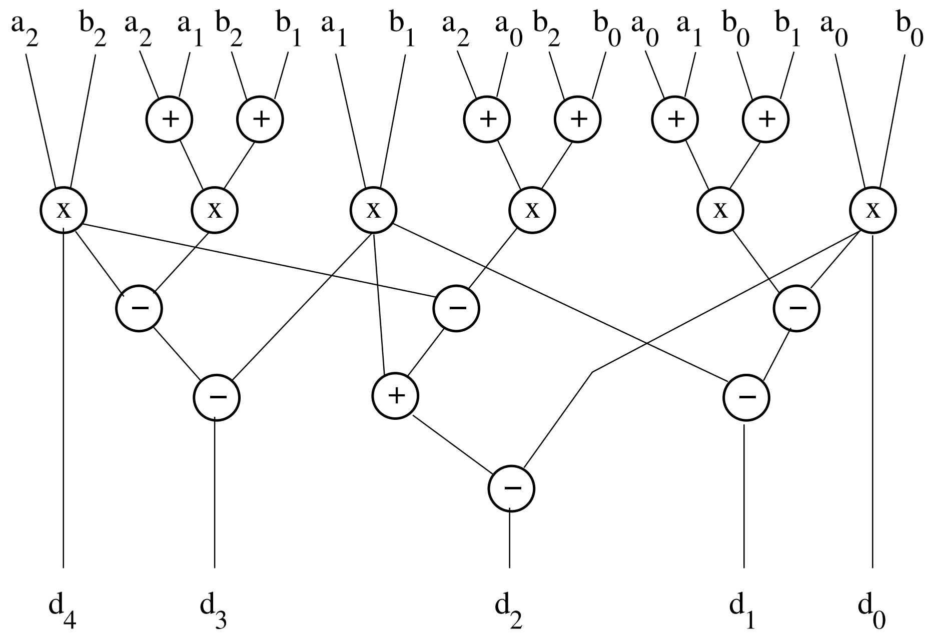
Step 9 of Algorithm 2.2 includes a squaring operation in $GF(3^m)$. The input for this squaring operation is μ and the output is $r = \mu^2$. Here we have for the check-symbols : $w_\mu = \mu^2$ and $w_r = (\mu^2)^2$. Thus, we can extract w_r from w_μ by simply squaring it, which has an overhead of 2 square operations in $GF(3^m)$.

5.2.2 Previous Work

In [19], it is stated that $GF(3^{6m})$ can be considered as an extension field over $GF(3^{2m})$ with irreducible polynomial $z^3 - z \pm 1$. This way, the multiplication in $GF(3^6 m)$ is realized in two steps: i) Karatsuba multiplication for polynomials with coefficients from $GF(3^{2m})$, and ii) reduction with irreducible polynomial $z3 - z \pm 1$. Reader can profitably refer to [19] for further details.

In Figure 5.1, $GF(3^{6m})$ Karatsuba multiplier unit, as proposed in [19], is illustrated, where nodes represent the $GF(3^{2m})$ adders, subtracters, and multipliers. Similarly, $GF(3^2 m)$ is also an extension field over $GF(3^m)$ with irreducible polynomial $y^2 + 1$. Since the adder/subtracter units operate on the corresponding coefficients of the operand polynomials, their structure is the same as $GF(3^m)$ adders. $GF(3^{2m})$ multiplier, however, consists of $GF(3^m)$ adders, subtracters, and multipliers as seen in Figure 5.2.

As seen in Figure 5.1, $GF(3^{6m})$ Karatsuba multiplier has five $GF(3^{2m})$ elements as output. The result of the Karatsuba multiplier has the form $\tilde{d}_4 z^4 + \tilde{d}_3 z^3 + \tilde{d}_2 z^2 + \tilde{d}_1 z + \tilde{d}_0$. Since $z^3 = z + 1$ from the irreducible polynomial, we have $(\tilde{d}_2 + \tilde{d}_4)z^2 + (\tilde{d}_1 + \tilde{d}_4 + \tilde{d}_3)z +$



a_2 b_2 a_2 a_1 b_2 b_1 a_1 b_1 a_2 a_0 b_2 b_0 a_0 a_1 b_0 b_1 a_0 b_0
d_4
d_3
d_2
d_1
d_0

Figure 5.1: $GF(3^{6m})$ multiplier unit from [19]

$(\tilde{d}_0 + \tilde{d}_4)$.

To summarize, 18 $GF(3^m)$ multipliers and 52 $GF(3^m)$ adders are used in one $GF(3^{6m})$ multiplier. For our robust approach, this is extremely high cost. Every adder and multiplier that is included in the circuit needs an extra multiplier. There are a total of 70 units, which adds extra 70 multipliers to the circuit. This is more than 300% increase in the area. Detailed calculations for this will be given in Section 5.2.6.

5.2.3 Our $GF(3^{6m})$ Robust Multiplier Subblock

The first and most expensive $GF(3^{6m})$ block is for performing multiplication operation. This multiplication unit could be implemented as explained in Section 5.1.1. Recall from

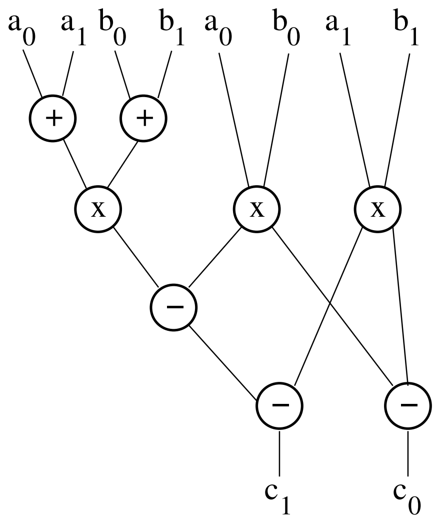


Figure 5.2: $GF(3^{2m})$ multiplier unit from [19]

Section 5.1.1 that

$$
\begin{aligned}
f &= f_0 + f_1 \cdot \sigma + f_2 \cdot \rho + f_3 \cdot \sigma\rho + f_4 \cdot \rho^2 + f_5 \cdot \sigma\rho^2 \\
g &= g_0 + g_1 \cdot \sigma + g_2 \cdot \rho - \rho^2 \;\; (g_3 = g_5 = 0, g_4 = -1) \\
r &= f \cdot g \\
&= (f_0g_0 - f_1g_1 + f_5g_2 - f_2) + \\
&\quad (f_1g_0 - f_0g_1 + f_4g_2 - f_3) \cdot \sigma + \\
&\quad (f_2g_0 - f_3g_1 + f_0g_2 + f_5g_2 - f_2 - f_5) \cdot \rho + \\
&\quad (f_3g_0 - f_2g_1 + f_1g_2 + f_4g_2 - f_3 - f_4) \cdot \sigma\rho + \\
&\quad (f_5g_0 - f_4g_1 + f_2g_2 - f_0 - f_5) \cdot \rho^2 + \\
&\quad (f_4g_0 - f_5g_1 + f_3g_2 - f_1 - f_4) \cdot \sigma\rho^2
\end{aligned}
$$

As stated before, multiplication operation in $GF(3^{6m})$ involves 18 multiplications, 8 additions and 16 subtractions in $GF(3^m)$. It should be noted that the robust approach adds $18 + 24 = 42$ multiplier units to the hardware, which is not illustrated in the figure.

5.2.4 Our $GF(3^{6m})$ Robust Cubing Subblock

The second $GF(3^{6m})$ block is for performing cubing operation and as in the case of the multiplier it is constructed using arithmetic units of the base field $GF(3^m)$. As shown in Figure 5.3, $GF(3^{6m})$ cubing circuitry includes 6 adder/subtracter and 6 cubing blocks in $GF(3^m)$. Recall that

$$\begin{aligned} f &= f_0 + f_1 \cdot \sigma + f_2 \cdot \rho + f_3 \cdot \sigma\rho + f_4 \cdot \rho^2 \\ &\quad + f_5 \cdot \sigma\rho^2 \\ f^3 &= \left(f_0^3 + f_2^3 + f_4^3\right) - \left(f_1^3 + f_3^3 + f_5^3\right) \cdot \sigma + \left(f_2^3 - f_4^3\right) \cdot \rho \\ &\quad + \left(f_5^3 - f_3^3\right) \cdot \sigma\rho + f_4^3 \cdot \rho^2 - f_5^3 \cdot \sigma\rho^2 \end{aligned}$$

Thanks to the efficient $GF(3^m)$ cubing blocks, implementing $GF(3^{6m})$ cubing block with parallel blocks does not consume much area and allows to finish the operation in one clock cycle. As stated before, cubing operation in $GF(3^{6m})$ involves 6 cubings, 4 additions and 2 subtractions in $GF(3^m)$. It should be noted that the robust approach adds 6 multiplier units to the hardware, which is not illustrated in the figure.

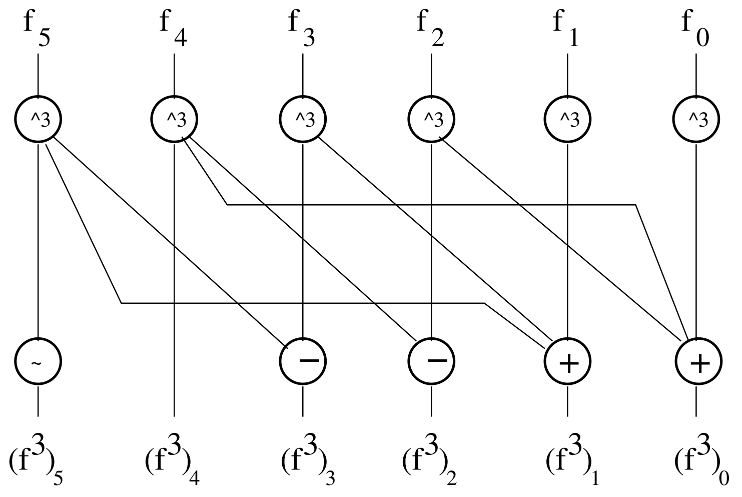


Figure 5.3: Our $GF(3^{6m})$ cubing unit

5.2.5 Our Coprocessor Architecture

After building the efficient blocks that are needed for our accelerator, we design a control unit and a datapath for the Tate Pairing operation. The operation may be divided into two big phases as initialization and loop. In Table 5.4 operations are described in detail.

In the initialization phase, four $GF(3^m)$ elements are input into the accelerator. For this part we use 2m-bit long bus structure and connect it to all four related registers. With address selection and write signals, data are written into the accelerator in four clock cycles. Cubing operations in the steps 3 and 4 also take place during the initialization. Since our cubing block is purely combinational, no extra clock cycles are used at these steps. The length of the databus can be adjusted depending on place-and-route and timing issues.

When the initialization is completed, accelerator starts operating in a loop. Our control unit is composed of mainly two counters. First counter counts the loop's execution

	Step	Operation	clock cycle	total cycle for m = 97
initialization	1	$\alpha = x_p$	1	1
initialization	2	$\beta = y_p$	1	1
initialization	3	$x = x_r^3$	1	1
initialization	4	$y = y_r^3$	1	1
Loop	5	$\alpha = \alpha^3, \beta = \beta^3$	1	97
Loop	6	$\alpha = \alpha^3, \beta = \beta^3$	1	97
Loop	7	$u = \alpha + x + d$	1	97
Loop	8	$\gamma = (-\mu^2)\zeta^0 + (-\beta y)\zeta^1$ $+(-\mu)\zeta^2 + (-1)\zeta^4$	97	97*97
Loop	9	$t = t^3$	1	97
Loop	10	$t = t * \zeta, y = -y, d = d - 1 \bmod 3$	97	97*97

Table 5.4: Explanations for number of clock cycles required for modified Duursma-Lee algorithm

number to end the operation when completed. Second counter determines which step to be executed.

For the entire operation, we use only one $GF(3^{6m})$ multiplier for step 10, one $GF(3^{6m})$ cubing circuit for step 9, two $GF(3^m)$ cubing circuits for steps 5 and 6, two $GF(3^m)$ multipliers for step 8 and a number of adders. Each block starts working according to the counter 2. We also overlap the operations that do not depend on each others' outputs to reduce the number of clock cycles. For instance, in step 10 three operations are done in the same clock cycle. The main advantage of our accelerator is that most of the operations are completed in a single clock cycle. If the adder and cubing circuits were implemented with registers, clock count would increase around by 400 and registers would increase the area of the accelerator.

It should be noted that each multiplier and adder circuits included in our design are implemented with the robust approach. Therefore, each subblock has an additional overhead as explained in Section 5.2.1 and Section 5.2.1.

5.2.6 Implementation Results

The presented architecture was developed into Verilog modules and synthesized using the Synopsys Design Compiler tool. In the synthesis we used the TSMC 0.13 μm ASIC library. The synthesis results for the subblocks for arithmetic in $GF(3^m)$ are shown in Table 5.5

The synthesis results for the subblocks for arithmetic in $GF(3^{6m})$ are shown in Table 5.6.

5.3 Robust Counters

As vividly displayed by the [29] attack, the loop index is perhaps the weakest point in a Tate Pairing implementation. Note, however, that the attack may be easily detected by using the robust encoding introduced by Gaubatz et al. [14]. The proposed technique may be incorporated into the datapath of a hardware implementation or when a software implementation is concerned into the code, or may be implemented directly in the microprocessor architecture.

The technique assumes that operands are represented as k-bit integers and works by computing the check bits $w_x = x^2 \pmod{p}$ where p is picked as a prime integer less than (but close to) 2^k, i.e. the word boundary. Then the error detection performance of the code for a nonzero e pattern is bounded as $\max\{Q(e)\} = 2^{-r}\max\{4, 2^k - p + 1\}$ where r denotes the number of bits used to represent p.

To develop a robust increment unit, we consider the check bits of the incremented counter $x + 1$: $w_{x+1} = (x + 1)^2 \pmod{p} = w_x + 2x + 1 \pmod{p}$. Hence we may build a predictor that takes as input the robust encoded operand (x, w_x) and predicts the check bits for the incremented value by computing only one shift to implement the multiplication by 2 and two additions in $GF(p)$. Note that we also require a modular squaring operation to be performed on the incremented value. Hence, (ignoring the cost for shifting) the total overhead becomes 1 squaring and two additions in $GF(p)$.

	Area(Gates)	Critical Path Delay(CPD)
$GF(3^m)$ Multiplier	7947	3.33 ns
$GF(3^m)$ Adder	1284	0.41 ns
$GF(3^m)$ Cubing	1469	1.06 ns

Table 5.5: Implementation results of $GF(3^m)$ components of the Tate Pairing hardware.

	Non-Robust		Robust		Overhead	
	Area (gates)	CPD (ns)	Area (gates)	CPD (ns)	Area (gates)	CPD (ns)
$GF(3^{6m})$ Multiplier	184948	4.05	541369	4.28	143%	5.68%
$GF(3^{6m})$ Cubing	18302	1.69	129917	4.02	609%	138%
Entire Hardware	241438	5.23	752483	5.92	212%	14%

Table 5.6: Implementation results of $GF(3^{6m})$ components of the Tate Pairing hardware.

Chapter 6

Conclusion

In this dissertation we aimed at developing efficient and tamper-resilient architectures to support pairing computations for Identity-Based cryptography. We presented a scalable and unified architecture to support arithmetic in $GF(2^n)$, $GF(3^m)$, and $GF(p)$. Our design makes use of the redundant signed digit representation, (RSD) which reduces the critical path delay and simplifies the support for characteristic three arithmetic. Previous unified architectures are exclusively designed to implement field multiplication operations and thus carry-save representation they utilized makes it very difficult to perform other operations such as comparison and subtraction. Consequently, classical unified architectures have to transform redundant representation to non-redundant representation to perform these operations. However, these operations benefit from the proposed architecture. For instance, a subtraction operation result in no overhead compared to addition since it can be done by wiring in hardware.

Although there has been a consensus on the benefits of unified architectures, no attempt has been reported in the literature to this date to quantify this benefit. We, for the first time, characterized and compared our unified architecture in terms of the $\{\text{Area} \times \text{CPD}\}$ metric and provided extensive implementation results to give a concretely establish the value of the proposed architecture. We have found out that the proposed unified architecture provides at least 24.88% and 32.07% improvement over non-unified architectures and classical unified architectures, respectively.

Our design is pipelined for improved efficiency and is scalable. Hence, different precisions can be easily supported without the redesign of the core. The number of processing units can be adjusted to given silicon area and/or the desired performance. We believe, this highly versatile architecture will fulfill a critical need in supporting elliptic curve cryptography, RSA/DH schemes, and identity based cryptography using a single architecture in an efficient manner.

Furthermore, we presented a novel scheme for building strong error detection capabilities into Tate pairing computations when realized either in hardware or software. The proposed scheme over the extension field provides quantifiable levels of protection in a weak attacker model. The other proposed scheme built on the base field provides quantifiable levels of protection in a well defined strong attacker model.

We determined the overhead incurred by implementing the robust approach in hardware at various operating points. IThe overhead varies for hardware implementations. When a more serial implementation approach is taken (reducing the arithmetic units and increasing the number of iterations), then the latency overhead is reduced, and eventually almost vanishes, when the number of multipliers employed to realize the robust approach is more than half of the number of multipliers employed for the standard implementation.

Finally, we developed a mathematical framework that allows one to build arithmetic components as well as robust counters to realize looping in the Tate pairing algorithms by Kwon [23] and Barreto et al. [4].

Bibliography

[1] L.-S. Au and N. Burgess, *Unified Radix-4 Multiplier for GF(p) and GF(2^n)*, ASAP, IEEE Computer Society, 2003, pp. 226–236.

[2] A. Avizienis, *Signed-Digit Number Representations for Fast Parallel Arithmetic.*, IRE Trans. Electron. Computers **EC** (1961), no. 10, 389–400.

[3] J.-C. Bajard, L. Imbert, C. Nègre, and T. Plantard, *Efficient Multiplication in GF(p^k) for Elliptic Curve Cryptography*, IEEE Symposium on Computer Arithmetic, IEEE Computer Society, 2003, pp. 181–187.

[4] P. S. L. M. Barreto, S. D. Galbraith, C. O'hEigeartaigh, and M. Scott, *Efficient Pairing Computation on Supersingular Abelian Varieties*, Cryptology ePrint Archive, Report 375/2004, 2004.

[5] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott, *Efficient Algorithms for Pairing-Based Cryptosystems*, Cryptology ePrint Archive, Report 2002/008, 2002.

[6] G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar, and T. Wollinger, *Efficient*

$GF(p^m)$ Arithmetic Architectures for Cryptographic Applications, Topics in Cryptology — CT-RSA 2003 (M. Joye, ed.), vol. LNCS 2612, Springer-Verlag, April 13-17, 2003, pp. 158–175.

[7] G. Bertoni, J. Guajardo, S. S. Kumar, G. Orlando, C. Paar, and T. J. Wollinger, *Efficient GF(p^m) Arithmetic Architectures for Cryptographic Applications*, Topics in Cryptology - CT RSA 2003 (M. Joye, ed.), LNCS, vol. 2612, Springer, 2003, pp. 158–175.

[8] D. Boneh and M. Franklin, *Identity-Based Encryption from the Weil Pairing*, Advances in Cryptology—CRYPTO 2001 (J. Kilian, ed.), LNCS, vol. 2139, Springer-Verlag, 2001, pp. 213–229.

[9] J.-S. Coron, *Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems*, CHES 1999 (Ç. K. Koç and C. Paar, eds.), LNCS, vol. 1717, Springer, 1999, pp. 292–302.

[10] I. Duursma and H. S. Lee, *Tate Pairing Implementation for Hyperelliptic Curves $y^2 = x^p - x + d$*, Advances in Cryptology - Asiacrypt, Lecture Notes in Computer Science, vol. 2894, Springer-Verlag, 2003, pp. 111–123.

[11] I. M. Duursma and H.-S. Lee, *Tate Pairing Implementation for Hyperelliptic Curves $y^2 = x^p - x + d$*, Advances in Cryptology - Asiacrypt 2003 (C.-S. Laih, ed.), LNCS, vol. 2894, Springer, 2003, pp. 111–123.

[12] S. D. Galbraith, K. Harrison, and D. Soldera, *Implementing the Tate Pairing*, Algorithmic Number Theory Symposium (J. Feigenbaum, ed.), Lecture Notes in Computer Science, vol. 2369, Springer, 2002, pp. 324–337.

[13] G. Gaubatz and B. Sunar, *Robust Finite Field Arithmetic for Fault-Tolerant Public-Key Cryptography.*, FDTC'05 (L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, eds.), Lecture Notes in Computer Science, vol. 4236, Springer, 2005, pp. 196–210.

[14] G. Gaubatz, B. Sunar, and M. G. Karpovsky, *Non-linear Residue Codes for Robust Public-Key Arithmetic.*, FDTC (L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, eds.), Lecture Notes in Computer Science, vol. 4236, Springer, 2006, pp. 173–184.

[15] P. Grabher and D. Page, *Hardware Acceleration of the Tate Pairing in Characteristic Three*, Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag LNCS 3659, August 2005, pp. 398–411.

[16] P. Grabher and D. Page, *Hardware Acceleration of the Tate Pairing in Characteristic Three*, CHES (J. R. Rao and B. Sunar, eds.), Lecture Notes in Computer Science, vol. 3659, Springer, 2005, pp. 398–411.

[17] J. Großschädl, *A Bit-Serial Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^m)*, CHES 2001 (Ç. K. Koç, D. Naccache, and C. Paar, eds.), LNCS, vol. 2162, Springer, 2001, pp. 202–219.

[18] M. G. Karpovsky and A. Taubin, *New Class of Nonlinear Systematic Error Detecting Codes.*, IEEE Transactions on Information Theory **50** (2004), no. 8, 1818–1820.

[19] T. Kerins, W. P. Marnane, E. M. Popovici, and P. S. L. M. Barreto, *Efficient Hardware for the Tate Pairing Calculation in Characteristic Three*, CHES 2005 (J. R. Rao and B. Sunar, eds.), LNCS, vol. 3659, Springer, 2005, pp. 412–426.

[20] T. Kerins, E. Popovici, and W. P. Marnane, *Algorithms and Architectures for Use in FPGA Implementations of Identity Based Encryption Schemes*, Field Programmable Logic and Applications, LNCS, vol. 3203, Springer-Verlag, 2004, pp. 74–83.

[21] Ç. K. Koç and T. Acar, *Montgomery Multiplication in GF*(2^k), Proceedings of Third Annual Workshop on Selected Areas in Cryptography (Queen's University, Kingston, Ontario, Canada), August 15–16 1996, pp. 95–106.

[22] K. Kulikowski, M. G. Karpovsky, and A. Taubin, *Robust Codes for Fault Attack Resistant Cryptographic Hardware*, FDTC'05 (L. Breveglieri and I. Koren, eds.), Sep 2005.

[23] S. Kwon, *Efficient Tate Pairing Computation for Supersingular Elliptic Curves over Binary Fields*, ACISP (C. Boyd and J. M. Gonzáles Nieto, eds.), Lecture Notes in Computer Science, vol. 3574, Springer-Verlag, July 2005, pp. 134–145.

[24] P. L. Montgomery, *Modular Multiplication without Trial Division.*, Mathematics of Computation **44** (1985), no. 170, 519–521.

[25] E. Oswald, S. Mangard, and N. Pramstaller, *Secure and Efficient Masking of AES- A Mission Impossible*, Tech. report, Technical Report IAIK-TR 2003/11/1, 2004, http://eprint.iacr.org/.

[26] E. Ozturk, B. Sunar, and E. Savas, *A Versatile Montgomery Multiplier Architecture with Characteristic Three Support*, Computers and Electrical Engineering (2008).

[27] Erdinç Öztürk, Gunnar Gaubatz, and Berk Sunar, *Tate Pairing with Strong Fault Resiliency*, FDTC, 2007, pp. 103–111.

[28] D. Page and N. P. Smart, *Hardware Implementation of Finite Fields of Characteristic Three*, Cryptographic Hardware and Embedded Sytems — CHES 2002 (B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, eds.), LNCS, vol. 2523, Springer-Verlag Berlin, 2002, pp. 529–539.

[29] D. Page and F. Vercauteren, *A Fault Attack on Pairing Based Cryptography*, IEEE Transactions on Computers **55** (2006), no. 9, 1075–1080.

[30] F. Regazzoni, S. Badel, T. Eisenbarth, J. Grobschadl, A. Poschmann, Z. Toprak, M. Macchetti, L. Pozzi, C. Paar, Y. Leblebici, et al., *A Simulation-Based Methodology for Evaluating the DPA-Resistance of Cryptographic Functional Units with Application to CMOS and MCML Technologies*, International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2007 (2007), 209–214.

[31] A. Reyhani-Masoleh and M.A. Hasan, *Error Detection in Polynomial Basis Multipliers over Binary Extension Fields*, CHES 2002 (Heidelberg) (B. S. Kaliski, Jr., Ç. K. Koç, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2523, Springer, 2002, 4th International Workshop, Redwood Shores, CA, USA, pp. 515–528.

[32] Arash Reyhani-Masoleh and M. Anwar Hasan, *Towards Fault-Tolerant Cryptographic Computations over Finite Fields*, ACM Transactions on Embedded Computing Systems **3** (2004), no. 3, 593–613.

[33] A. Satoh and K. Takano, *A Scalable Dual-Field Elliptic Curve Cryptographic Processor*, IEEE Transactions on Computers **52** (2003), no. 4, 449–460.

[34] E. Savaş, A. F. Tenca, and Ç. K. Koç, *A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$*, Cryptographic Hardware and Embedded Sytems — CHES 2000 (Ç. K. Koç and C. Paar, eds.), LNCS, vol. 1965, Springer-Verlag, 2000, pp. 277–292.

[35] E. Savaş, A. F. Tenca, M. E. Çifçibaşi, and Ç. K. Koç, *Multiplier Architectures for GF(p) and GF(2^n)*, IEE Proceedings Computers and Digital Techniques **151** (2004), no. 2, 147–160.

[36] A. Shamir, *Identity-Based Cryptosystems and Signature Schemes*, Advances in Cryptology - CRYPTO'84 (New York, NY, USA) (G. R. Blakley and D. Chaum, eds.), Lecture Notes in Computer Science, vol. 196, Springer-Verlag, 1985, pp. 47–53.

[37] A. F. Tenca and Ç. K. Koç, *A Scalable Architecture for Montgomery Multiplication.*, Cryptographic Hardware and Embedded Sytems - CHES 1999 (Ç. K. Koç and C. Paar, eds.), volume 1717 of LNCS, Springer, Berlin, Germany, 1999, pp. 94–108.

[38] A. F. Tenca, E. Savaş, and Ç. K. Koç, *A Design Framework for Scalable and Unified Multipliers in GF(p) and GF(2^m)*, International Journal of Computer Research **13** (2004), no. 1, 68–83.

[39] K. Tiri, M. Akmal, and I. Verbauwhede, *A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards*, Proceedings of the 28th IEEE European Solid-State Circuits Conference, ESSCIRC 2002 (2002), 403–406.

[40] Z. Toprak and Y. Leblebici, *Low-Power Current Mode Logic for Improved DPA-Resistance in Embedded Systems*, IEEE International Symposium on Circuits and Systems, ISCAS 2005 (2005), 1059–1062.

[41] J. H. van Lint, *Introduction to Coding Theory*, second ed., Graduate Texts in Mathematics, vol. 86, Springer-Verlag, Berlin, 1992.

[42] C. D. Walter and D. Samyde, *Data Dependent Power Use in Multipliers*, ARITH '05: Proceedings of the 17th IEEE Symposium on Computer Arithmetic (Washington, DC, USA), IEEE Computer Society, 2005, pp. 4–12.

[43] J. Wolkerstorfer, *Dual-Field Arithmetic Unit for GF(p) and GF(2^m)*, Cryptographic

Hardware and Embedded Sytems - CHES 2002 (B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, eds.), Volume 2523 of LNCS, Springer, Berlin, Germany, 2002, pp. 500–514.