Multiplication in Binary Fields on Modern Computers and its Applications

Ming-Shing Chen

February 20, 2020

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

This thesis presents the implementations of the arithmetic of finite field, especially the multiplication, on modern computers and shows various applications employing the techniques. The implementation of arithmetics in finite fields is an essential topic in the areas of code and cryptography. We can always find different implementations for the same field concerning different circumstances or computation platforms. Even in the same software, it is common to use different representations for the same field in different components for optimizing the performance. The development of software follows the same philosophy on developing high-speed cryptography in [Sch11] – designing and implementing software with careful choices of high-level parameters, low-level optimization of software on a given microarchitecture, and considerations of their subtle interactions.

In the first part of the thesis, we review and discuss the implementations of the finite fields of characteristic 2, i.e., the binary fields. We will first review the constructions of binary fields mainly on the polynomial constructions and the construction of vector space [Can89]. We then introduce the useful instructions on modern CPUs, especially on SIMD instruction sets, for implementing the multiplications in binary fields. After the preliminaries, we present the practical implementations of field multiplications based on the different size of fields or circumstances of the multiplications.

In the second part of this thesis, we show four different applications for a particular technique of multiplications. The applications are listed as follows:

1. Encoding and Decoding of the erasure correcting codes in the RAID system: it corresponds to multiplications in small field with SIMD instruction set [CCC+09, Anv11, CYC13, PGM13, GRU14].
2. Developing multivariate public key cryptosystems: it requires technique on multiplication in the cryptographic softwares [CLP+18].
3. Evaluating univariate polynomials at multi-points with additive FFTs: it corresponds to multiplication by subfield elements [CCK+17].
4. Multiplying Boolean polynomials: it corresponds to multiplication in (slightly) large field with carryless multiplications [GK14, LK16, CLP+18, LCK+18].

Since we focus on the implementations of these applications in this thesis, we present the materials based on engineering's perspective instead of the discussion of computational complexities based on bit-operations in the theoretical works. In other words, we work on real platforms equipped with powerful instruction sets, e.g., for multiplying short boolean polynomials or working in SIMD manner. We will describe the details resulting in high-performance software on real CPUs. These computation platforms and instruction sets contribute to different strategies on implementing the multiplications or the design of software, although they usually do not change the level of complexity.

1.2 The Applications

1.2.1 Codes for RAID

The current mainstream RAID(Redundant Array of Independent Disks, a technology for massive data storage) will not be capable for recovering lost data from the same amount of redundant data when the size of disk array becomes larger due to its erasure correcting code(ECC). For example, with the current ECC, we can recover any kind of lost data for 3 data disks when we set up 3 checksum disks in a disk array of 255 total disks. When we set up with 4 checksum disks, however, we will not recover some lost data for 4 data disks from the 4 checksum disks even if we run a disk array of only 28 total disks.

Therefore, based on the current ECC, we present an erasure correcting code, named RAIDq, to *partly* fix the issue. The RAIDq can recover any kind of lost data for 4 data disks from 4 checksums for a disk array of 96 total disks. It is designed with backward compatibility to the current RAID in both algorithm and redundant data and thus restricts its ability for data recovery. More importantly, the RAIDq is emphasized with its high performance which is arguably one of the strong reason for current RAID technology. The performance of the RAIDq depends on minimizing the arithmetic operations for generating checksums and exploiting high-performance instructions on modern CPUs.

Implementation with SIMD instructions is the most basic technique for high-performance software, especially for the arithmetic on binary fields. Through the application, we show the SIMD arithmetic technique facilitates the implementation of the erasure correcting code for RAIDs.

This chapter of code is based on the joint work with Bo-Yin Yang and Chen-Mou Cheng published in [CYC13].

1.2.2 MPKCs

Multivariate Public Key Cryptosystems(MPKCs), which are often touted as future-proof cryptosystems against Quantum Computers [DY08]. It is usually advertised with its high-performance implementations [CCC+09]. In this chapter, we review the Rainbow/TTS [DS05,DYC+08], a derivative of MPKC signatures, and its implementations. With the powerful instruction sets on modern CPUs, we show the techniques of high-performance implementation on central components of MPKCs including evaluating multivariate quadratic polynomials and solving linear equations with Gauss eliminations.

In practice, a security system can be broken because of its implementation instead of the cryptography. A famous example is that some AES implementations were attacked due to the leakage on side channel information [BM06]. The side channel resilience is an essential requirement for cryptographic software. In other words, the secret data should be independent of memory access, and the program should maintain time constancy when processing secret data in a cryptographic software. Hence, the implementations of arithmetic for general multiplications may not be suitable in the cryptographic world.

Through the application of MPKCs, we show the implementations of arithmetic in fields under the cryptographic requirements. We implement the constant-time field multiplications with SIMD instructions for high-performance software and demonstrate the different strategies of constant-time implementations for various components in the MPKCs.

This chapter is base on the joint work with Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang and Chen-Mou Cheng published in [CLP+18].

1.2.3 The Additive FFT and its Implementations

A fast Fourier transformation(FFT) is an algorithm that evaluates a polynomial at a set of particular points, which is a handy tool in many areas of computer science. The additive FFT evaluates polynomials at an additive subgroup instead of multiplicative subgroups in usual FFTs. It was developed by Cantor [Can89], Gao and Mateer [GM10], Lin, Chung, and Han [LCH14],and Lin, Al-Naffouri, and Han [LANH16]. We discuss its variant for binary fields, specifically, with respect to Cantor basis. In the variant of additive FFTs, the main algorithm is generally divided into two main subroutines which are a basis conversion (for polynomials) and one butterfly network.

For the high-performance implementation of the additive FFT, we use the techniques on multiplying subfield elements to accelerate the field multiplications in the butterfly network. Although the basis conversion runs in higher complexity level, the butterfly network consumes the most computational power in practice and can be optimized by the subfield multiplications. Besides the field multiplications, we also describe the fast calculation of constants in the butterflies and the better memory access model for implementing the algorithm.

The materials about the implementations are based on joint work with Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. The preprint version can be found in [CCK+17].

1.2.4 Multiplying Boolean Polynomials

The last application in chapter 7 presents a method for multiplying boolean polynomials with the best-known performance. For this fundamental problem in computer science, it has been many types of researches, e.g., [BGTZ08, HvdHL16, CCK+17, vdHLL17] focusing on fast implementations, based on various FFTs, on modern CPUs. For practitioners, it was surprising that van der Hoeven *et al.* could still get a new multiplier in 2017 with a two times improvement over their previous implementation [HvdHL16]. The new multiplier multiplies polynomials with a new Frobenius FFT [vdHL17] instead of the usual Kronecker substitution [GG13, Chap. 8]. In this chapter, we show how to cooperate the technique of Frobenius FFT with the additive FFT and result in a

new algorithm for multiplying Boolean polynomials.

We also present the implementation of the new multiplier and compare the performance with other previous softwares. In our implementation, we perform the field multiplications with the `PCLMULQDQ` instruction, which is a hardware instruction for multiplying 64-bits Boolean polynomials, since we target on the multiplications in 64-bits or 128-bits fields The other techniques include the truncated additive FFT and perform the truncated FFT with linear transformations. We note at last that the application itself can be the critical component for implementing the multiplication in further large fields.

This chapter is based on the joint work with Wen-Ding Li, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang published in [LCK+18].

1.3 Outline of the Thesis

The thesis comprises topics for the multiplications in binary fields and four chapters for their applications.

After this introduction, Chapter 2 introduces the notations, defines the constructions of fields used in the thesis, and reviews the useful instructions on modern computers. Chapter 3 then presents various implementations of field multiplication for different sizes of fields and requirements.

The rest of the thesis is composed of 4 relatively independent topics. Chapter 4 describes a code for extending the disks in RAID. It corresponds to the general multiplications for small fields. Chapter 5 describes the implementations of MPKCs. It corresponds to the constant-time multiplications. In chapter 6, we introduce the algorithm of additive FFT and describe its implementations. The chapter is the background knowledge for multiplying Boolean polynomials. Chapter 7 describes the algorithm for multiplying Boolean polynomials. The multiplier presented in the chapter can be the critical component of the multiplication in larger binary fields.

Chapter 2

Preliminaries

2.1 Notations

We follow the notations in [GG13] and summarize the usual conventions in this thesis as follows:

- Set, Ring, or Field. A set is notated as the capital letter, e.g., R or F. Some particular sets are $\mathbb{N}$ for natural numbers, $\mathbb{Z}$ for integers, and $\mathbb{F}$ for finite fields.
- Vectors. $(v_i)_{0 \le i \le 7} = (v_0, \ldots, v_7)$, or a bold small letter $\mathbf{v}$.
- Polynomials. We use a small italic symbol, e.g., f, g, and h to represent a polynomial, and the symbol $f(x)$ indicates a polynomial with the indeterminate x. The subscripts of a polynomial indicate the coefficients, e.g., $f(x) = f_0 + f_1 x + f_2 x^2 + \cdots$. We also present a polynomial $f(x)$ in the vector form $(f_0, f_1, \ldots)$.
- Matrices. Matrices are presented in capital letters, e.g., A, B, etc.
- Numbers in binary expansion or boolean vectors is presented in typewriter fonts, e.g., `0xff`. It usually represents the data formatted on computers.
- An algorithm or a procedure on computers is presented in typewriter fonts, e.g., `Encode`.

2.2 The Construction of Binary Fields

In this section, we discuss the constructions and arithmetic of fields of algebraic extensions of $\mathbb{F}_2$, or binary fields. The finite field(or Galois field, GF) of two elements, denoted as $\mathbb{F}_2$, is the set $\{0, 1\}$ with multiplication and addition. The multiplication of $\mathbb{F}_2$ is logic `AND` and addition is logic `XOR`.

How the field element is represented affects the procedure of performing the arithmetics. In this thesis, we will discuss how to perform the arithmetics under particular representations of binary fields for different applications. We will apply interchangeable representations of fields and switch representations for better efficiency of field multiplication. All fields of the same size are isomorphic,

and the cost of changing representations is to perform the isomorphism(a linear transformation).

We refer the readers to [GG13, Chapter 25] [LN86, chapter 1,2] for the fundamental concept of algebra.

2.2.1 The Polynomial Constructions

In this section, we discuss the representations of binary fields in the polynomial form as well as the method for performing arithmetics for the representation.

Polynomials in $\mathbb{F}_2[x]$

A polynomial over $\mathbb{F}_2$(or a boolean polynomial) is an expression of this form

$$f(x) = f_0 + \cdots + f_d x^d = \sum_{i=0}^{d} f_i x^i \quad \text{,where } f_i \in \mathbb{F}_2 \text{ for } 0 \leq i \leq d \ .$$

Here the indeterminate x is just a placeholder. $\mathbb{F}_2[x]$ denotes the ring of univariate polynomials over $\mathbb{F}_2$ with the indeterminate x. We also represent a polynomial $f(x)$ as a vector $f = (f_0, f_1, \ldots)$ with entries from $\mathbb{F}_2$ and a finite number of coefficients f_i's are nonzero. Since a natural basis for $\mathbb{F}_2[x]$ is $(1, x, x^2, \ldots)$, we can associate the basis elements to the boolean vectors with one nonzero entry, i.e., $1 = (1, 0, \ldots), x = (0, 1, \ldots), x^2 = (0, 0, 1, \ldots), \ldots$ etc.

Clearly, we can use the integer i to denote a boolean vector $(i_0, i_1, \ldots, i_d)$ if i is a nonnegative integer whose binary expansion is $i = i_d i_{d-1} \cdots i_0$ or

$$i = \sum_{j=0}^{d} i_j \cdot 2^j \quad \text{where } i_j \in \{0, 1\} \ . \tag{2.1}$$

We can then define the boolean polynomial corresponding to the binary expansion of i

$$\omega(i) = \sum_{j=0}^{d} i_j \cdot x^j \ . \tag{2.2}$$

Conversely, we have $\omega^{-1}(x^j) = 2^j$ with respect to the basis $(x^j)_{j=0,1,\ldots}$ of $\mathbb{F}_2[x]$. Hence, we can store the polynomials of degree d in $\mathbb{F}_2[x]$ as Boolean vectors of length $(d+1)$-bits.

We also adopt a convention that a hex number denotes explicitly the value of a boolean vector. For example, the polynomial $x^8 + x^4 + x^3 + x + 1 \in \mathbb{F}_2[x]$ is represented as the Boolean vector $\texttt{0x11b} = 1, 0001, 1011_b = 2^8 + 2^4 + 2^3 + 2 + 1$, i.e., $\omega(\texttt{0x11b}) = x^8 + x^4 + x^3 + x + 1$.

The addition and multiplication of polynomials are defined by $(a_0, a_1, \ldots) + (b_0, b_1, \ldots) = (a_0 + b_0, a_1 + b_1, \ldots)$ and $(a_0, a_1, \ldots) \cdot (b_0, b_1, \ldots) = (c_0, c_1, \ldots)$, with $c_n = \sum_{i=0}^{n} a_i b_{n-i}$. The polynomials in $\mathbb{F}_2[x]$ are the basic form of algebraic objects in this thesis since there are hardware instructions for multiplying Boolean polynomials of some fixed lengths.

Binary Fields of Boolean Polynomials

An element in the binary field $\mathbb{F}_{2^m}$ can be represented as a polynomial in $\mathbb{F}_2[x]_{<m}$ where the subscript denotes the restriction of its degree. The basic construction of $\mathbb{F}_{2^m}$ is the ideal of $\mathbb{F}_2[x]$ modulo(mod) an irreducible polynomial of degree m. Just as a Boolean polynomial can be represented as a Boolean vector, so too the field $\mathbb{F}_{2^m} := \mathbb{F}_2[x]_{<m}$ can be represented as the vector space $\mathbb{F}_2^m$. The "vectors" $\{1, x, \ldots, x^{m-1}\}$ form a basis for $\mathbb{F}_2[x]_{<m}$. Hence, with respect to the basis $(1, x, \ldots, x^{m-1})$ of $\mathbb{F}_{2^m}$, let i be a nonnegative integer with binary expansion $i = \sum_{j=0}^{d} i_j \cdot 2^j$ as in Eq. 2.1. The field element corresponding to the binary expansion i is

$$\omega_{\mathbb{F}_{2^m}}(i) = \sum_{j=0}^{m-1} i_j \cdot x^j \ . \tag{2.3}$$

Explicit Constructions of particular fields Probably due to its one-byte storage, the field of 256 elements($\mathbb{F}_{256}$) plays an important role on computers, especially in the area of cryptography and code. In Chapter 4, we use the construction of $\mathbb{F}_{256}$ as

$$\mathbb{F}_{256} = \mathbb{F}_{2^8} := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x^2 + 1) \ . \tag{2.4}$$

The construction consists with the field used in the erasure correcting code of standard RAID-6 [Anv11]. Note that it differs from the one in Advanced Encryption Standard(AES) [Nat01], which constructs its field with respect to a different irreducible polynomial

$$\mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1) \ .$$

We will use the term $\texttt{0x11b}\,\mathbb{F}_{256}$ when referring to the field of 256 elements in AES.

In Chapter 7, we use the fields of 64 and 128-bits. The field of 64-bits is constructed as

$$\mathbb{F}_{2^{64}} := \mathbb{F}_2[x]/\left(x^{64} + x^4 + x^3 + x + 1\right) \ . \tag{2.5}$$

For the field of 128-bits, we choose the same construction as in AES-GCM:

$$\mathbb{F}_{2^{128}} := \mathbb{F}_2[x]/\left(x^{128} + x^7 + x^2 + x + 1\right) \ . \tag{2.6}$$

The Multiplication of Field Elements We perform the multiplication in $\mathbb{F}_{2^m}$ in two steps. The first step multiplies two corresponding polynomials in $\mathbb{F}_2[x]$. The second step reduces the degree of the resulted polynomial in the first step by a modulo of the irreducible polynomial of degree m.

We show an example of multiplications in $\mathbb{F}_4 := \mathbb{F}_2[x]/(x^2 + x + 1)$. Given $a, b \in \mathbb{F}_4$ and their polynomial forms $a(x) = a_0 + a_1 x$ and $b(x) = b_0 + b_1 x$. To compute $c = a \cdot b$, the first step calculates

$$(a_0 + a_1 x) \cdot (b_0 + b_1 x) = a_0 \cdot b_0 + (a_0 \cdot b_1 + a_1 \cdot b_0)x + (a_1 \cdot b_1)x^2 = c_0 + c_1 x + c_2 x^2 \ . \tag{2.7}$$

Here we have $c_0 = a_0 \cdot b_0$, $c_1 = a_0 \cdot b_1 + a_1 \cdot b_0$, and $c_2 = a_1 \cdot b_1$. In the second step, since

$$x^2 \equiv x + 1 \mod x^2 + x + 1 \ , \tag{2.8}$$

the term $c_2 \cdot x^2$ reduces to $c_2 x + c_2$. We have the result

$$c = (c_0 + c_2) + (c_1 + c_2)x = (a_0 \cdot b_0 + a_1 \cdot b_1) + (a_0 \cdot b_1 + a_1 \cdot b_0 + a_1 \cdot b_1)x \ . \tag{2.9}$$

It costs 4 `AND` and 3 `XOR` in total.

To perform the multiplication on computers, the straightforward method is to perform all the bit-operations. One can first isolate all coefficients of the two input polynomials and perform all the arithmetics in $\mathbb{F}_2$. We can also perform the multiplication relying on the hardware instructions on modern computers. We will discuss the method in Sec. 3.1 and 3.4.

Although we do not discuss the calculation in this thesis, a multiplicative inverse may be obtained via the Euclidean algorithm or Fermat's little theorem.

Polynomials over $\mathbb{F}_{2^m}$

Besides representing the field elements as boolean polynomials, we can also use the polynomials over $\mathbb{F}_{2^m}$(i.e., $\mathbb{F}_{2^m}[x]$) to represent elements in the extension field of $\mathbb{F}_2$. More precisely, given a field $\mathbb{F}_{2^m}$ and an irreducible polynomial $f(x) \in \mathbb{F}_{2^m}[x]$ with $\deg f(x) = n$, we can construction the extension field $\mathbb{F}_{2^{m \cdot n}} := \mathbb{F}_{2^m}[x]/f(x)$. $\mathbb{F}_{2^{m \cdot n}}$ is isomorphic to $\mathbb{F}_{2^m}^n$ as $\mathbb{F}_{2^m} \cong \mathbb{F}_2^m$.

In Chapter 4, we use polynomials in $\mathbb{F}_{256}[X]_{\leq 1}$ to represent elements in $\mathbb{F}_{256^2}$ and construct the field as

$$\widetilde{\mathbb{F}}_{256^2} := \mathbb{F}_{256}[X]/(X^2 + \omega_{\mathbb{F}_{256}}(\texttt{0x8}) \cdot X + 1) \ . \tag{2.10}$$

Note that the symbol $\widetilde{\mathbb{F}}_{256^2}$ associates to the specific construction in this thesis. In the construction, an element of $\widetilde{\mathbb{F}}_{256^2}$ can also be represented as a vector of two components in $\mathbb{F}_{256}^2$. Since the constant term of polynomials in $\mathbb{F}_{256}[X]$ is the same as $\mathbb{F}_{256}$, an element in $\mathbb{F}_{256}$ can be represented as a constant polynomial with the same value for the constant coefficient. Hence, we say the representation of $\widetilde{\mathbb{F}}_{256^2}$ is compatible with $\mathbb{F}_{256}$.

In [AJ86], Adleman and Lenstra presented an algorithm for constructing the following tower of fields:

$$\begin{aligned} \mathbb{F}_4 &:= \mathbb{F}_2[x_0]/(x_0^2 + x_0 + 1), \\ \widetilde{\mathbb{F}}_{16} &:= \mathbb{F}_4[x_1]/(x_1^2 + x_1 + x_0), \\ \widetilde{\mathbb{F}}_{256} &:= \widetilde{\mathbb{F}}_{16}[x_2]/(x_2^2 + x_2 + x_1 x_0), \\ \widetilde{\mathbb{F}}_{65536} &:= \widetilde{\mathbb{F}}_{256}[x_3]/(x_3^2 + x_3 + x_2 x_1 x_0), \\ &\vdots \end{aligned} \tag{2.11}$$

In these constructions, an element in $\widetilde{\mathbb{F}}_{65536}$ is represented as a first degree polynomial over $\widetilde{\mathbb{F}}_{256}$, an element in $\widetilde{\mathbb{F}}_{256}$ is represented as a first degree polynomial over $\widetilde{\mathbb{F}}_{16}$, and so on. We can eventually decompose an element a of binary fields to a multivariate polynomial over $\mathbb{F}_2$, i.e., $a \in \mathbb{F}_2[x_0, x_1, x_2, \ldots]$.

To construct the vector form of elements in the tower fields, given m is a power of 2, i.e., $m = 2^{l_m}$, we can associate a binary expansion of $k = \sum_{j=0}^{l_m - 1} k_j \cdot 2^j$ with the monomial

$$\text{monomial}(k) = x_0^{k_0} x_1^{k_1} \cdots x_{l_m - 1}^{k_{l_m - 1}} \quad \text{for } 0 \leq k < m \ . \tag{2.12}$$

The basis for $\widetilde{\mathbb{F}}_{2^m}$ is

$$(\text{monomial}(k))_{0\le k<m} = \left(\omega_{\mathbb{F}_2[x_0,x_1,\ldots,x_{l_m-1}]}(2^k)\right)_{0\le k<m} \ . \tag{2.13}$$

For example, we have the basis for the vector space $\widetilde{\mathbb{F}}_{256}$:

$$(1, x_0, x_1, x_0x_1, x_2, x_0x_2, x_1x_2, x_0x_1x_2) = (\text{monomial}(k))_{k=0,1,\ldots,7} \ .$$

As the second example, we illustrate the polynomial form of $\omega_{\mathbb{F}_2[x_0,x_1,\ldots]}(\texttt{0xf}) \in \widetilde{\mathbb{F}}_{16}$

$$\omega_{\mathbb{F}_2[x_0,x_1,\ldots]}(\texttt{0xf}) = 1 + x_0 + x_1 + x_0x_1 = (1 + x_0) + (1 + x_0)x_1 \ .$$

Note that $(1+x_0)$ is an element in $\mathbb{F}_4$. Hence, the tower fields are also compatible with their sub-fields.

The multiplication Under the representation of polynomials over $\mathbb{F}_{2^m}$, the field multiplication is similar to polynomial multiplication with reduction. However, the coefficient multiplication is in $\mathbb{F}_{2^m}$ instead of $\mathbb{F}_2$.

The multiplication in tower fields is more complicated. Since the field is constructed as $\widetilde{\mathbb{F}}_{2^m} := \widetilde{\mathbb{F}}_{2^{m/2}}[y]/y^2 + \cdots$, the field multiplication comprises a polynomial multiplication and a reduction for the term of degree 2. However, we have to perform the coefficient multiplication in $\widetilde{\mathbb{F}}_{2^{m/2}}$ in the same way of multiplication in $\widetilde{\mathbb{F}}_{2^m}$. We have to apply the rules recursively down to polynomials of $\mathbb{F}_2$ coefficients.

2.2.2 Finite Field as Linear Space

While describing the fields as vector spaces, in [Can89], Cantor assumed the fields are chosen so that

$$\mathbb{F}_2 = \mathbb{F}_{2^{2^0}} \subset \mathbb{F}_{2^{2^2}} \subset \mathbb{F}_{2^{2^3}} \subset \cdots$$

and let $\widetilde{\mathbb{F}} = \bigcup_k \mathbb{F}_{2^{2^k}}$ be the union of all these fields. Cantor described a construction of finite field for $\mathbb{F}_{2^m}$, where m is a power of 2, from the kernel of a linear transformation

$$S : \left\{ \begin{array}{ccc} \widetilde{\mathbb{F}} & \to & \widetilde{\mathbb{F}} \\ x & \mapsto & x^2 - x \ . \end{array} \right. \tag{2.14}$$

S can be written in the polynomial form as

$$s_1(x) = x^2 - x = \prod_{a\in\mathbb{F}_2} (x - a) \ . \tag{2.15}$$

Define the *subspace polynomials* $s_0(x) = x$, $s_1(x) = x^2 - x$, and inductively

$$s_{i+1}(x) = s_1(s_i(x)) \ , \quad i = 1, 2, \ldots \tag{2.16}$$

Thus, if $x \in \widetilde{\mathbb{F}}$, then $S^i x := s_i(x)$. Clearly, $\deg s_i(x) = 2^i$.

Cantor define the vector space W_i over $\mathbb{F}_2$ as the kernel space of S^i, i.e.,

$$W_0 = \{0\}, \quad W_i = \{x \in \widetilde{\mathbb{F}} | S^i x = 0\} = \{x \in \widetilde{\mathbb{F}} | s_i(x) = 0\} \ . \tag{2.17}$$

Then W_i is a subspace of $\widetilde{\mathbb{F}}$ over $\mathbb{F}_2$. Under this definition, $\dim_{\mathbb{F}_2} W_i = i$ and $W_m = \mathbb{F}_{2^m}$ if m is a power of 2.

For the basis of W_m, Cantor defined that

Definition 1. Let a sequence $u_0, u_1, u_2, \ldots$ of elements from the algebraic closure of $\mathbb{F}_2$ satisfy

$$u_j^2 + u_j = (u_0 u_1 \cdots u_{j-1}) + [\text{a sum of monomials of lower degrees}],$$

where "a sum of monomials of a lower degree" means

$$\sum_i c_i u_0^{i_0} u_1^{i_1} \cdots u_{j-1}^{i_{j-1}},$$

where $c_i \in \mathbb{F}_2$ and $(i_0, i_1, \ldots, i_{j-1})$ is the binary expansion of $i = \sum_{k=0}^{j-1} i_k \cdot 2^k$. Then, we can define the basis

$$y_i = u_0^{i_0} u_1^{i_1} \cdots u_j^{i_j}$$

so that $y_{2^r} = u_r$ for $r = 0, 1, 2, \ldots$.

Cantor then proved

Theorem 1. *The sequence $(y_0, y_1, y_2, \ldots)$ is a basis for $\widetilde{\mathbb{F}}$. $(y_0, \ldots, y_{m-1}) = (y_i)_{0 \le i < m}$ is a basis for W_m and $y_m \in W_{m+1} \setminus W_m$.*

The Basis Compatible with Polynomial Construction

We provide an explicit choice of elements $(u_0, u_1, \ldots)$. If we choose

$$Su_k = (u_0 u_1 \cdots u_{k-1}) \quad \text{for } k = 1, 2, 3, \ldots \ , \tag{2.18}$$

we have

$$\begin{aligned} u_0^2 + u_0 &= 1 \\ u_1^2 + u_1 &= u_0 \\ u_2^2 + u_2 &= u_0 u_1 \\ u_3^2 + u_3 &= u_0 u_1 u_2 \\ &\vdots \end{aligned}$$

and the construction of fields $\{W_1, W_2, W_4, W_8, \ldots\}$ are exactly the same as Eq. 2.11. In this case, we can associate the symbol

$$u_i = x_i$$

and the basis element

$$y_i = \text{monomial}(i) \ .$$

Hence, we can use the same value for the same element in the two representations of the tower fields(Eq. 2.13) and the particular type of Cantor's construction(i.e., Def. 1 and Eq. 2.18). We also have the same arithmetic rules for the both representations.

The Cantor basis

In practice, Cantor mentioned a different representation for elements of W_m for implementation purpose. Gao and Mateer [GM10] termed the representation as "Cantor basis". The Cantor basis $(v_i)_{0 \le i < m}$ satisfies

$$v_0 = 1, \quad v_i^2 + v_i = v_{i-1} \text{ for } i > 0 \ . \tag{2.19}$$

Given an element $a = \omega_{(v_i)_{0 \le i < m}}(i) \in W_m$ in the Cantor basis, we can apply the linear operation S to a efficiently

$$S \cdot a = \omega_{(v_i)_{0 \le i < m}}(i/2) = \omega_{(v_i)_{0 \le i < m}}(i \gg 1) \in W_{m-1} \ . \tag{2.20}$$

With respect to the basis $(v_i)_{0 \le i < m}$, its sequence of subspaces are

$$W_0 := \{0\}, \quad W_i := \text{span}\{v_0, v_1, \ldots, v_{i-1}\} \text{ for } i > 0 \ ,$$

which consist with Eq. 2.17. Recall that W_m is a field with the basis $(v_i)_{0 \le i < m}$ if m a power of 2. We note that $W_0 \subset W_1 \subset W_2 \subset \cdots \subset W_m$. Hence, we can arbitrarily transform the elements in a smaller space to a larger space by padding zero to the extra dimensions *without any cost*. We thus treat elements in smaller fields as elements in larger fields arbitrarily.

Field Isomorphism

Table 2.1 lists the field isomorphism for the basis elements from the Cantor basis to $\widetilde{\mathbb{F}}_{65536}$. For finding the binary expansion of an element in $\widetilde{\mathbb{F}}_{65536}$ that is the same element in the Cantor basis, the change of field representations is a matrix-vector multiplication over $\mathbb{F}_2$, and the matrix is precisely the Tab. 2.1.

To generate the matrix of isomorphism, we first set the same representative for $1 = \texttt{0x1}$ in both representations. Then we solve the equation $s_1(x) = x^2 + x = 1$ in $\widetilde{\mathbb{F}}_{65536}$ to find the role playing v_1 in $\widetilde{\mathbb{F}}_{65536}$. It results in $x = \omega_{\widetilde{\mathbb{F}}_{65536}}(\texttt{0x2})$ which is the v_1 represented in $\widetilde{\mathbb{F}}_{65536}$. To find v_3 in the representation of $\widetilde{\mathbb{F}}_{65536}$, we solve $x^2 + x = v_2 = \omega_{\widetilde{\mathbb{F}}_{65536}}(\texttt{0x4})$ and result in $v_3 = \omega_{\widetilde{\mathbb{F}}_{65536}}(\texttt{0xa})$. Following the same process, we can fill out all the contents of the matrix.

2.3 Selected Instructions

In this section, we discuss the useful instructions for multiplication in binary fields. To perform the arithmetic in finite fields, the rule of thumb is always to choose an equivalent native instruction supported on the platform. However, there are only a few fields where multiplications correspond to native hardware instructions in mainstream CPUs, so the efficient software implementation of arithmetic is a topic of great interest in computer engineering.

An instruction describes the real operation that a computer manipulates the data. The machines see a program as a sequence of instructions. A typical computer does: (1) pick up the value at the location a, (2) pick up the value at the location b, (3) do some operation specified by the instruction to the two values, (4) and write back the result to the location c. Here the location refers to memory or a register. The registers are the fastest storage and are usually limits on its size. The width of registers is termed a machine word which is generally

Table 2.1: Field Isomorphism: Cantor basis to $\widetilde{\mathbb{F}}_{65536}$.

	Cantor basis		$\widetilde{\mathbb{F}}_{65536}$				
W_1	v_0	0x1	$(1,0,0,0,$	$0,0,0,0,$	$0,0,0,0,$	$0,0,0,0)$	0x1
W_2	v_1	0x2	$(0,1,0,0,$	$0,0,0,0,$	$0,0,0,0,$	$0,0,0,0)$	0x2
W_4	v_2	0x4	$(0,0,1,0,$	$0,0,0,0,$	$0,0,0,0,$	$0,0,0,0)$	0x4
	v_3	0x8	$(0,1,0,1,$	$0,0,0,0,$	$0,0,0,0,$	$0,0,0,0)$	0xa
W_8	v_4	0x10	$(0,0,1,0,$	$1,0,0,0,$	$0,0,0,0,$	$0,0,0,0)$	0x14
	v_5	0x20	$(0,0,0,0,$	$0,1,0,0,$	$0,0,0,0,$	$0,0,0,0)$	0x20
	v_6	0x40	$(0,0,1,1,$	$0,0,1,0,$	$0,0,0,0,$	$0,0,0,0)$	0x4c
	v_7	0x80	$(0,1,1,1,$	$0,1,0,1,$	$0,0,0,0,$	$0,0,0,0)$	0xae
W_{16}	v_8	0x100	$(0,1,0,0,$	$1,0,1,0,$	$1,0,0,0,$	$0,0,0,0)$	0x152
	v_9	0x200	$(0,1,1,1,$	$0,1,0,0,$	$0,1,0,0,$	$0,0,0,0)$	0x22e
	v_{10}	0x400	$(0,0,0,0,$	$0,0,0,1,$	$0,0,1,0,$	$0,0,0,0)$	0x480
	v_{11}	0x800	$(0,0,1,1,$	$0,0,1,0,$	$0,1,0,1,$	$0,0,0,0$	0xa4c
	v_{12}	0x1000	$(0,1,0,1,$	$1,1,1,1,$	$0,0,1,0,$	$1,0,0,0)$	0x14fa
	v_{13}	0x2000	$(0,1,0,0,$	$0,0,0,0,$	$0,0,0,0,$	$0,1,0,0)$	0x2002
	v_{14}	0x4000	$(0,1,0,0,$	$1,1,0,0,$	$0,0,1,1,$	$0,0,1,0)$	0x4c32
	v_{15}	0x8000	$(0,0,0,1,$	$1,0,0,1,$	$0,1,1,1,$	$0,1,0,1)$	0xae98

the unit that a computer manipulates data. It is used to temporarily store the data and provide the data to arithmetic or logic units performing the real operations. We assume that a program loads inputs from memory into registers, executes the computations in the registers, and finally stores the outputs back to memory.

To make a high-performance software on a particular purpose, we generally aim for 2 directions: (1) minimize the number of instructions used and (2) reduce the number of access to memoryy. It relies on keeping desired values in registers as much as possible. The modern computers usually support additional instructions or registers through instruction set extensions. A powerful instruction helps to reduce the number of instructions applied. Our first strategy on optimizations is choosing the best specific instructions that result in the best performance.

2.3.1 Bit Operations: Multiplications in $\mathbb{F}_2$ and $\mathbb{F}_4$

$\mathbb{F}_2$ is probably the only field with full hardware support, which multiplication and addition correspond to `AND` and `XOR` respectively. However, CPUs usually manipulate data in 32-bit or larger machine words today, instead of one "bit" for $\mathbb{F}_2$. The main issue for implementing software over $\mathbb{F}_2$ relies on how to utilize the full width of the machine word. It usually involves discovering the parallelism in the particular works.

In the case of $\mathbb{F}_4$, we believe that the best way to multiply is still to use bit operations. For this, the 2-bits element in $\mathbb{F}_4$ is often stored in separate registers, or "bit-sliced." A multiplication in $\mathbb{F}_4$ costs 4 `AND` and 3 `XOR` by Eq. 2.9.

2.3.2 Instruction for Multiplying Boolean Polynomials

For multiplying polynomials in $\mathbb{F}_2[x]$, fortunately, there are native instructions for this purpose on various platforms. In the x86 platform, the instruction is `PCLMULQDQ` [GK14] which was available since 2010 from the Westmere microarchitecture. On the ARMv8 platform, the equivalent instruction for polynomial multiplication is named `PMULL` or `PMULL2`. In this thesis, we demonstrate our techniques with the instructions in the x86 platform.

The `PCLMULQDQ` instruction performs the carryless multiplication of 2 64-bits polynomials, i.e., $\texttt{PCLMULQDQ} : \mathbb{F}_2[x]_{<64} \times \mathbb{F}_2[x]_{<64} \rightarrow \mathbb{F}_2[x]_{<127}$. Since the instruction operates on polynomials of fixed length, the main work on implementation is to adjust the sequence of instructions that result in the least running time for the various length of polynomials.

2.3.3 Exploiting Data-level Parallelism

Whenever the same computations need to be carried out on multiple independent inputs, these computations can in principle be carried out in parallel. This kind of parallelism is called data-level parallelism.

The SIMD Instruction Set

Single Instruction Multiple Data(SIMD) is a standard way to realize data-parallelism on modern computers. We some use the term vector instructions for the SIMD instructions alternatively. The idea of SIMD is to keep multiple values of the same type in one register, and then perform the arithmetic operations on all of these values in parallel with vector instructions.

Today, the vector instructions are usually supported through an instruction set extension and are commonly available on various platforms. These instruction sets typically differ from the number of registers provided, the width of registers, and the collection of instructions supported. For example, there are mainly SSE, AVX, and AVX-512 [Int15], providing 128-, 256-, and 512-bits registers respectively, available on the x86 platform. There is the NEON instruction set available on the ARM platform as well.

In this thesis, we develop the techniques for the arithmetic of field based on the SSE instruction set. The more advanced instruction set, e.g., AVX in x86, can usually be seen as an extension of SSE with more registers and larger machine word. The SSE instruction set provides 16×128-bit `xmm` registers, which can be accessed as vectors of 32-bit floats, 64-bit (double-precision) floats, 64-, 32-, 16-, or 8-bit signed or unsigned integers. It also contains a set of SIMD instructions including arithmetic, logic, and data movement instructions.

2.3.4 SIMD Table Lookup Instruction

Among the common SIMD instructions, one of the most useful vector instructions in this thesis is the SIMD table lookup instruction. In addition to the table lookup, we also use the instruction to move byte data.

The SIMD table lookup instruction is common on modern CPUs. On the x86 platform, the table lookup instruction is `PSHUFB`, which is available since the SSSE3 instruction set. In one instruction, `PSHUFB` can perform 16 table lookups

simultaneously. In the AVX-2 instruction set, its 256-bits version `VPSHUFB` simply performs the `PSHUFB` twice in one instruction. On ARM platforms with the NEON instruction set, the table lookup instruction are named `VTBL` or `VTBX`. We adopt the basic `PSHUFB` to develop further implementations in this thesis.

The `PSHUFB` instruction takes two 16-byte sources which one is a table with 16 one-byte entries $(T[0], \ldots, T[15])$ and the other are 16 indices $(i_0, \ldots, i_{15})$. In its destination register, A CPU will fill in $T[i_j \mod 16]$ for $i_j \geq 0$ or 0 for $i_j < 0$ in the j-th byte, i.e.,

$$\begin{aligned}&\texttt{PSHUFB}((T[0], T[1], \ldots, T[15]), (i_0, i_1, \ldots, i_{15}))\\ &\rightarrow (T[i_0 \bmod 16], T[i_1 \bmod 16], \ldots, T[i_{15} \bmod 16])\ .\end{aligned}$$

Here is an example of usage: a table $(x_0, x_1, x_2, ..., x_{15})$ is loaded as the first operand, and the indices as the second operand, e.g.,

$$\begin{aligned}&\texttt{PSHUFB}((x_0, x_1, x_2, ..., x_{15}), (1, 0, 3, 17, -1, -17, 9, ...))\\ &\rightarrow (x_1, x_0, x_3, x_1, 0, 0, x_9, ...).\end{aligned}$$

2.4 Fast Linear Algebra with Vector Instructions

In this section, we implement standard subroutines of linear algebra over $\mathbb{F}_2$ with the SSE instruction set on the x86 platform.

2.4.1 Matrix Transpose

The first technique is an efficient method for matrix transpose. In [War12], Warren describes a divide-and-conquer method for transposing 8×8 bit-matrix. For implementation with vector instruction set, Van der Hoeven *et al.* [vdHLL17] and Chen *et al.* [CCK+17] showed similar techniques for the bit- and byte-matrix in AVX-2 instruction set. We depict the methods in this section.

Generally speaking, to transpose a matrix $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$, we first rearrange the contents of M to $\begin{bmatrix} A & C \\ B & D \end{bmatrix}$ and then perform the process of transpose to all 4 sub-matrices A, B, C, and D. The process ends if the sub-matrices $A, \ldots, D$ cannot be divided anymore. Based on the minimized size of the sub-matrix, we perform the data shuffling even in bit-levels. Warren had shown how to accomplish the shuffle of bit-level data with basic bit-operations in [War12].

The abstract idea for transposing bit- or byte-level matrices is the same. However, we implement the two types of matrix transpose with different instructions. For bit-level transpose, we follow Warren's implementation of bit-operations but program it with vector instructions for performing multiple transposes in parallel. For byte-level transpose, the data movement in byte-, word-, or other broader structures is well supported with SSE instruction set, and we thus implement the transpose of byte-matrix with instructions of data swizzling.

We show examples of the byte-matrix transpose in the rest of this section. The interpretation of data plays a vital role for transpose in a SIMD instruction set. For example of transposing a 4×4 byte-matrix $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$, let $A =$

$\begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix}$ is a 2×2 sub-matrix, where a_i is one byte for $i = 0, \ldots, 3$, and similar symbols for sub-matrices $B = \begin{bmatrix} b_0 & b_1 \\ b_2 & b_3 \end{bmatrix}$, C, and D. If the matrix M is formatted in one 16-bytes vector register, we can accomplish the 4×4 transpose of M in one `PSHUFB` instruction if the data A, B, C, and D located in the same 16-byte register.

$$\boxed{(a_0, a_1, a_2, a_3), (b_0, b_1, b_2, b_3), (c_0, c_1, c_2, c_3), (d_0, d_1, d_2, d_3)}$$
$$\xrightarrow{\texttt{PSHUFB}} \boxed{(a_0, a_2, a_1, a_3), (c_0, c_2, c_1, c_3), (b_0, b_2, b_1, b_3), (d_0, d_2, d_1, d_3)} .$$

Here a row box represents the data in the same register.

While the contents of matrices locate across registers, we can perform many transposes in parallel by swapping data between registers. We show an example of a 4×4 transpose among 4 registers. Note that there are 2 swaps performed in each step.

$$\begin{array}{|l|} \hline (a_0, a_1, \underline{b_0, b_1}), \ldots \\ \hline (a_2, a_3, \underline{b_2, b_3}), \ldots \\ \hline (\underline{c_0, c_1}, d_0, d_1), \ldots \\ \hline (\underline{c_2, c_3}, d_2, d_3), \ldots \\ \hline \end{array} \xrightarrow{\text{2 swaps}} \begin{array}{|l|} \hline (a_0, \underline{a_1}, c_0, \underline{c_1}), \ldots \\ \hline (\underline{a_2}, a_3, \underline{c_2}, c_3), \ldots \\ \hline (b_0, \underline{b_1}, d_0, \underline{d_1}), \ldots \\ \hline (\underline{b_2}, b_3, \underline{d_2}, d_3), \ldots \\ \hline \end{array} \xrightarrow{\text{2 swaps}} \begin{array}{|l|} \hline (a_0, a_2, c_0, c_2), \ldots \\ \hline (a_1, a_3, c_1, c_3), \ldots \\ \hline (b_0, b_2, d_0, d_2), \ldots \\ \hline (b_1, b_3, d_1, d_3), \ldots \\ \hline \end{array} .$$

2.4.2 Linear Transformation over $\mathbb{F}_2$

In this section, we demonstrate how to perform a particular linear transformation, i.e., the multiplication of a pre-defined matrix by a vector over $\mathbb{F}_2$. Algorithmically, we perform the linear transform with the famous method of the four Russians(M4R) [AH74] [ABH10]. Comparing the usual M4R [ABH10] which is based on memory access with the value of data, we use the SIMD table lookup instruction in our implementations. Hence, we perform a batch of multiplications in parallel with vector instructions.

4×4 matrix-vector multiplication

The `PSHUFB` instruction performs precisely the multiplication of a pre-defined 4×4 matrix by a batch of arbitrary vectors. Given a matrix $A \in \mathbb{F}_2^{4 \times 4}$, we can prepare a table whose contents are $(A \cdot \texttt{0x0}, A \cdot \texttt{0x1}, \ldots, A \cdot \texttt{0xf})$, i.e., all possible products. Then, in one `PSHUFB` instruction, we can perform 16 multiplications of A by $b_0, b_1, \ldots, b_{15}$, where $b_i \in \mathbb{F}_2^4$ for all i.

$$\begin{aligned} &\texttt{PSHUFB}((A \cdot \texttt{0x0}, A \cdot \texttt{0x1}, \ldots, A \cdot \texttt{0xf}), (b_0, b_1, \ldots, b_{15})) \\ &\rightarrow (A \cdot b_0, A \cdot b_1, \ldots, A \cdot b_{15}) \ . \end{aligned}$$

8×8 matrix-vector multiplication

Although the `PSHUFB` queries table with only 16 byte-entries, we can still multiply an 8×8 matrix with 2 `PSHUFB`'s due to the linearity. Suppose we multiply

a matrix $A \in \mathbb{F}_2^{8\times 8}$ by vectors $b_0, b_1, \ldots, b_{15}$ where $b_i \in \mathbb{F}_2^8$ for all i. If we divide the b_i's into low- and high- nibbles, i.e., $b_i = b_{i,\text{low 4-bits}} + b_{i,\text{high 4-bits}}$, we can accomplish $A \cdot b_i$ with two multiplications

$$A \cdot b_i = A \cdot b_{i,\text{low 4-bits}} + A \cdot b_{i,\text{high 4-bits}} \ .$$

Since the $b_{i,\text{low 4-bits}}$ or $b_{i,\text{high 4-bits}}$ both contains 16 possible values, we can perform the multiplications by 4-bits multipliers with the `PSHUFB` instruction. Note that $b_{i,\text{low 4-bits}}$ is 4-bits data, but $A \cdot b_{i,\text{low 4-bits}}$ is 8-bits data. Now we have to prepare 2 tables for values of

$$\texttt{TAB}_{\texttt{LOW}}(A) = (A \cdot \texttt{0x0}, A \cdot \texttt{0x1}, \ldots, A \cdot \texttt{0xf}) \in \mathbb{F}_2^{8\times 16}$$

and

$$\texttt{TAB}_{\texttt{HIGH}}(A) = (A \cdot \texttt{0x00}, A \cdot \texttt{0x10}, \ldots, A \cdot \texttt{0xf0}) \in \mathbb{F}_2^{8\times 16},$$

where all entries of the tables are in $\mathbb{F}_2^8$ (one byte). The products of $A \cdot b_i$'s can be calculated with Alg. 1.

Algorithm 1: SIMD matrix-vector multiplication over $\mathbb{F}_2$

1 `mat_mul_8x8(` $\texttt{TAB}_{\texttt{LOW}}(A), \texttt{TAB}_{\texttt{HIGH}}(A), (b_0, \ldots, b_{15})$) :
input : A multiplication table for A : $\texttt{TAB}_{\texttt{LOW}}(A)$
A multiplication table for A : $\texttt{TAB}_{\texttt{HIGH}}(A)$
$(b_0, \ldots, b_{15}) \in \mathbb{F}_2^{8\times 16}$.
output: $(A \cdot b_0, \ldots, A \cdot b_{15}) \in \mathbb{F}_2^{8\times 16}$.
2 $B_{low} \in \mathbb{F}_2^{4\times 16} \leftarrow (b_0, \ldots, b_{15})$ `AND` $(\texttt{0xf}, \ldots, \texttt{0xf})$.
3 $B_{high} \in \mathbb{F}_2^{4\times 16} \leftarrow ((b_0, \ldots, b_{15}) \gg 4)$ `AND` $(\texttt{0xf}, \ldots, \texttt{0xf})$.
4 $C_{low} \in \mathbb{F}_2^{8\times 16} \leftarrow \texttt{PSHUFB}(\texttt{TAB}_{\texttt{LOW}}(A), B_{low})$.
5 $C_{high} \in \mathbb{F}_2^{8\times 16} \leftarrow \texttt{PSHUFB}(\texttt{TAB}_{\texttt{HIGH}}(A), B_{high})$.
6 **return** $C_{low} + C_{high}$.

$m \times m$ matrix-vector multiplication

We can continue to extend the size of the matrix as the generalization of the 8×8 matrix-vector product. Suppose we multiply a pre-defined $m \times m$ matrix $A \in \mathbb{F}_2^{m\times m}$ by a vector $b \in \mathbb{F}_2^m$ for m is a power of 2. We assume a basis $(v_i)_{0\le i<m}$ for $\mathbb{F}_2^m$, i.e., $v_0 = (1, 0, \ldots), v_1 = (0, 1, 0, \ldots), v_2 = (0, 0, 1, \ldots), \ldots, v_{m-1} = (0, \ldots, 1)$. With M4R of 4-bits indices, we first prepare $m/4$ tables for all possible products of A and all vectors in $\text{span}(v_0, \ldots, v_3)$, ... , and $\text{span}(v_{m-4}, \ldots, v_{m-1})$. By splitting b to 4-bit chunks, i.e.,

$$b = \sum_{i=0}^{(m/4)-1} b_i \quad \text{where } b_i \in \text{span}(v_{4\cdot i}, v_{4\cdot i+1}, v_{4\cdot i+2}, v_{4\cdot i+3}) \text{ for } 0 \le i < \frac{m}{4} \ ,$$

we can then compute

$$A \cdot b = \sum_{i=0}^{(m/4)-1} A \cdot b_i,$$

where each $A \cdot b_i$ is calculated by the query of one particular table, and one query comprises $m/8$ `PSHUFB` instructions due to the one-byte results of `PSHUFB`.

However, we have to rearrange the format of the input data to work with SIMD instructions efficiently. For example of $m = 32$, suppose that we multiply a pre-defined matrix $A \in \mathbb{F}_2^{32\times 32}$ by continuing 32-bits inputs $\{a \in \mathbb{F}_2^{32}, b \in \mathbb{F}_2^{32}, \ldots\}$ where $a = a_0||a_1||a_2||a_3$ is the concatenation of four 8-bits data. To apply the `PSHUFB` to desired data, we have to collect all first bytes to the first register, all second bytes to the second register, and so on.

$$\begin{array}{|c|}\hline (a_0, a_1, a_2, a_3), \ldots \\ \hline (b_0, b_1, b_2, b_3), \ldots \\ \hline (c_0, c_1, c_2, c_3), \ldots \\ \hline (d_0, d_1, d_2, d_3), \ldots \\ \hline \end{array} \xrightarrow{4\times 4 \text{ Transpose}} \begin{array}{|c|}\hline (a_0, b_0, c_0, d_0), \ldots \\ \hline (a_1, b_1, c_1, d_1), \ldots \\ \hline (a_2, b_2, c_2, d_2), \ldots \\ \hline (a_3, b_3, c_3, d_3), \ldots \\ \hline \end{array} .$$

Then we can apply the Algo. 1 to the transposed inputs with proper tables for the transposed results of the products. In other words, before and after the Algo. 1 is applied, we have to perform an $(m/8) \times (m/8)$ byte-matrix transpose, which is also performed in the SIMD way with the method in Sec. 2.4.1.

Chapter 3

Multiplication in Binary Fields

We discuss four implementations of multiplications in binary fields in this chapter. In section 3.1, we describe the multiplication in fields of polynomial representations with SIMD instructions. In section 3.2, we discuss the multiplication by subfield elements in tower fields. In section 3.3, we discuss the implementations of multiplication under cryptographic requirements. Last, Section 3.4 describes the multiplication in bigger fields with the `PCLMULQDQ` instruction.

3.1 Multiplication in $\mathbb{F}_{2^m}$ for $m \leq 16$

Many researchers have discussed fast implementations with SIMD tabular lookup instructions on modern CPUs for the polynomial representation of binary fields, e.g., [CCC+09, Anv11, CYC13, PGM13, GRU14]. They reported many results in the area of coding and cryptography for various platforms including x86 and ARM. In contrast to standard implementations, these SIMD optimizations provide about an order of magnitude improvement. We discuss these methods in this section.

In Section 3.1.1, we show multiplying a particular element in finite fields is equal to performing a linear transformation. In Secction 3.1.2, we discuss the multiplication of $\mathbb{F}_{256}$ as well as some fast multiplication by specific elements. In Section 3.1.3, we discuss the techniques for multiplying elements in $\mathbb{F}_{256^2}$.

3.1.1 Field Multiplication as Linear Transformation

In this section, we show that multiplying a particular element in a finite field is equal to a linear transformation. The rule can be generalized easily from the following example.

For example of multiplication in $\mathbb{F}_4 := \mathbb{F}_2[x]/(x^2 + x + 1)$, we multiply a field element $a \in \mathbb{F}_4$ by an element $b \in \mathbb{F}_4$. Let the polynomial forms of $a(x) = a_0 + a_1 x, b(x) = b_0 + b_1 x \in \mathbb{F}_2[x]_{<2}$ and the vector forms $\boldsymbol{a} = [a_0, b_0]^T, \boldsymbol{b} = [b_0, b_1]^T \in \mathbb{F}_2^2$. The first step of the multiplication(Eq. 2.7) is equal to a matrix-

vector multiplication

$$a(x) \cdot b(x) \rightarrow \begin{bmatrix} b_0 & 0 \\ b_1 & b_0 \\ 0 & b_1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \quad .$$

The reduce step of the multiplication(Eq. 2.8) is equivalent to a linear transformation

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \quad ,$$

in which the matrix is depended on the construction of $\mathbb{F}_4$. We can combine the two step into a linear transformation

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_0 & 0 \\ b_1 & b_0 \\ 0 & b_1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} b_0 & b_1 \\ b_1 & b_0 + b_1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad ,$$

which is equal to the operations in Eq. 2.9. Hence, the multiplication of a by b is equal to a matrix-vector multiplication, and the matrix depends on the particular form of b.

Since we can perform a linear transformation with the SIMD `PSHUFB` instruction in Sec. 2.4.2, we can perform multiple field multiplications in parallel with the same technique. However, in the previous example, we have to prepare tables for all possible value of b since one particular value of b corresponds to one specific form of the matrix.

In the case of multiplication in $\mathbb{F}_{16}$, suppose that we multiply a bunch of $(a_0, a_1, \ldots, a_{15}) \in \mathbb{F}_{16}^{16}$ by an element $b \in \mathbb{F}_{16}$. We can prepare 16 (multiplication) tables for the products of all possible values of b. Then, by load the correct table with the value of b, we can perform the multiplications with one `PSHUFB`, i.e.,

$$\begin{aligned} &(\text{table of } b \in \mathbb{F}_2^4), (a_0, \quad a_1, \quad \ldots \quad , \quad a_{16}) \\ &\xrightarrow{\texttt{PSHUFB}} (a_0 \cdot b, \quad a_1 \cdot b, \quad \ldots, \quad a_{15} \cdot b) \ . \end{aligned}$$

3.1.2 Multiplication in $\mathbb{F}_{256}$

The multiplication of elements in $\mathbb{F}_{256}$ has been highly optimized in software today. For software design, it has been reported in the literature [CCC+09, CYC13, PGM13, GRU14] that SIMD table lookup instructions, e.g., `PSHUFB` (Packed-Shuffle-Byte) in x86, can accelerate multiplication in small finite fields. These proposed methods are basically the same for $\mathbb{F}_{256}$.

The scalar multiplication over $\mathbb{F}_{256}$ is similar to $\mathbb{F}_{16}$ except that the number of prepared tables becomes 256 and one multiplication costs 2 `PSHUFB`. Once the correct tables are loaded into registers, the operations are the same as Alg. 1.

For further optimization of multiplications in $\mathbb{F}_{256}$, we detail the operations of multiplications in the viewpoint of polynomials. An element in $\mathbb{F}_{256}$ is represented as a polynomial of degree-7 over $\mathbb{F}_2$. The multiplication in $\mathbb{F}_{256}$ is performed as multiplication of 2 involved $\mathbb{F}_2[x]$ polynomials and then mod the chosen polynomial ω(`0x11d`) to reduce the degree of the resulting polynomial. Suppose that we multiply a vector $(a^{(0)}, a^{(1)}, \ldots, a^{(15)}) \in \mathbb{F}_{256}^{16}$ by a scalar

$b \in \mathbb{F}_{256}$. We can cut each $a^{(i)}$ into two nibbles corresponding to higher- and lower-degree parts of their polynomial form as follows:

$$\begin{aligned} a^{(i)} \cdot b &= \left(a_7^{(i)} x^7 + \cdots + a_0^{(i)}\right) \cdot b(x) \\ &= \left(a_7^{(i)} x^3 + \cdots + a_4^{(i)}\right) \cdot \left(x^4 b(x)\right) + \left(a_3^{(i)} x^3 + \cdots + a_0^{(i)}\right) \cdot b(x) \ . \end{aligned} \tag{3.1}$$

Therefore, we can obtain the desired result by loading two pre-computed multiplication tables for the values of $x^4 b(x)$ and $b(x)$, and followed by two `PSHUFB`'s and one `XOR` for results.

Last but not least, we have to ensure the size of all tables in the multiplication fits into the fastest(L1) cache of moder CPUs. There are 256 possible values for $b(x)$, and each value of $b(x)$ requires two tables of 16 byte-entries for multiplying $x^4 b(x)$ and $b(x)$ respectively. Hence, the total size of all tables is $2 \cdot 256 \cdot 16 = 8192$ bytes = 8 KB, which is smaller than a typical L1 cache(16 - 64 KB) on modern CPUs.

Multiplying Specific Elements in $\mathbb{F}_{256}$

We can further optimize for some scalars for fewer instructions than general $\mathbb{F}_{256}$ multiplication. We start with Anvin's technique [Anv11] for the obvious one, namely, the multiplication of $a \in \mathbb{F}_{256}$ by $b = x = \omega(\texttt{0x2}) \in \mathbb{F}_2[x]$.

$$\begin{aligned} a \cdot b &= (a_7 x^7 + \cdots + a_0) \cdot (x) && (\text{mod } \omega(\texttt{0x11d})) \\ &= (a_6 x^7 + \cdots + a_0 x) + a_7 x^8 && (\text{mod } \omega(\texttt{0x11d})) \\ &\to (a_6 x^7 + \cdots + a_0 x) + a_7 \cdot \omega(\texttt{0x1d}) \end{aligned} \tag{3.2}$$

The standard way to compute this in C language is

`((a<<1)&0xff)^((a&0x80)?0x1d:0)`.

The `((a<<1)&0xff)` can be accomplished by bit-shift or addition in byte-level. For `((a&0x80)?0x1d:0)`, one can treat a_7 as a predicate for conditional add a constant `0x1d`. Since the `PSHUFB` in x86 platform treats a_7 in a complementary way `((a&0x80)?0:...)`, we implemented it in x86 as

`((a&0x80)?0:0x1d)^0x1d`.

We noted that, in certain situations, one can yield the complementary

`((a&0x80)?0:0xe2)`,

which is one `XOR` fewer than the previous implementation.

Similarly, we can implement the efficient multiplication by small powers of $\omega(\texttt{0x2})$ with one fewer `PSHUFB`, comparing to multiplication by a generic element, as follows:

$$\begin{aligned} & a \cdot \omega_{\mathbb{F}_{256}}(x^3) \\ = \ & \left(a_4 x^7 + \cdots + a_0 x^3\right) \\ & + \left(a_7 x^{10} + a_6 x^9 + a_5 x^8\right) \bmod (x^8 + x^4 + x^3 + x^2 + 1) \\ \to \ & (a_4 x^7 + \cdots + a_0 x^3) + (a_7 x^2 + a_6 x + a_5) \cdot (x^4 + x^3 + x^2 + 1) \ . \end{aligned} \tag{3.3}$$

The modular part still requires one `PSHUFB`, but the other `PSHUFB` for $a_0, \ldots a_4$ can be replaced by a logical shift (`PSLLW` or `PSRLW`). Since no preloaded table requires for bit-shift instructions, this optimization results in higher throughput and lower latency, as well as lower register pressure. However, we note that the lack of byte-sized `PSLLB` or `PSRLB` makes things more awkward than expected.

3.1.3 Multiplication in $\mathbb{F}_{256^2}$

We discuss the SIMD multiplication in $\mathbb{F}_{256^2}$ on modern CPUs in this section.

Recall that we construct the field $\mathbb{F}_{256^2}$ as an extension field of $\mathbb{F}_{256}$, i.e.,

$$\mathbb{F}_{256^2} := \mathbb{F}_{256}[X]/(X^2 + \omega_{\mathbb{F}_{256}}(\texttt{0x8}) \cdot X + 1) \ .$$

Hence, an $\mathbb{F}_{256^2}$ element is represented as a polynomial of degree 1 over $\mathbb{F}_{256}$ or a vector with 2 entries ($\mathbb{F}_{256}^2$). We select the irreducible polynomial $X^2 + \omega_{\mathbb{F}_{256}}(\texttt{0x8})X + 1$, abbreviated as `0x10801`, for the fast multiplication of the coefficient $\omega_{\mathbb{F}_{256}}(\texttt{0x8})$ in $\mathbb{F}_{256}$ in Section 3.1.2.

For multiplication in the extension field of $\mathbb{F}_{256}$, multiplying a by b for $a, b \in \mathbb{F}_{256^2}$ is performed as multiplying polynomials of degree 1 over $\mathbb{F}_{256}$. In other words, let a be $a(X) \in \mathbb{F}_{256}[x]_{<2}$ and b be $b(X) \in \mathbb{F}_{256}[x]_{<2}$. We implement $a \cdot b$ as $a(X) \cdot b(X)$ over $\mathbb{F}_{256}$ with Karatsuba's multiplication (please refer to [Ber08]).

$$\begin{array}{rl} & (a_1X + a_0) \cdot (b_1X + b_0) \\ = & a_1b_1X^2 + [a_1b_1 + a_0b_0 + (a_0 + a_1)(b_0 + b_1)]X + a_0b_0 \\ \rightarrow & [\omega_{\mathbb{F}_{256}}(\texttt{0x8}) \cdot a_1b_1 + a_1b_1 + a_0b_0 + (a_0 + a_1)(b_0 + b_1)]\, X \\ & + a_1b_1 + a_0b_0 \ . \end{array} \tag{3.4}$$

In summary, there are three general multiplications in $\mathbb{F}_{256}$ for $a_1 \cdot b_1$, $a_0 \cdot b_0$, and $(a_0 + a_1) \cdot (b_0 + b_1)$ and one multiplication by $\omega_{\mathbb{F}_{256}}(\texttt{0x8})$ which can be done by Eq. 3.3. Again, since we perform the multiplication in $\mathbb{F}_{256^2}$ in a SIMD way with `PSHUFB` on modern CPUs, it costs $3 \times 2 + 1$ `PSHUFB` instructions in total. As for the pre-computed multiplication tables, this construction uses the same tables with $\mathbb{F}_{256}$, which are 256×2 tables (16 bytes for each, 8 Kbytes in total).

The construction of $\mathbb{F}_{256^2}$ is backward compatible with $\mathbb{F}_{256}$ for both the stored values of field elements and the multiplication in $\mathbb{F}_{256}$. Since $\mathbb{F}_{256^2}$ is represented as an extension field of $\mathbb{F}_{256}$, an $\mathbb{F}_{256}$ element can naturally map to $\mathbb{F}_{256^2}$ as a constant polynomial in $\mathbb{F}_{256}[X]$. This construction also allows us to reuse the precomputed tables and highly-optimized implementations of $\mathbb{F}_{256}$. Furthermore, a multiplication in $\mathbb{F}_{256^2}$ by an element in $\mathbb{F}_{256}$ would be preserved byte-wise in the extension fields; in other words, it behaves as two multiplications in $\mathbb{F}_{256}$.

Comparison with the representation of polynomials over $\mathbb{F}_2$

Besides the compatibility of $\mathbb{F}_{256}$, a more critical reason for adopting the extension field of $\mathbb{F}_{256}$ instead of polynomials of degree-15 over $\mathbb{F}_2$ is the size of multiplication tables. We compare the $\mathbb{F}_{256^2}$ multiplication with the multiplication in $\mathbb{F}_{2^{16}}$(the field represented as polynomials of degree-15 over $\mathbb{F}_2$) presented by Plank, Greenan, and Miller [PGM13]. The multiplication is a straightforward generalization of multiplication as a linear transformation in Sec. 3.1.1. In their proposal, a 16-bit element is divided into four nibbles, and one multiplication can be computed using eight table lookups (two for each nibble). It requires 8 tables of 16 byte-entries for one element in $\mathbb{F}_{2^{16}}$ and the total storage for multiplication tables are $65536 \times 8 \times 16$ bytes, which cannot possibly fit into the L1 cache of any of today's processors. Recall that the $\mathbb{F}_{256^2}$ multiplication, on the other hand, requires 8192 bytes of multiplication tables.

We can also compare the number of `PSHUFB` instructions in the multiplication of the two representations. The $\mathbb{F}_{256^2}$ multiplication costs 7 `PSHUFB` instructions.

However, the $\mathbb{F}_{2^{16}}$ multiplication requires 8 `PSHUFB` instructions. Hence, we conclude that the multiplication in $\mathbb{F}_{256^2}$ is more practical than in $\mathbb{F}_{2^{16}}$ on modern CPUs.

Data arrangement for $\mathbb{F}_{256^2}$

The data arrangement for storing $\mathbb{F}_{256^2}$ elements, which is 2 byte one element, also affects the efficiency of multiplication. Take 128-bit register as an example. The consecutive elements of $b(X) \in \mathbb{F}_{256^2}$ natively format as

$$(b_0^{(0)}, b_1^{(0)}, b_0^{(1)}, b_1^{(1)}, \ldots, b_0^{(7)}, b_1^{(7)}), (b_0^{(8)}, b_1^{(8)}, b_0^{(9)}, b_1^{(9)}, \ldots, b_0^{(15)}, b_1^{(15)}), \ldots \ .$$

For efficient SIMD multiplication in $\mathbb{F}_{256^2}$, one has to split the high and low bytes, i.e., the $b_1^{(i)}$ and $b_0^{(i)}$ since they associate to terms of different degree in $\mathbb{F}_{256}[X]$. One possible rearrangement of the data is

$$\begin{aligned}&(b_0^{(0)}, b_0^{(8)}, b_0^{(1)}, b_0^{(9)} \ldots, b_0^{(7)}, b_0^{(15)}),\\ &(b_1^{(0)}, b_1^{(8)}, b_1^{(1)}, b_1^{(9)}, \ldots, b_1^{(7)}, b_1^{(15)}) \ .\end{aligned}$$

If one can design the data format of $\mathbb{F}_{256^2}$ at the beginning, the coefficients of $b(X)$ can be stored in a split way, i.e., storing 16 elements of $b(X)$ in 2 memory slots as

$$(b_0^{(0)}, \ldots, b_0^{(15)}), (b_1^{(0)}, \ldots, b_1^{(15)}) \ .$$

Therefore there is no cost for splitting data in registers, and we can still maintain the backward compatibility for $\mathbb{F}_{256}$.

Fast multiplication with particular elements in $\mathbb{F}_{256^2}$

We can deduce the fast multiplication in $\mathbb{F}_{256^2}$ by some specific elements from multiplying polynomials of degree 1 in $\mathbb{F}_{256}$. Since the construction of $\mathbb{F}_{256^2}$ is designed to be backward compatible with $\mathbb{F}_{256}$, the coefficient multiplication in $\mathbb{F}_{256}[X]$ remains the same as $\mathbb{F}_{256}$, and multiplication between elements in $\mathbb{F}_{256^2}$ and $\mathbb{F}_{256}$ is simply to multiply a 2 terms polynomial by a constant polynomial. The fast multiplication by specific elements of $\mathbb{F}_{256}$ is still fast.

Beside the natural acceleration of multiplication by an element in $\mathbb{F}_{256}$, multiplication by elements in $\mathbb{F}_{256^2}$ can also be fast as long as the coefficient multiplication in $\mathbb{F}_{256}[X]$ belongs to fast multiplication in Section 3.1.2. We can present the fast multiplication for $b(X) \in \mathbb{F}_{256}[X] \cong \mathbb{F}_{256^2}$ by $X \in \mathbb{F}_{256^2}$ as

$$\begin{aligned} b(X) \cdot \omega_{\mathbb{F}_{256^2}}(X) &= (b_1 X + b_0) \cdot X \pmod{\texttt{0x10801}} \\ &= b_0 X + b_1 X^2 \mod \texttt{0x10801} \\ &\to (b_0 + \omega_{\mathbb{F}_{256}}(\texttt{0x8}) \cdot b_1) X + b_1 \ . \end{aligned} \tag{3.5}$$

The multiplication comprises one fast multiplication by $\omega_{\mathbb{F}_{256}}(\texttt{0x8})$ and one byte `XOR` operation.

3.2 Multiplication in Tower Fields

In this section, we show how to perform the multiplication efficiently in tower fields on modern computers.

3.2.1 Multiplication in $\widetilde{\mathbb{F}}_{16}$ and $\widetilde{\mathbb{F}}_{256}$

Recall the tower of fields in Eq. (2.11).

$$\begin{aligned} \mathbb{F}_4 &:= \mathbb{F}_2[x_0]/(x_0^2 + x_0 + 1), \\ \widetilde{\mathbb{F}}_{16} &:= \mathbb{F}_4[x_1]/(x_1^2 + x_1 + x_0), \\ \widetilde{\mathbb{F}}_{256} &:= \widetilde{\mathbb{F}}_{16}[x_2]/(x_2^2 + x_2 + x_1 x_0), \\ &\vdots \end{aligned}$$

The basic idea for multiplying elements in tower fields is to decompose the multiplication into several multiplications in a smaller field recursively. The multiplication in $\widetilde{\mathbb{F}}_{2^{2w}}$ is performed as polynomial multiplications in $\widetilde{\mathbb{F}}_{2^w}[x]$ which is a linear transformation over $\widetilde{\mathbb{F}}_{2^w}$. The multiplication of the coefficients $\widetilde{\mathbb{F}}_{2^w}$ again is a linear transformation over $\widetilde{\mathbb{F}}_{2^{w/2}}$, and the multiplication in $\widetilde{\mathbb{F}}$ is a linear transformation over $\mathbb{F}_2$ in the end.

Since the multiplication in $\widetilde{\mathbb{F}}_{16}$ or $\widetilde{\mathbb{F}}_{256}$ is also a linear transformation over $\mathbb{F}_2$, for multiplying elements in $\widetilde{\mathbb{F}}_{16}$ or $\widetilde{\mathbb{F}}_{256}$, we can use the same method of multiplication with `PSHUFB` in $\mathbb{F}_{256}$ in Sec. 3.1.2. However, we have different contents of multiplication tables concerning different field representations.

3.2.2 Decomposing Field Multiplication over $\widetilde{\mathbb{F}}_{256}$

Since modern computers can cache the multiplication tables of fields of 256 elements mentioned in Sec. 3.1.2, we choose the multiplication in $\widetilde{\mathbb{F}}_{256}$ as the building blocks for multiplication in extension fields of $\widetilde{\mathbb{F}}_{256}$ and decompose the multiplication in the extension fields into multiple multiplications in $\widetilde{\mathbb{F}}_{256}$. For example of multiplication in $\widetilde{\mathbb{F}}_{2^{16}}$, we regard elements in $\widetilde{\mathbb{F}}_{2^{16}}$ as polynomials in $\widetilde{\mathbb{F}}_{256}[x]_{<2}$ and perform the field multiplication as multiplying polynomials over $\widetilde{\mathbb{F}}_{256}$. We can reduce the number of field multiplication in $\widetilde{\mathbb{F}}_{256}$ with the Karatsuba's multiplication. Similar to $\widetilde{\mathbb{F}}_{2^{16}}$, we can perform the multiplication in $\widetilde{\mathbb{F}}_{2^{32}}$ as polynomial multiplication over $\widetilde{\mathbb{F}}_{2^{16}}$ and further decompose into field multiplications in $\widetilde{\mathbb{F}}_{256}$. In the end, all field multiplication in extension fields of $\widetilde{\mathbb{F}}_{256}$ is performed with some particular field multiplications in $\widetilde{\mathbb{F}}_{256}$ and some data movement to multiply the right components of the extension field.

3.2.3 Subfield Multiplication in Tower Fields

From the viewpoint of multiplying polynomials by a scalar, we regard the multiplication by a subfield element as a vector-scalar multiplication over the subfield. Hence, a subfield multiplication not only costs fewer operations than general multiplication, but also contains natural data-level parallelism. The cost of subfield multiplication depends on the size of the subfield and the length of the "vector". For example, to multiply $a \in \widetilde{\mathbb{F}}_{2^{2w}}$ by $b \in \widetilde{\mathbb{F}}_{2^w}$, the a is represented as a polynomial $a := a_0 + a_1 x \in \widetilde{\mathbb{F}}_{2^w}[x]$ with $a_0, a_1 \in \widetilde{\mathbb{F}}_{2^w}$. Hence the product of $a \cdot b \in \widetilde{\mathbb{F}}_{2^{2w}}$ is calculated as vector-scalar product over $\widetilde{\mathbb{F}}_{2^w}$, i.e., $(a_0 + a_1 x) \cdot b$. It is easy to generalize to the following proposition.

Proposition 1. *Given $a \in \widetilde{\mathbb{F}}_{2^{l_1}} = V_{l_1}, b \in \widetilde{\mathbb{F}}_{2^{l_2}} = V_{l_2}$, and $l_2 | l_1$, $a \cdot b \in V_{l_1}$ can be performed with l_1/l_2 field multiplications in $\widetilde{\mathbb{F}}_{2^{l_2}}$.*

The Multiplication in the subfield $\widetilde{\mathbb{F}}_{2^{l_2}}$ has to be further decomposed into field multiplications over $\widetilde{\mathbb{F}}_{256}$ for fitting the multiplication tables to L1 cache. We show some implementations of multiplication in $\widetilde{\mathbb{F}}$ in the following of this section.

3.2.4 Implementations

$\widetilde{\mathbb{F}}_{2^{128}}$ Multiplication by elements in $\widetilde{\mathbb{F}}_{256}$

The operation for multiplying an element of tower fields by an subfield element is implemented as a scalar multiplication of a vector over various subfields by a scalar with Prop. 1. In the case of multiplication between $a \in \widetilde{\mathbb{F}}_{2^{128}}$ and $b \in \widetilde{\mathbb{F}}_{256}$. We first translate $a = (a_0, \ldots, a_{15}) \in \widetilde{\mathbb{F}}_{256}^{16}$. The multiplication become

$$a \cdot b = (a_0, \ldots, a_{15}) \cdot b \quad \text{for } a_0, \ldots, a_{15}, b \in \widetilde{\mathbb{F}}_{256},$$

which can be performed with Alg. 1.

$\widetilde{\mathbb{F}}_{2^{128}}$ Multiplication by elements in $\widetilde{\mathbb{F}}_{2^{16}}$ or $\widetilde{\mathbb{F}}_{2^{32}}$

We focus on the data movement for $\widetilde{\mathbb{F}}_{2^{128}}$ multiplication by elements in $\widetilde{\mathbb{F}}_{2^{16}}$ or $\widetilde{\mathbb{F}}_{2^{32}}$ in this section. Since we use multiplication in $\widetilde{\mathbb{F}}_{256}$ as the essential building blocks, we consider the byte-data move for better efficiency with SIMD multiplication.

Suppose we are multiplying $\boldsymbol{a} \in \widetilde{\mathbb{F}}_{2^{16}}^{8} \cong \widetilde{\mathbb{F}}_{2^{128}}$ by $c \in \widetilde{\mathbb{F}}_{2^{16}}$, the byte sequence of these elements are

$$\boldsymbol{a} = ((a_0, a_1), (a_2, a_3), \ldots, (a_{14}, a_{15})) \in \widetilde{\mathbb{F}}_{2^{16}}^{8} \cong (a_0, \ldots, a_{15}) \in \widetilde{\mathbb{F}}_{256}^{16} \cong \widetilde{\mathbb{F}}_{2^{128}}$$

and $c = (c_0, c_1) \in \widetilde{\mathbb{F}}_{256}^{2} \cong \widetilde{\mathbb{F}}_{2^{16}}$. To multiply $\boldsymbol{a}$ by c over $\widetilde{\mathbb{F}}_{256}$, we may first split $\boldsymbol{a}$ to its odd and even bytes as

$$\boldsymbol{a} = \boldsymbol{a_{even}} + \boldsymbol{a_{odd}} = (a_0, 0, a_2, 0, \ldots, a_{14}, 0) + (0, a_1, 0, a_3, \ldots, 0, a_{15}) \ .$$

Let $\boldsymbol{a'_{odd}} = (a_1, 0, a_3, 0, \ldots, a_{15}, 0)$. Then we can perform the $\boldsymbol{a} \cdot c$ as three vector-scalar multiplications $\boldsymbol{a_{even}} \cdot c_0$, $\boldsymbol{a'_{odd}} \cdot c_1$, and $(\boldsymbol{a'_{odd}} + \boldsymbol{a_{even}}) \cdot (c_0 + c_1)$ over $\widetilde{\mathbb{F}}_{256}$ with Karatsuba's method. However, the 0's in $\boldsymbol{a_{even}}$ and $\boldsymbol{a_{odd}}$ reduce the efficiency of the SIMD multiplication in this implementation.

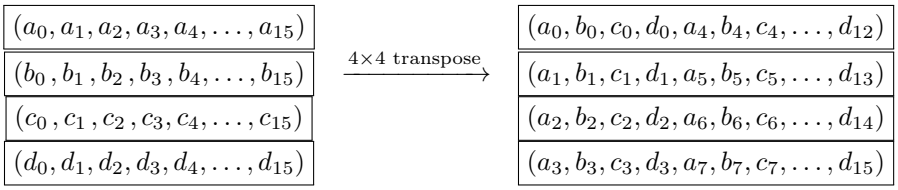
While multiplying two elements $[\boldsymbol{a}, \boldsymbol{b} \in \widetilde{\mathbb{F}}_{2^{128}}]$ by a subfield element $c \in \widetilde{\mathbb{F}}_{2^{16}}$, we can combine the even and odd bytes of $\boldsymbol{a}$ and $\boldsymbol{b}$ for better efficiency of the SIMD multiplication over $\widetilde{\mathbb{F}}_{256}$. Hence, we can use a 2×2 transposition, which converts $\boldsymbol{a} = (a_0, \ldots, a_{15})$ and $\boldsymbol{b} = (b_0, \ldots, b_{15})$ to $(a_0, b_0, a_2, b_2, \ldots, b_{14})$ and $(a_1, b_1, a_3, \ldots, b_{15})$, to split the even and odd bytes. The 2×2 transposition involves only 2 registers which can be efficiently performed on modern CPUs.

For multiplication by an element in $\widetilde{\mathbb{F}}_{2^{32}}$, we need to 4×4 transpose our data. The following Fig. 3.1 depicts the process of data rearrangement. The transpose can be implemented with the technique in Sec. 2.4.1.

3.3 Constant-time Multiplication in Binary Fields

In the cryptographic circumstance, loading a table with the address depended on a secret value is harmful since an attacker can reveal the secret value by

Figure 3.1: Data rearrangement for multiplication in $\widetilde{\mathbb{F}}_{2^{128}}$ by elements in $\widetilde{\mathbb{F}}_{2^{32}}$.



probing the difference of access-time of the cache line. Hence, how a table is loaded into registers is a critical issue while performing field multiplication with Alg. 1.

In this section, we provide two methods for constant-time multiplication, meaning the running-time is independent of the values of two multipliers. Sec. 3.3.1 describes the multiplication with log/exp tables. Sec. 3.3.2 describes how to generate multiplication tables for preventing loading the tables with the address depending on the values of multipliers.

3.3.1 Multiplication with Logarithm Tables

We use the logarithm and exponential tables for our first constant-time multiplication with the `PSHUFB` instruction. In the method, we can load the log/exp tables of a fixed address and lookup some entries of the table by the secret values with the `PSHUFB` instruction. Hence, we avoid loading a multiplication table with a secret value of the multiplier.

Table 3.1: Logarithm table for $\widetilde{\mathbb{F}}_{16}$.

`0x0`	`0x1`	`0x2`	`0x3`	`0x4`	`0x5`	`0x6`	`0x7`
−42	0	5	10	1	4	2	8
`0x8`	**`0x9`**	**`0xa`**	**`0xb`**	**`0xc`**	**`0xd`**	**`0xe`**	**`0xf`**
6	13	9	7	11	12	3	14

Table 3.1 gives the logarithm table for the field $\widetilde{\mathbb{F}}_{16}$. The table contains 16 entries, one byte for each, and stores in a fixed address in memory. To compute $a \cdot b$ for $a, b \in \widetilde{\mathbb{F}}_{16}$, we perform

$$a \cdot b = \exp_{\texttt{0x4}}((\log_{\texttt{0x4}} a + \log_{\texttt{0x4}} b) \bmod 15) \ .$$

To increase the efficiency with the method, we can store the field elements in logarithm form to avoid some table lookups.

Although the element 0 is not in the multiplicative group, we can set $\log 0 = -42$, and the `PSHUFB` ensures the correct products of 0 for querying negative values. The value -42 is affordable for 2 consecutive multiplications before it changes its sign or underflows the byte range.

For the multiplication in $\widetilde{\mathbb{F}}_{256}$, we perform the polynomial multiplication over $\widetilde{\mathbb{F}}_{16}$ and use the log/exp tables for $\widetilde{\mathbb{F}}_{16}$. Recall that

$$\widetilde{\mathbb{F}}_{256} := \widetilde{\mathbb{F}}_{16}[x_2]/(x_2^2 + x_2 + x_1 x_0).$$

As a polynomial multiplication over $\widetilde{\mathbb{F}}_{16}$, a time-constant $\widetilde{\mathbb{F}}_{256}$ multiplication costs about 3 $\widetilde{\mathbb{F}}_{16}$ multiplications with the Karatsuba method. Here we can use the multiplication table for reducing the degree-2 term since multiplying elements by the known x_1x_0 leak no side-channel information with `PSHUFB`.

3.3.2 Generating Multiplication Tables On-the-fly

In this section, we describe how to generate the multiplication tables for particular variables efficiently. Instead of loading tables with secret values, we can produce the desired multiplication tables for the Alg. 1. We can store the table with the indexes of the variables containing secret values, and thus avoid the leakage of the values from loading the table by the values. In other words, we transform the memory access indexed by a secret value to sequential access by the index of variables to prevent leakage of side-channel information.

Generating multiplication tables for elements in $\widetilde{\mathbb{F}}_{16}$

In this section, we present the method to generate 16 multiplication tables for $\mathbf{w} = (w_0, w_1, \ldots, w_{15}) \in \mathbb{F}_{16}^{16}$, i.e., $(w_0 \cdot \texttt{0x0}, \ldots, w_0 \cdot \texttt{0xf}), \ldots, (w_{15} \cdot \texttt{0x0}, \ldots, w_{15} \cdot \texttt{0xf})$ with the SIMD instruction set.

Algorithm 2: Generating multiplication tables for $\widetilde{\mathbb{F}}_{16}$.

```
1 multab_16( w) :
  input : w = (w_0, w_1, ..., w_15) ∈ F̃_16^16.
  output: 16 multiplication tables for (w_0, ..., w_15).
2 Let TAB_w be a 16 × 16 byte-matrix for the 16 tables.
3 for i ← 0 to 15 do
4 |   TAB_w[i] ← PSHUFB(multiplication table for i, w) .
5 end
6 return transpose_16x16(TAB_w) .
```

Algorithm 2 shows the generation of tables for elements in $\widetilde{\mathbb{F}}_{16}$. In a nutshell, we first multiply $\mathbf{w}$ by all 16 elements in $\widetilde{\mathbb{F}}_{16}$. After the multiplications, all the products of $\mathbf{w}$ and all possible elements in $\widetilde{\mathbb{F}}_{16}$ are generated in 16 registers. However, we have to collect the desired product in the correct position. By a 16×16 transpose of byte matrix, we can generate the multiplication tables for $\mathbf{w}$. We can perform the matrix transpose with the method in Sec. 2.4.1. Figure3.2 illustrates the process.

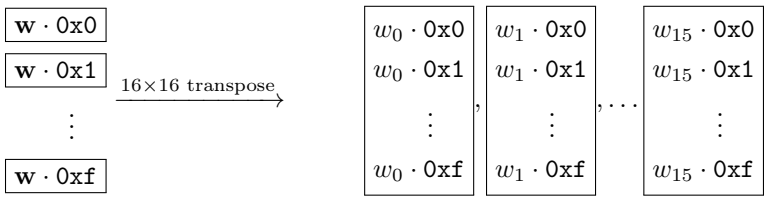
Generating multiplication tables for elements in $\widetilde{\mathbb{F}}_{256}$

We describe the method to generate multiplication tables for elements in $\widetilde{\mathbb{F}}_{256}$ in this section. Recall that a $\widetilde{\mathbb{F}}_{256}$ element is a degree-1 polynomial over $\widetilde{\mathbb{F}}_{16}$

$$\widetilde{\mathbb{F}}_{256} := \widetilde{\mathbb{F}}_{16}[x_2]/(x_2^2 + x_2 + x_1x_0).$$

We can generate multiplication tables for $\widetilde{\mathbb{F}}_{256}$ elements with the multiplication tables of two $\mathbb{F}_{16}$ coefficients.

Figure 3.2: Generating multiplication tables for $\mathbf{w} = (w_0, w_1, \dots w_{15})$.



After computing $\mathbf{w} \cdot \texttt{0x0}$, $\mathbf{w} \cdot \texttt{0x1}$, ..., $\mathbf{w} \cdot \texttt{0xf}$, each row stores one product, and the columns are the desired multiplication tables. We can thus generate the tables of w_0, w_1, ...,w_{15} by collecting data in columns.

Let $a \in \widetilde{\mathbb{F}}_{256}$ be the element that we want to generate the multiplication table. Suppose the polynomial form of a over $\widetilde{\mathbb{F}}_{16}$ is $a(x) = a_0 + a_1 x \in \widetilde{\mathbb{F}}_{16}[x]$ and $\texttt{TAB}_{a_0}$ and $\texttt{TAB}_{a_1}$ are the tables for multiplying a_0 and a_1 in $\widetilde{\mathbb{F}}_{16}$. Assume we multiply an arbitrary $b(x) = b_0 + b_1 x \in \widetilde{\mathbb{F}}_{16}[x]$ by $a(x)$:

$$a(x) \cdot b(x) = (a_0 + a_1 x) \cdot b_0 + (a_0 x + a_1 x^2) \cdot b_1 \ .$$

So we can calculate the low-bits part(corresponding to b_0) of the multiplication table of a by

$$\texttt{TAB}_{low}(a) = \texttt{TAB}_{a_0} + (\texttt{TAB}_{a_1} << 4) \ . \tag{3.6}$$

Since $x^2 = x + x_1 x_0$, we can calculate the high-bits part(corresponding to b_1) of the table by

$$\texttt{TAB}_{high}(a) = ((\texttt{TAB}_{a_0} + \texttt{TAB}_{a_1}) << 4) + (\texttt{TAB}_{a_1} \cdot x_1 x_0) \ . \tag{3.7}$$

The two tables $\texttt{TAB}_{low}(a)$ and $\texttt{TAB}_{high}(a)$ are the two inputs in Alg. 1 for multiplying elements by a in $\widetilde{\mathbb{F}}_{256}$.

3.4 Multiplication in $\mathbb{F}_{2^m}$ for $m = 64$ and 128

In this section, we describe the implementations of multiplication in $\mathbb{F}_{2^{64}}$ and $\mathbb{F}_{2^{128}}$ with the `PCLMULQDQ` instruction, i.e., the hardware instruction for multiplying Boolean polynomials. Since the hardware Boolean polynomial multiplication, the implementation simplifies the programming burden on multiplying polynomials.

For multiplication in $\mathbb{F}_{2^{64}}$, Lemire and Kaser [LK16] presented an efficient method under the representation

$$\mathbb{F}_{2^{64}} := \mathbb{F}_2[x] / \left(x^{64} + x^4 + x^3 + x + 1\right) \ .$$

Algorithm 3 shows the processes for multiplying inputs in $\mathbb{F}_{2^{64}}$. In the implementation, one `PCLMULQDQ` multiplies 2 Boolean polynomials of degree 63 to a polynomial of degree 126, and then the other `PCLMULQDQ` reduces the terms of

Algorithm 3: Multiplication in $\mathbb{F}_{2^{64}}$

```
mul_64( a, b) :
  input : a = a_0 + a_1 x + ··· + a_63 x^64 ∈ F_2^64
          b = b_0 + b_1 x + ··· + b_63 x^64 ∈ F_2^64 .
  output: c = a · b ∈ F_2^64 .
d ← PCLMULQDQ(a, b) ∈ F_2[x].
Let d = e + f' where deg e < 64 and degrees of terms in f' ≥ 64.
Let f = f'/x^64.
g ← PCLMULQDQ(f, ω(0x1b)) .
Let g = l + h' where deg l < 64 and and degrees of terms in h' ≥ 64.
Let h = h'/x^64.
r ← PSHUFB(0x1b, h) .
return e + l + r .
```

degree 64 to 126 back to a remainder polynomial of degree 66. One `PSHUFB` finishes the reduction for the terms of degree 64 to 66.

For multiplication in $\mathbb{F}_{2^{128}}$, we choose the same representation as AES-GCM:

$$\mathbb{F}_{2^{128}} := \mathbb{F}_2[x]/\left(x^{128} + x^7 + x^2 + x + 1\right) \ .$$

Algorithm 4 shows the processes for multiplication in $\mathbb{F}_{2^{128}}$. In our implementa-

Algorithm 4: Multiplication in $\mathbb{F}_{2^{128}}$

```
mul_128( a, b) :
  input : a = a_0 + a_1 x + ··· + a_127 x^127 ∈ F_2^128
          b = b_0 + b_1 x + ··· + b_127 x^127 ∈ F_2^128 .
  output: c = a · b ∈ F_2^128 .
Let a = a_l + a_h where deg a_l < 64 and degrees of terms in a_h ≥ 64.
a_h ← a_h/x^64.
Let b = b_l + b_h where deg b_l < 64 and degrees of terms in b_h ≥ 64.
b_h ← b_h/x^64.
c_0 ← PCLMULQDQ(a_l, b_l).
c_2 ← PCLMULQDQ(a_h, b_h).
c_1 ← PCLMULQDQ(a_h + a_l, b_h + b_l) + c_0 + c_1.
Let c_2 = c_2l + c_2h where deg c_2l < 64 and degrees of terms in c_2h ≥ 64.
c_2h ← c_2h/x^64.
c_1 ← c_1 + PCLMULQDQ(c_2h, ω(0x87)).
Let c_1 = c_ll + c_1h where deg c_ll < 64 and degrees of terms in c_1h ≥ 64.
c_2l ← c_2l + c_1h/x^64.
return c_0 + c_1l · x^64 + PCLMULQDQ(c_2l, ω(0x87)) .
```

tion, the multiplication costs 5 `PCLMULQDQ`(3 for multiplying 128-bit polynomials with Karatsuba's method and 2 for reducing the 256-bit result back to 128 bits with linear folding). More implementations of multiplications in $\mathbb{F}_{2^{128}}$ can be found in [GK14].

Chapter 4

Erasure Correcting Codes for RAID

Through the codes, we show the general technique of multiplying elements in binary fields with the SIMD instruction set. This chapter is based on the joint work with Bo-Yin Yang and Chen-Mou Cheng published in [CYC13].

4.1 Introduction

Code and Storage systems

A Redundant Array of Independent Disks(RAID) has been a mainstream technology to combine multiple disks into a massive storage system and increase the reliability of storage systems at a cost of using redundant disks for storing checksums The capability of RAID systems depends on its underlying erasure correcting code(ECC) to correct disk or sector failures. A code is a collection of codewords, which comprises data elements and checksum (redundant) elements. An erasure is a missing element in a codeword with known positions, and an erasure correcting code is a code designed for correcting erasures.

Although the mappings between code elements and physical components in a RAID can be complicated, we assume each element of a codeword is associated with a physical disk in this thesis. In this basic mapping model, a codeword comprises a stripe of data came from the same positions among all drives, and an erasure corresponds to the failure of a particular disk. Since disks usually work in units of sectors, the model also results in natural data parallelism for processing multiple independent codewords simultaneously. In this these, we only assume the basic model since we focus more on the efficiency of coding and the possible length of a codeword rather than its mapping to real physical devices.

The Code in Standard RAID-6 Storage Systems

In 1997, Plank [Pla97] presented a code, termed "Plank's code," for storage systems. The code is currently the de facto code of RAID-6 in the implementation of Linux kernel [Anv11]. In brief, a Plank's code is a systematic code, in which the data elements of the codeword are the same with real data and checksums

are generated by multiplying the data contents with a Vandermonde matrix. For example of the RAID-6, which is capable for recovering 2 disk failures, the first checksum is simply the `XOR` of all data symbols, and the second checksum accumulates the results of data multiplying the specified element '`0x2`' in the `0x11d` $\mathbb{F}_{256}$. Anvin [Anv11] presented an efficient implementation for generating the second checksum with the SSSE3 instruction set, and it became the default implementation in Linux kernel.

Plank's code was sometimes confused with the famous Reed-Solomon code, for which a Vandermonde matrix generates the *whole* codeword. Compared with applying Vandermonde matrix to only the checksum parts, Plank's code turns out to be *neither identical nor isomorphic* to any Reed-Solomon code.

4.1.1 The Problem of Plank's Code

In the language of code, the design of Plank's code is hard to satisfy the maximum distance separable(MDS) property when the length n of a codeword is large. For storage systems, we usually expect an underlying MDS code, which can correct the same number of erasures as the number of checksums. When we check the length n of a codeword for Plank's code under the MDS requirement, we have $n = 255, 255$, and unfortunate 27 for 2,3, and 4 checksums. The consequence is that some data might be unrecoverable for some failures of 4 disks if the total number of drives is more than 27.

The defect of Plank's code was identified several times in the literature, and the usual solution suggests to abandon Plank's code. Lancan and Fimes [LF04] pointed out that a systematic code with checksums generated by a Vandermonde matrix is not necessarily an MDS code for an arbitrary codeword length. In fact, Plank himself discovered and corrected this unfortunate error in a subsequent note [PD05]. However, they fixed the issues by simply proposing other codes.

4.1.2 Our Solution: the RAIDq Code

We propose RAIDq, a new way to *extend* Plank's codes, to deal with the MDS issue to some extent. In other words, we *still* use generalized Plank's code for building RAIDs. In RAIDq, one can build a reasonably large RAID system by carefully choosing checksums in the field $\widetilde{\mathbb{F}}_{256^2}$ instead of the original $\mathbb{F}_{256}$. The $\widetilde{\mathbb{F}}_{256^2}$ is represented as polynomials over original $\mathbb{F}_{256}$ in the RAID-6. While we recommend $n = 96$ for the performance, in fact, the maximum n of MDS can be 168 with four checksums in RAIDq.

Although the MDS problem, of course, can be solved simply by changing to another MDS code, the advantage of the RAIDq is the backward compatibility to Plank's code for RAID-6 in both the data and performance aspects. For the compatibility of data, we would ideally expect to upgrade an existing RAID system by inserting a new checksum disk for better data integrity protection without rebuilding the entire system. The construction of the $\widetilde{\mathbb{F}}_{256^2}$ keeps the values of elements in $\mathbb{F}_{256}$ unchanged while treating all codewords in $\widetilde{\mathbb{F}}_{256^2}$ and thus makes the existing contents of disks unchanged.

To maintain the remarkable performance of the RAID-6 is another reason for the backward compatible code. To keep the current performance, we maintain the generation of 3 checksums unchanged in $\mathbb{F}_{256}$. Furthermore, the 4-th checksum is designed to be an element for fast multiplication in $\widetilde{\mathbb{F}}_{256^2}$.

With the construction of the working field $\widetilde{\mathbb{F}}_{256^2}$, we extend the current code in RAID-6 for a reasonably large system with 4 checksums while maintaining its original advantage of high performance and removing the penalty for rebuilding original checksums

4.1.3 Chapter Overview

Section 4.2 describes the details of Plank's code, and we introduce the RAIDq code in section 4.2.6. Section 4.3 describes the encoding and decoding process as well as their high-performance implementations. We refer the reader to Sec. 3.1 for the preliminaries of the fast arithmetic in finite fields. Section 4.4 shows the benchmarks of RAIDq with the arithmetic of fields. Section 4.5 summarizes the chapter.

4.2 Extending RAID-6 for More Checksums

4.2.1 Terminology

An (n, k) code $\mathcal{C}$ over an algebraic structure denotes the set of all codewords $\boldsymbol{c}$'s. Let $\mathbb{F}_q$ be a finite field with q elements. A codeword $\boldsymbol{c} \in \mathcal{C}$ denotes a column vector of dimension n over $\mathbb{F}_q$, i.e.,

$$\boldsymbol{c} = [c_0, \ldots, c_{n-1}]^T \in \mathbb{F}_q^n$$

or a column matrix $\mathbb{F}_q^{n\times 1}$. The n blocks of $\boldsymbol{c}$ comprise k information blocks and $m = n - k$ redundant blocks. Let $\boldsymbol{d} = [d_0, \ldots, d_{k-1}]^T \in \mathbb{F}_q^k$ denote the k blocks of data and $\boldsymbol{p} = [p_0, \ldots, p_{m-1}]^T \in \mathbb{F}_q^m$ be the m redundant blocks in the codeword $\boldsymbol{c}$. For practical and efficiency reasons, we only consider systematic codes, which the data blocks $\boldsymbol{d}$ remain unchanged after encoding. The encoding process generates the checksums $\boldsymbol{p}$ with the input of $\boldsymbol{d}$. Hence, a codeword can be represented as

$$\boldsymbol{c} = [\boldsymbol{d}, \boldsymbol{p}] = [d_0, \ldots, d_{k-1}, p_0, \ldots, p_{m-1}]^T \in \mathbb{F}_q^n \ .$$

The minimal distance d denotes the minimum Hamming distance between any two codewords in the code $\mathcal{C}$ and the code may also be denoted as (n, k, d) code. It can even describe the capability of erasure correction. A code is denoted as Maximum Distance Separable(MDS) code if $d > n - k$, which providing tolerance of any $n - k$ erasures.

In this chapter, we use two specific representations of binary fields in Eq. (2.4)

$$\mathbb{F}_{256} = \mathbb{F}_{2^8} := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x^2 + 1) \qquad \text{(2.4 revisited)}$$

and (2.10)

$$\widetilde{\mathbb{F}}_{256^2} := \mathbb{F}_{256}[X]/(X^2 + \omega_{\mathbb{F}_{256}}(\texttt{0x8}) \cdot X + 1) \ . \qquad \text{(2.10 revisited)}$$

Since the fields are consistent in this chapter, we usually omit the 'ω' symbols while denoting field elements with hex numbers for abbreviation. For example, the symbol '`0x2`" represents the field element 'x' in $\mathbb{F}_{256}$ or $\widetilde{\mathbb{F}}_{256^2}$. The symbol '`0x100`" represents the field element 'X' in $\widetilde{\mathbb{F}}_{256^2}$. The symbol '`0x10801`" represents the irreducible polynomial $(X^2 + \omega_{\mathbb{F}_{256}}(\texttt{0x8}) \cdot X + 1)$ in $\mathbb{F}_{256}[X]$.

Another important convention in this chapter is the symbol 'a.' For the consistent of symbols in the literature, the 'a' represents the specific element '`0x2`' in $\mathbb{F}_{256}$.

4.2.2 Plank's code

The erasure correcting code of RAID-6 in Linux kernel [Anv11] has been heavily influenced by Plank [Pla97]:

Definition 2. A *Plank's code* is a linear (n, k) systematic code over $\mathbb{F}_q$ whose generator matrix $G \in \mathbb{F}_q^{n\times k}$ is of the (systematic) form

$$G = \begin{bmatrix} I_k \\ V \end{bmatrix}.$$

Here $V \in \mathbb{F}_q^{m\times k}$ is a Vandermonde matrix

$$\mathrm{Van}(\alpha_0, \ldots, \alpha_{m-1}; k) := \begin{bmatrix} \alpha_0^0 & \alpha_0^1 & \cdots & \alpha_0^{k-1} \\ & \vdots & & \\ \alpha_{m-1}^0 & \alpha_{m-1}^1 & \cdots & \alpha_{m-1}^{k-1} \end{bmatrix},$$

where α_i's are distinct nonzero elements in $\mathbb{F}_q$, and $I_k \in \mathbb{F}_q^{k\times k}$ the identity matrix. This will be termed the *Plank's code generated by checksum generator* $[\alpha_0, \ldots, \alpha_{m-1}]$. Usually we set $\alpha_0 := 1$. When $\alpha_i := \alpha^i$ for some fixed α, we call it the *Plank's code generated by* α (of length $n = k + m$ over $\mathbb{F}_q$).

Encoding of data blocks $\boldsymbol{d} \in \mathbb{F}_q^k$ to a codeword $\boldsymbol{c} \in \mathbb{F}_q^n$ is the matrix multiplication $\boldsymbol{c} = G \cdot \boldsymbol{d}$. The upper part of $\boldsymbol{c}$ is the same with $\boldsymbol{d}$ from I_k in G (systematic). Decoding is related to recovery $\boldsymbol{d}$ from a codeword $\boldsymbol{c}$ with some missing elements (erasures) or from a corrupted $\boldsymbol{c}$ (containing errors).

With $\alpha_i := \alpha^i$ for a generator $\alpha \in \mathbb{F}_q$, Plank's code would allow very efficient encoding, erasure decoding, and even error correction. For $m = 1$ or 2 and $q = 256$, they are *exactly* the codes used in RAID-5 and 6, as implemented and discussed by Anvin [Anv11]. In the case of RAID-6, Plank's code is MDS for n up to 255, i.e., a $(255, 253, 3)$ code.

4.2.3 About MDS property of Plank's code

Lancan and Fimes [LF04] mentioned that a code with generator matrix $G = [I_k|A]^T$, where A is a $(n - k) \times k$-matrix, is MDS if and only if all square submatrices of A are nonsingular. A minimum distance of a code is equivalent to minimum nonzero elements among all possible codewords because the difference of codewords is still a codeword in a linear code. If we find a singular submatrix of size $m' \times m'$ in the matrix A, we can create a column with all zero elements in the selected submatrix by the linear combination of selected columns of G. The corresponding codeword of the linear combined columns of G contains at most m' elements in upper I_k part and $m-m'$ elements in A part, and $m'+(m-m') = m < m + 1$, which is not a MDS code.

For checking the maximum n of MDS Plank's code with m checksums, we enumerate all submatrices of $\mathrm{Van}(\alpha_0, \ldots, \alpha_{m-1}; k)$ and increase k until we find a singular submatrix. The maximum $n = k + m$ without a singular submatrix is the maximum size of MDS Plank's code.

4.2.4 Plank's Code for RAID: the Successful Cases

As proposed by Plank [Pla97] and implemented in Linux kernel by Anvin [Anv11], Plank's code for RAID-6 is over `0x11d` $\mathbb{F}_{256}$ with one checksum generator 'a,' which is also a primitive generator $a :=$ `0x02` for $\mathbb{F}_{256}$. In general, the checksum generator is of the form of $[a^0, a^1, \ldots, a^{m-1}]$. Applying to $m = 2$ in RAID-6, it is $[1, a]$, resulting in a $(255, 253, 3)$ code. For more details, the starting [1] represents the first `XOR` checksum $p_0 = \sum_{i=0}^{k} d_i$, which is the checksum in RAID-5, and $[a]$ represents the second checksum $p_1 = \sum_{i=0}^{k} a^i \cdot d_i$, which can be efficiently calculated with Horner's rules.

Following the same form of checksum generators in RAID-6, one can build a Plank's code by $[1, a, a^2]$ resulting in a $(255, 252, 4)$ MDS code. Note that all elements a and a^2 are elements with fast multiplication in $\mathbb{F}_{256}$ from Sec. 3.1.2. Another choice of the generators for 3 checksums is $[1, a, a^{\frac{1}{2}}]$, where $a^{\frac{1}{2}} = a^{128} =$ `0x85`, resulting in a same $(255, 252, 4)$ MDS code. This choice also satisfies the definition of one generator since $(a^{\frac{1}{2}})^2 = a$.

4.2.5 Plank's Code for RAID: the 4th Checksums

Following the same construction in RAID-6, the checksum generators are $[1, a, a^2, a^3]$ or $[1, a, a^{\frac{1}{2}}, a^{\frac{3}{2}}]$ for Plank's code with 4 checksums. However, both settings result in a short $(25, 21, 5)$ code if the MDS property ($d \geq 5$) is required. This construction might not be able to recover data blocks from redundant blocks in some patterns of erasures when the number of data blocks is over 21.

For building a large RAID system, we have searched all Plank's codes of $[1, a, a^i, a^j]$ for $i \neq j$, constructions, which maintains the compatibility with Plank's code in RAID-6, and confirm that the same $(25, 21, 5)$ code is indeed the best we can do. Without the compatibility with RAID-6, we have further searched the Plank's codes of $[a^i, a^j, a^k, a^l]$ for $i \neq j \neq k \neq l$, and found a maximum $(37, 33, 5)$ code over $\mathbb{F}_{256}$.

4.2.6 RAIDq with 4 Checksums

To fix the MDS issue in RAID systems of Plank's code, we present RAIDq, a new extending method of Plank's code. We tried constructing the RAIDq code over the field $\widetilde{\mathbb{F}}_{256^2}$ after knowing that it is impossible to build an MDS Plank's code in $\mathbb{F}_{256}$ for over 33+4 disks. The field $\widetilde{\mathbb{F}}_{256^2}$ is an extension field from original $\mathbb{F}_{256}$ and thus maintains the backward compatibility in data. Hence, we searched the checksum generators in the elements for fast multiplication from Sec. 3.1.3 in $\widetilde{\mathbb{F}}_{256^2}$.

The recommended checksums of RAIDq is $[1, a, a^{\frac{1}{2}}, X]$ over the $\widetilde{\mathbb{F}}_{256^2}$, resulting in a $(96, 92, 5)$ code. In this construction, we give up using *one* α for generating all checksum generators as α^i. However, three out of the four checksums are actually in $\mathbb{F}_{256}$, and the construction is compatible with RAIDs of 3 checksums over $\mathbb{F}_{256}$. Furthermore, the choice of the fourth checksum X is the element with fast multiplications in $\widetilde{\mathbb{F}}_{256^2}$ from Sec. 3.1.3.

We can further investigate the general choices of checksum generators in a Plank's code for a larger code length of MDS. By exhaustive search, we list the good candidates in Table 4.1 for codes in the form of $[1, a, a^{\frac{1}{2}}, \alpha_3]$ and

$[1, a, a^2, \alpha_3]$ for $\alpha_3 \in \widetilde{\mathbb{F}}_{256^2}$. All parameters are selected for maintaining backward compatible to RAID-6 and minimizing the computation cost by choosing the elements with fast multiplication in Section 3.1.3 as possible. The maximum code length is $164 + 4$ for a code in the above forms. For construction with faster multiplication, the maximum code length is $143 + 4$, generated by $[1, a, a^{\frac{1}{2}}, a^2X + 1]$.

Table 4.1: Good candidates for RAIDq 8 in $\widetilde{\mathbb{F}}_{256^2}$

Checksum generators	Code length n	binary rep. of 4th checksum
$[1, a, a^{1/2}, X]$	$92 + 4$	0x0100
$[1, a, a^{1/2}, X + 1]$	$107 + 4$	0x0101
$[1, a, a^{1/2}, a^2X + 1]$	$143 + 4$	0x0401
$[1, a, a^{1/2}, a^{115}X]$	$151 + 4$	0x7C00
$[1, a, a^{1/2}, a^{141}X]$	$151 + 4$	0x1500
$[1, a, a^{1/2}, a^{85}X + a^{28}]$	$164 + 4$	0xD618
$[1, a, a^{1/2}, a^{85}X + a^{160}]$	$164 + 4$	0xD6E6
$[1, a, a^{1/2}, a^{186}X + a^6]$	$164 + 4$	0x6E40
$[1, a, a^{1/2}, a^{186}X + a^{129}]$	$164 + 4$	0x6E17
$[1, a, a^2, X]$	$55 + 4$	0x0100
$[1, a, a^2, a^3X]$	$107 + 4$	0x0800
$[1, a, a^2, aX + 1]$	$113 + 4$	0x0201
$[1, a, a^2, a^{111}X]$	$143 + 4$	0xCE00
$[1, a, a^2, a^{146}X]$	$143 + 4$	0x9A00
$[1, a, a^2, a^{120}X + a^5]$	$164 + 4$	0x3B20
$[1, a, a^2, a^{120}X + a^{169}]$	$164 + 4$	0x3BE5
$[1, a, a^2, a^{173}X + a^{58}]$	$164 + 4$	0xF669
$[1, a, a^2, a^{173}X + a^{222}]$	$164 + 4$	0xF68A

4.3 Implementation

We describe the encoding and decoding techniques for PAIDq code in this section.

4.3.1 Encoding and decoding Plank's codes

Since Plank's code is a systematic code, we define `Encode` : $\mathbb{F}_q^k \mapsto \mathbb{F}_q^m$. The `Encode` computes the m checksum blocks with matrix-vector multiplication

$$\begin{bmatrix} \alpha_0^0 & \alpha_0^1 & \cdots & \alpha_0^{k-1} \\ & \vdots & & \\ \alpha_{m-1}^0 & \alpha_{m-1}^1 & \cdots & \alpha_{m-1}^{k-1} \end{bmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{k-1} \end{bmatrix} = \begin{bmatrix} p_0 \\ \vdots \\ p_{m-1} \end{bmatrix} .$$

After completing `Encode`, a full codeword comprises data blocks and checksums $\boldsymbol{c} = [\boldsymbol{d}, \boldsymbol{p}]^T$. As for data updating, only the difference of updating parts are "encoded" for the difference of checksums between the original and updated codewords.

We can also generate the checksums by evaluating a polynomial whose coefficients correspond to the data at the points of checksum generators as:

$$p_i = \sum_{j=0}^{m-1} \alpha_i^j d_j \quad \text{for } i = 0, \ldots, m-1.$$

When evaluating polynomials by Horner's rule

$$p_i = d_0 + \alpha_i(d_1 + \alpha_i(d_2 + \cdots)), \tag{4.1}$$

we calculate the checksum by accumulating the product of multiplication by some *fixed elements* α_i, allowing the acceleration while multiplying elements with fast multiplication in Sec. 3.1.2.

4.3.2 Implementing the Encoder

For generating checksums in the RAIDq with Honer's rule(Eq. (4.1)), we can implement the `Encode` with the multiplications by some specific elements. Hence, we can optimize the encoder of the RAIDq with fast multiplications for elements of a,a^2, and X.

Recall the GF Arithmetic

We first review the fast multiplications in $\mathbb{F}_{256}$ and $\widetilde{\mathbb{F}}_{256^2}$ for the self-contain of this chapter.

- We perform the general multiplication in $\mathbb{F}_{256}$ with Eq. (3.1). It costs 2 `PSHUFB` instructions in SIMD instruction set with Algo. 1.
- We multiply $b \in \mathbb{F}_{256}$ by $a := x \in \mathbb{F}_2[x]$ in $\mathbb{F}_{256} \cong \mathbb{F}_2[x]_{<8}$ with Eq. (3.2)

  $$b(x) \cdot x \to (b_6 x^7 + \cdots + b_0 x) + b_7 \cdot \omega(\texttt{0x1d}) \ .$$

  It costs roughly one `PSHUFB` instruction.
- We multiply $b \in \mathbb{F}_{256}$ by $a^2 := x^2 \in \mathbb{F}_2[x]$ in $\mathbb{F}_{256} \cong \mathbb{F}_2[x]_{<8}$ in a similar way of Eq. (3.3)

  $$b(x) \cdot x^2 \to (b_5 x^7 + \cdots + b_0 x^2) + (b_7 x + b_6) \cdot (x^4 + x^3 + x^2 + 1) \ .$$

  It costs roughly one `PSHUFB` instruction.
- We perform the general multiplication between $a(X) \in \mathbb{F}_{256}[X]$ and $b(X) \in \mathbb{F}_{256}[X]$ in $\widetilde{\mathbb{F}}_{256^2} \cong \mathbb{F}_{256}[X]_{<2}$ with Eq. (3.4)

  $$\begin{aligned}&(a_1 X + a_0) \cdot (b_1 X + b_0) \bmod \texttt{0x10801} \\ &\to [\omega_{\mathbb{F}_{256}}(\texttt{0x8}) \cdot a_1 b_1 + a_1 b_1 + a_0 b_0 + (a_0 + a_1)(b_0 + b_1)]\, X + a_1 b_1 + a_0 b_0 \ .\end{aligned}$$

  It costs roughly 7 `PSHUFB` instructions.
- We multiply $b \in \widetilde{\mathbb{F}}_{256^2}$ by $X \in \mathbb{F}_{256}[X]$ in $\widetilde{\mathbb{F}}_{256^2} \cong \mathbb{F}_{256}[X]_{<2}$ with Eq. (3.5)

  $$\begin{aligned} b(X) \cdot \omega_{\mathbb{F}_{256^2}}(X) &= (b_1 X + b_0) \cdot X \bmod \texttt{0x10801} \\ &\to (b_0 + \omega(\texttt{0x8}) \cdot b_1) X + b_1 \ . \end{aligned}$$

  It costs roughly one `PSHUFB` instruction and some data movements.

Comments about SIMD

Since we multiply the data by checksum generators in the SIMD instruction set, we evaluate multiple polynomials of Eq. (4.1) in parallel. In other words, we encode various codewords simultaneously, and the order of parallelism depends on the width of used SIMD instruction sets.

Further, we adopt the data structure of $\widetilde{\mathbb{F}}_{256^2}$ in section 3.1.3 that separate the high and low bytes of elements in $\widetilde{\mathbb{F}}_{256^2}$ for avoiding the rearrangement the data in advance.

Generating the $[a^{\frac{1}{2}}]$ Checksum

Besides the elements with fast multiplication, we have to overcome the generation of the 3rd checksum $[a^{\frac{1}{2}}]$ since $a^{\frac{1}{2}} = \texttt{0x85}$ is not an element with fast multiplication in $\mathbb{F}_{256}$ in Sec. 3.1.2. Instead of multiplying by `0x85` with the general multiplication in $\mathbb{F}_{256}$, we suggest a different process for generating the $[a^{\frac{1}{2}}]$ checksum by dividing the calculation into two buffers of RAID-6 checksum computations:

$$\begin{aligned} p_3 &= \textstyle\sum_{i=0,1,\cdots} a^{\frac{i}{2}} \cdot d_i \\ &= \textstyle\sum_{i=0,2,4,\cdots} a^{\frac{i}{2}} \cdot d_i + a^{\frac{1}{2}} (\sum_{i=1,3,5,\cdots} a^{\frac{(i-1)}{2}} \cdot d_i) \\ &= \textstyle\sum_{i=0,1,2,\cdots} a^{i} \cdot d_{2i} + a^{\frac{1}{2}} (\sum_{i=0,1,2,\cdots} a^{i} \cdot d_{2i+1}) \ . \end{aligned} \tag{4.2}$$

With this method, we only perform the multiplication by $a^{\frac{1}{2}}$ once for generating the checksum $[a^{\frac{1}{2}}]$ at the expense of extra butter. After amortizing the cost of one $\times a^{\frac{1}{2}}$ to all data blocks, the cost of computing the $[a^{\frac{1}{2}}]$ checksum is similar to $[a]$.

4.3.3 Erasure Decoder

To decode erasures, we use syndrome decoding for reusing optimized encoder. To reuse the highly optimized encoder while decoding, we replace the missing data symbols with 0 and encode them again to get a set of new checksums. The differences between original and new checksums are named syndromes. We can recover the missing symbols by solving the linear system composed of the syndromes and part of the generating matrix corresponding to the positions of the missing symbols.

For example, suppose we want to decode two erasures d_i, d_j at known positions i and j. We first encode $\boldsymbol{d'} = [d_0, \ldots, d'_i = 0, \ldots, d'_j = 0, \ldots, d_{k-1}]^T$ for new checksums

$$\boldsymbol{p'} = [p'_0, \ldots, p'_{m-1}]^T = \texttt{Encode}(\boldsymbol{d'}) \ .$$

We denoted the syndrome $\boldsymbol{r} = [r_0, \ldots, r_{m-1}]^T \in \mathbb{F}_q^m$ as the difference of checksums $\boldsymbol{p'} - \boldsymbol{p} = [p'_0 - p_0, \ldots, p'_{m-1} - p_{m-1}]^T$. The erasure $[d_i, d_j]^T$ can be solved by the following linear relations:

$$\begin{bmatrix} \alpha_0^i & \alpha_0^j \\ \vdots & \\ \alpha_{m-1}^i & \alpha_{m-1}^j \end{bmatrix} \cdot \begin{bmatrix} d_i \\ d_j \end{bmatrix} = \begin{bmatrix} r_0 \\ \vdots \\ r_{m-1} \end{bmatrix} = \begin{bmatrix} p'_0 - p_0 \\ \vdots \\ p'_{m-1} - p_{m-1} \end{bmatrix} .$$

For small RAID systems, it is possible to enumerate all erasure modes, and hence we can prepare all possible inverse matrices to avoid computing them on the fly in decoding.

Due to the optimized `Encode`, the syndrome decoder can even perform better than other decoding methods which costs the same amount of multiplications as the encoder. Without the syndrome decoder, as described in [Riz97, ASI08], a simple way for decoding erasures is to strike out the corresponding rows of missing elements from the generating matrix and then multiply the remaining code symbols to the inverse of the remaining square matrix. Besides the inverse of the matrix is computed in run-time, one has to multiply the remaining data elements and checksums by up-predictable scalars in $\mathbb{F}_q$ which is usually done with general multiplication. We can observe the optimized multiplication in the `Encode` outperform the general multiplication from the experiments in the following section.

4.4 Experiments and Discuss

We benchmark coders for the RAID in this section. Through the experiments, we show the remarkable performance of the RAIDq.

4.4.1 The Experiment

We benchmark several encoders and decoders for the RAID in Table 4.2, including a Reed-Solomon(RS) coder and several parameters of RAIDq. The experiment measures the throughputs of memory reads and writes for en/decoding codewords of 64 data and m checksums. *Here, the decoding throughput is measured with maximal erasures in the data blocks and hence represents a worst-case scenario.* The inputs of a coder are $64+m$ pointers, typically pointing to a 4KB memory page, and each pointer corresponds to one particular position in the codeword. Hence, we implement the coders to process massive codewords simultaneously. In one call, the coder reads the data blocks, encodes the data, and writes the results to the checksum blocks.

We experiment on an Intel Xeon E3-1245 v3 processor (supporting the AVX2 instruction set) running at 3.40 GHz, and all coders are implemented with the AVX2 instruction set.

4.4.2 Results

In table 4.2, the throughput measures the average total amount of memory reads/writes in one second. Since the experiment are performed for codewords of $64 + m$ elements, the ratio between reads and writes is always $64/m$.

The rows of "naive op." show the performances of reading the inputs, accumulating them with basic bit operations(`XOR`, `OR`, `AND`, and `ANDNOT`), and then writing the results to destinations. It is not a real coder but represents the maximum capability of the CPU for processing the data.

The Reed-Solomon(RS) in the table is a systematic Reed-Solomon(RS) coder(in the BCH view), implemented with similar techniques of the RAIDq code. The codewords of the RS code is over $\mathbb{F}_{256}$, and each codeword is encoded to be the polynomial that is a multiple of the generator polynomial $g(x) := (x - 1)(x -$

Table 4.2: Throughputs (GB/s) of the coders for RAID with 64 data blocks.

Code	m	max. n	Encode r/w	Decode r/w
`XOR` (RAID-5)	1	255	51.5/0.8	
Naive op.	2		44.7/1.4	
$[1, a]$ (RAID-6)	2	255	21.6/0.7	20.3/0.6
Reed-Solomon	2	255	12.1/0.4	10.8/0.3
Naive op.	3		34.5/1.6	
$[1, a, a^{\frac{1}{2}}]$	3	255	16.1/0.8	14.6/0.7
$[1, a, a^2]$	3	255	12.6/0.6	11.3/0.5
Reed-Solomon	3	255	8.7/0.4	8.3/0.4
Naive op.	4		26.1/1.6	
$[1, a, a^{\frac{1}{2}}, X]$	4	96	10.2/0.6	9.7/0.6
$[1, a, a^{\frac{1}{2}}, a^{141}X]$	4	155	8.2/0.5	6.2/0.4
$[1, a, a^{\frac{1}{2}}, a^{186}X + a^6]$	4	168	6.3/0.4	5.7/0.4
Reed-Solomon	4	255	6.4/0.4	5.9/0.4

Benchmarks on Intel Xeon E3-1245 v3 @ 3.40 GHz.

`0x2`)$\cdots$ of degree m. Hence, the encoder multiplies elements of data by the polynomials $x^i \bmod g(x)$ corresponding to its position i. The multiplication is performed with the general multiplication in Sec. 3.1.2. The decoder for the RS code is also the syndrome decoder.

4.4.3 Discuss

Encoding

The RAIDq code of the recommended parameter $[1, a, a^{1/2}, X]$ always performs best for encoding in table 4.2. For the cases of multiple checksums, it is roughly a factor of 1/2 comparing to the maximum capability of "naive op.". In the case of 3 checksums, the $[1, a, a^{1/2}]$ code performs better than $[1, a, a^2]$ code because the multiplication by a is faster than a^2 and the technique in Eq. (4.2) improves the calculation of the $[a^{1/2}]$ checksum. For the case of 4 checksums, the RAIDq code $[1, a, a^{1/2}, a^{186}X + a^6]$ of maximum supporting discs shows an equal performance with the RS code. However, the advantage of RAIDq relies on the efficiency of coders of recommended parameters and the backward compatibility to RAID-5 and 6. The user can choose the coders depends on the trade-off between the throughput and supporting discs.

For comparing with Reed-Solomon code, the encoding throughputs of the coders for RAIDq always outperform the Reed-Solomon coder in Tab. 4.2. We analyze the result at the algorithmic level. The main difference of the encoders are the underlying multiplications in the finite field. In the Reed-Solomon code, we always perform the general field multiplication. In the coders for RAID, the multiplication in the encoder are always optimized with the technique in Sec. 3.1.2 and 3.1.3. RAIDq also has the edge over Reed-Solomon RAID even without the above optimizations because RAIDq still uses an `XOR` checksum, which is not possible with general Reed-Solomon codes.

Dncoding

For decoding, we can observe the decoding efficiency of $[1, a, a^{1/2}, X]$ RAIDq is even better than the encoding throughput of the RS code. Although we expect the performance gain from encoding will affect decoding due to syndrome decoding, it is almost true except the decoding throughput of the $[1, a, a^{\frac{1}{2}}, a^{186}X + a^6]$ code. We have slightly worse decoder than the Reed-Solomon code because the decoder performs additional expensive multiplications in $\mathbb{F}_{2^{16}}$ besides the encoding.

Last, we emphasize that the encoder is the most critical component for the performance of RAIDs, since the decoders are called only when the disk failure occurs. Even in the case of decoding, all performance results presented here represent worst-case scenarios. *We expect that in practice, the most common disks failures can be recovered from the much faster RAID 5 or 6 checksum computation, which is possible because RAIDq includes them as special cases.*

4.5 Summary

We present the design and implementation of RAIDq, a software-friendly, multiple-parity extension of Plank's code. RAIDq fixes the flaw of Plank's code in mainstream RAIDs at the case of 4 checksums to addresses bigger RAIDs of practical interest. However, RAIDq does have a limit of $164 + 4$ disks on the number of supporting drives, which is still lower than the expecting 255 drives in the original RAID systems. Another benefit of RAIDq is that it includes existing RAID-5 and 6 as special cases and hence has 100% backward compatible codewords. The backward compatibility also allows RAIDq to reuse the efficient coding algorithms and implementations of RAID-5 and 6. Last but not least, RAIDq is optimized for software implementation, as its encoding only involves `XOR` and multiplication by several fixed elements in $\mathbb{F}_{256}$ and $\widetilde{\mathbb{F}}_{256^2}$. The advantage of the efficient `Encode` results in the best throughputs for processing data in our experiments.

Chapter 5

Implementing Multivariate Public-Key Cryptosystems

In practice, a security system can be broken due to its implementation. In this chapter, we perform the arithmetic of fields under the cryptographic requirements and apply the technique to the implementation of MPKCs. This chapter is base on the joint work with Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang and Chen-Mou Cheng published in [CLP+18].

5.1 Introduction

5.1.1 The Requirements on Post-Quantum Security

Since Shor's algorithm [Sho97] was invented, it is clear that traditional public key cryptography(PKCs) based on discrete logarithm and RSA assumptions are going to be solved in polynomial time once large quantum computers are built. PKCs that retain sufficient security levels when quantum computers have arrived are said to be post-quantum. Such cryptosystems are also sometimes called Postquantum Cryptosystems or PQCs. There are four or five main classes of PQCs one of which comprise Multivariate public-key cryptosystems (MPKCs) [CJL+16, DY08].

5.1.2 Challenge in Cryptographic Software

The secure implementation is the priority for cryptographic software. In practice, a security system can be broken due to its implementation instead of the cryptography, e.g., the cache-timing attack to AES [BM06]. We would like reasonable implementations which retain as much as possible side channel resilience. This means that the secret data should be independent of memory access. In other words, time constancy is always an essential requirement when processing secret data.

Based on the secure implementation, MPKCs were usually advertised for speed. In 2009, Chen *et al.* [CCC+09] showed that MPKCs are easily a match for RSA and ECC at the 80-bit security level. It seems the basic security requirements has shifted to 128-bit, which can be seen from the call of new

post-quantum cryptographic schemes from NIST [oST16]. We have to look whether MPKC signature schemes remain viable in the age of 128-bit security.

5.1.3 Chapter Objectives

In this chapter, we will discuss the secure implementations of MPKCs and show the MPKCs still keep its speed advantage over the mainstream RSA and ECC in the age of 128-bits security by benchmarking our implementations in the eBACs [BL16], a standard platform for benchmarking cryptographic systems.

5.1.4 Chapter Overview

In section 5.2, we review the backgrounds on MPKC signatures. We then focus on the Rainbow, which is a typical MPKC signature. We give the detailed operation for signing process and the actual parameters for various security levels. After shortly recalling the field multiplication for cryptographic software in Sec. 3.3, we present the implementation for central components of MPKCs including the evaluation of quadratic polynomials and solving linear equations in section 5.3. We benchmark the implementations in section 5.4 and conclude in section 5.5.

5.2 Backgrounds on MPKC Signatures

5.2.1 MPKCs and its Security

MPKCs are PKCs whose public keys represent multivariate polynomials over a finite field(GF) $\mathbb{K} = \mathbb{F}_q$:

$$\mathcal{P} : \mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{K}^n \mapsto \mathbf{z} = (p_1(\mathbf{w}), \ldots, p_m(\mathbf{w})) \in \mathbb{K}^m \ .$$

Polynomials $p_1, p_2, \ldots$ have (almost always) been quadratic. In public-key cryptography, we can let $\mathcal{P}(\mathbf{0}) = \mathbf{0}$.

We first introduce the security of MPKCs to set the required parameters for the target security level(s).

Class $\mathcal{MQ}(q, n, m)$ and the $\mathcal{MQ}$ Problem

One can break all MPKCs if one is able to solve $\mathcal{MQ}$ problems efficiently. Given q, n, and m, the class $\mathcal{MQ}(q, n, m)$ consists of all systems of m quadratic polynomials in $\mathbb{F}_q$ with n variables. To choose a random system $\mathbf{S}$ from $\mathcal{MQ}(q, n, m)$, we write each polynomial $P_k(\mathbf{x})$ as $\sum_{1 \le i \le j \le n} a_{ijk} x_i x_j + \sum_{1 \le i \le n} b_{ik} x_i + c_k$, where every a_{ijk}, b_{ik}, c_k is chosen uniformly in $\mathbb{F}_q$.

Solving $\mathbf{S}(\mathbf{x}) = \mathbf{b}$ for any $\mathcal{MQ}$ system S is then known as the "multivariate quadratic" problem. It is an NP-complete problem [GJ79]. However, it is not easy to base a proof on worst-case hardness. Often the premise used is the hereto unchallenged average-case $\mathcal{MQ}$ hardness assumption [BGP06, LLY08].

In this thesis, we focus on the implementations instead of the details of hardness. The complexity of solving a random instance out of $\mathcal{MQ}(q, n, m)$ is estimated using Gröbner basis methods, often XL with sparse matrices [CKPS00, YCBC07], the FXL(also known as "the hybrid approach") [YCC04] ,or F5 [Fau02, BFSY05].

Extended Isomorphism of Polynomials (EIP)

Notice MPKCs cannot be random $\mathcal{MQ}$ polynomials, because the legitimate user would be equally unable to invert $\mathcal{P}$. Usually the public map of an MPKC have a structure in the "bipolar form": $\mathcal{P} = T \circ \mathcal{Q} \circ S$ where T and S are affine, and

$$\mathcal{P} : \mathbf{w} \in \mathbb{K}^n \overset{S}{\mapsto} \mathbf{x} \overset{\mathcal{Q}}{\mapsto} \mathbf{y} \overset{T}{\mapsto} \mathbf{z} \in \mathbb{K}^m.$$

The requirement for the quadratic *central map* $\mathcal{Q}$ is that it is easy to "invert" $\mathcal{Q}$ but not $\mathcal{P}$. In other words, given $y \in \mathbb{K}^m$, it is easy to compute x such that $\mathcal{Q}(x) = y$ but finding an x such that $\mathcal{P}(x) = y$ is hard. The structure is hidden away by S and T. Given this, the MPKC may be attacked via what is called structural attacks.

EIP and "Structural Attacks"

Given a class $\mathcal{C}$ of quadratic maps $\mathbb{K}^n \to \mathbb{K}^m$ and a quadratic map $\mathcal{P} : \mathbb{K}^n \to \mathbb{K}^m$, an associated EIP instance means to find S and T such that $\mathcal{P} = T \circ \mathcal{Q} \circ S$, where $\mathcal{Q} \in \mathcal{C}$. Defeating a bipolar-form MPKC through solving an EIP is known as a "structural" or Key-Recovery attack.

Note that solving an EIP problem is very ad hoc, depending very much on what $\mathcal{Q}$ is like, and again we do not go into the technical details but uses known EIP results in this paper. In other words, we assume the EIP, w.r.t. the existing schemes are hard unless new findings on the specific forms of EIP are proposed.

5.2.2 Recap of MPKC Signatures

In this section, we introduce the main procedures of MPKC signatures. It is almost universally accepted that it is challenging to design multivariate encryption schemes. Most encryption systems are either already been broken or have much larger sizes than signature schemes.

The Key Pair of Typical MPKCs

Recall the public key of an MPKC is a set of quadratic polynomials, which is a composition of two affine maps T and S and an easily invertible *central map* $\mathcal{Q}$, such that

$$\mathcal{P} : \mathbf{w} \in \mathbb{K}^n \overset{S}{\mapsto} \mathrm{M}_S\mathbf{w} + \mathbf{c}_S := \mathbf{x} \overset{\mathcal{Q}}{\mapsto} \mathbf{y} \overset{T}{\mapsto} \mathrm{M}_T\mathbf{y} + \mathbf{c}_T := \mathbf{z} \in \mathbb{K}^m.$$

The structure of $\mathcal{Q}$ is hidden away by S and T. Further, the structures of their Q's characterize various MPKCs. Therefore the secret key of an MPKC consists of the S, T, and $\mathcal{Q}$.

Main Procedures of Typical MPKC Signatures

The MPKC signature system comprises three main procedures: key generation, signing messages and verifying signatures.

To generate a key pair, the user randomly chooses a secret key which comprises invertible S, T, and $\mathcal{Q}$. The coefficients of public key $\mathcal{P}$ can be deduced using polynomial interpolation of $T \circ \mathcal{Q} \circ S$. We refer the reader to [Wol04]for the details of interpolation and other efficient key-generation methods.

To sign a message, the signer first computes the hash value of the message as the digest $\mathbf{z} \in \mathbb{K}^m$. With the secret key, the signer computes $\mathbf{y} = T^{-1}(\mathbf{z})$, $\mathbf{x} = \mathcal{Q}^{-1}(\mathbf{y})$, and $\mathbf{w} = S^{-1}(\mathbf{x}) \in \mathbb{K}^n$ which is the signature of the message. The details of $\mathcal{Q}^{-1}$ varies with specific schemes.

To verify a signature $\mathbf{w} \in \mathbb{K}^n$ with the digest $\mathbf{z}$, the user evaluates the public polynomials $\mathcal{P}(\mathbf{w})$ and checks whether the values $\mathcal{P}(\mathbf{w})$ are equal to the digest $\mathbf{z}$.

5.2.3 The Rainbow Signature

In this section, we demonstrate the Rainbow signature, which is the main MPKC signatures considered secure today, and show the parameters of Rainbow for 128-bit security. We will discuss the implementation in section 5.3.

Rainbow [DS05] is the stereotypical "small field" MPKC, where work on the "small" fields ($\mathbb{F}_{16}$, $\mathbb{F}_{31}$, and $\mathbb{F}_{256}$). Although a similar signature TTS [DYC+08] had been proposed earlier, it can be considered as Rainbow with a sparse $\mathcal{Q}$ in today's terminology. We will use "Rainbow" as a generic term for this branch of variants. The definitive analysis of security for Rainbow and the formulation of current instances can be found in the 2008 paper [DYC+08]. 80-bit secure parameters are chosen in [PBB10].

Central Map in Rainbow

Rainbow($\mathbb{F}_q, v_1, o_1, \ldots, o_u$) is characterized as follows as an u-stage UOV of the central map $\mathcal{Q}$ [DS05, DYC+08].

- The segment structure is given by a sequence $0 < v_1 < v_2 < \cdots < v_{u+1} = n$. For $l = 1, \ldots, u+1$, set labels for "vinegar" variables as $V_l := \{1, 2, \ldots, v_l\}$ so that $|V_l| = v_l$ and $V_1 \subset V_2 \subset \cdots \subset V_{u+1} = V$. Denote sets of "oil" variables by $o_l := v_{l+1} - v_l$ and $O_l := V_{l+1} \setminus V_l$ for $l = 1 \cdots u$.

- The central map $\mathcal{Q}$ comprises m structurized quadratic equations $\mathbf{y} = (y_{v_1+1}, \ldots, y_n) = (q_{v_1+1}(\mathbf{x}), \ldots, q_n(\mathbf{x}))$, where

$$y_k = q_k(\mathbf{x}) = \sum_{i=1}^{v_l} \sum_{j=i}^{v_{l+1}} \alpha_{ij}^{(k)} x_i x_j + \sum_{i<v_{l+1}} \beta_i^{(k)} x_i \ ,$$

  for $k \in O_l := \{v_l + 1, \ldots, v_{l+1}\}$.

- *Note that in every q_k, where $k \in O_l$, there is no cross-term $x_i x_j$ where both i and j are in O_l.* So given all the y_i with $v_l < i \leq v_{l+1}$, and all the x_j with $j \leq v_l$, we can easily compute $x_{v_l+1}, \ldots, x_{v_{l+1}}$.

Generating Signatures in Rainbow

To sign a message, the signer calculate the hash digest $\mathbf{z}$ of message and inverts $\mathcal{P}$ with the secret key T, S, and $\mathcal{Q}$ by

$$\mathbf{z} \in \mathbb{K}^m \overset{T^{-1}}{\longmapsto} \mathbf{y} \overset{\mathcal{Q}^{-1}}{\longmapsto} \mathbf{x} \overset{S^{-1}}{\longmapsto} \mathbf{w} \in \mathbb{K}^n \ ,$$

where $\mathbf{w}$ is the signature. The key step here is inverting the central map $\mathcal{Q}$. While inverting $\mathcal{Q}$ with given $\mathbf{y}$, the signer randomly guesses vinegar variables $\bar{\mathbf{x}} = (x_1, \dots x_{v_1})$ and solve $(x_{v_1+1}, \dots, x_{v_1+o_1})$ by

$$\begin{aligned} y_{v_1+1} &= \bar{\alpha}^{(v_1+1)}_{v_1+1} x_{v_1+1} + \cdots + \bar{\alpha}^{(v_1+1)}_{v_1+o_1} x_{v_1+o_1} + \bar{\beta}^{(v_1+1)}_{V_1} \\ &\vdots \\ y_{v_1+o_1} &= \bar{\alpha}^{(v_1+o_1)}_{v_1+1} x_{v_1+1} + \cdots + \bar{\alpha}^{(v_1+o_1)}_{v_1+o_1} x_{v_1+o_1} + \bar{\beta}^{(v_1+o_1)}_{V_1} \quad . \end{aligned} \tag{5.1}$$

Here $(\bar{\beta}^{(v_1+1)}_{V_1}, \dots, \bar{\beta}^{(v_1+o_1)}_{V_1})$ is an evaluation of secret-quadratic equations with secret values $\bar{\mathbf{x}}$ and the matrix

$$\begin{bmatrix} \bar{\alpha}^{(k)}_{i} & \cdots & \bar{\alpha}^{(k)}_{i'} \\ & \ddots & \\ \bar{\alpha}^{(k')}_{i} & & \bar{\alpha}^{(k')}_{i'} \end{bmatrix}, \text{ where } i, i' \text{ and } k, k' \in O_1 \ ,$$

denoted by $\texttt{matVO}(\bar{\mathbf{x}})$, is evaluated as linear forms in $\bar{\mathbf{x}}$. The signer then solves Eq. (5.1) with a linear solver for all x_i where $i \in O_l$. There are total u linear systems to be solved. The signer may have to repeat the process if any $\texttt{matVO}(\bar{\mathbf{x}})$ is a singular matrix. Hence, the main computation cost of the signing process depends on computing the matrices $\texttt{matVO}(\bar{\mathbf{x}})$ from vinegar variables $\bar{\mathbf{x}}$ and solving the corresponding linear equations.

Parameters of Modern Rainbow

In current Rainbow, u is always 2, with parameters (v, o, o), and at b-bit security $q^o \gtrsim 2^b$ (rank attacks [YC05]). The number of variables and equations are $(n, m) = (v + 2o, 2o)$. Against a Rainbow with m equations and n variables, the most pertinent attacks are substituting $n - m$ variables at random and trying to solve for the remaining m variables ("Direct Attack"), and a structural attack which involves solving an associated quadratic system with n variables and $n + m - 1$ equations ("Rainbow Band Separation"). Therefore we require $2^b \lesssim \min(C_{FXL}(m, m; q), C_{FXL}(n, m + n - 1; q))$ [DYC+08].

Ding et al. [DYC+08, CCC+08] suggest for 80-bit design security Rainbow/TTS with parameters $(\mathbb{F}_{2^4}, 24, 20, 20)$ and $(\mathbb{F}_{2^8}, 18, 12, 12)$. We modify the parameters for modern security requirements in Table 5.1.

Table 5.1: Parameters of Rainbow.

security	parameter	$\mathbb{F}_{16}$	$\mathbb{F}_{31}$	$\mathbb{F}_{256}$
128 bits	(v_1, o_1, o_2)	32,32,32	28,28,28	28,20,20
	$n \to m$	$96 \to 64$	$84 \to 56$	$68 \to 40$
192 bits	(v_1, o_1, o_2)	48,48,48	53,40,40	52,32,32
	$n \to m$	$144 \to 96$	$133 \to 80$	$116 \to 64$
256 bits	(v_1, o_1, o_2)	64,64,64	74,56,56	73,48,48
	$n \to m$	$192 \to 128$	$186 \to 112$	$169 \to 96$

5.3 Implementing Components for Rainbow

We discuss the implementations of central components for Rainbow in this section.

Before the main contents of this section, we shortly recall the two multiplications for cryptographic software in Sec. 3.3. The first technique performs the multiplication with Log/Exp tables for constant-time multiplication in $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$. The other method multiplies field elements with multiplication tables. Instead of loading the multiplication tables with secret values, however, we generate the multiplication tables for particular variables and loading the tables with the address of the variables. Since we have to pay the cost for generating multiplication tables, the method befits the occasion of plenty multiplications by the same variables.

5.3.1 Matrix-vector Multiplication

The first component for implementing MPKCs is the matrix-vector multiplication in the process of signing. We show the implementation for performing the affine map S and T in this section.

In the secret key, we store the linear maps S and T as column-major matrices. For performing $\boldsymbol{x} = T \cdot \boldsymbol{y}$, the calculation

$$\begin{bmatrix} \boxed{\begin{matrix} t_{00} \\ t_{10} \\ t_{20} \\ \vdots \end{matrix}} & \boxed{\begin{matrix} t_{01} \\ t_{11} \\ t_{21} \\ \vdots \end{matrix}} & \boxed{\begin{matrix} t_{02} \\ t_{12} \\ t_{22} \\ \vdots \end{matrix}} & \cdots \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \end{bmatrix}$$

is performed as $y_0 \cdot (t_{00}, t_{10}, \ldots) + y_1 \cdot (t_{01}, t_{11}, \ldots) + \cdots$. Here a boxed column represents the data in the same register. The multiplications are implemented with the Log/Exp tables to avoid side-channel leakage.

5.3.2 Evaluating Quadratic Systems

The evaluation of instances in $\mathcal{MQ}$ is the most critical component in MPKCs. It corresponds to the verification of a signature or the public map directly. The time constancy of the evaluation is depended on the circumstances. For example, the public map of MPKCs is usually an evaluation without the requirement on time constancy. However, for generating Eq. 5.1 in the secret map of Rainbow, the constant-time evaluation of secret polynomials is required.

Note on Lack of Special Structures in $\mathcal{MQ}$

For the evaluation of quadratic systems, there is no method to reduce the required computations since we expect to evaluate a random system unless particular patterns were designed into the equations (which only happens in unusual variant schemes which do not concern us here). Hence, we focus on reducing the running-time for evaluating instances in $\mathcal{MQ}$ via choosing the correct instruction sequences over various platforms since we expect the same amount of required computations. Most of the time, the fastest running-time equates to the fewest instruction counts.

Evaluating Quadratic Systems as Matrix-Vector Product

Figure 5.1: An example of parallel evaluation of polynomials.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} c_{11} \\ c_{21} \\ c_{31} \\ \vdots \end{bmatrix} \cdot x_1 + \begin{bmatrix} c_{12} \\ c_{22} \\ c_{32} \\ \vdots \end{bmatrix} \cdot x_2 + \cdots + \begin{bmatrix} c_{111} \\ c_{211} \\ c_{311} \\ \vdots \end{bmatrix} \cdot x_1 x_1 + \begin{bmatrix} c_{112} \\ c_{212} \\ c_{312} \\ \vdots \end{bmatrix} \cdot x_1 x_2 + \cdots$$

The registers $(y_1, y_2, y_3, \ldots)$ accumulates the results comprising $x_1 \cdot (c_{11}, c_{21}, c_{31}, \ldots)$, $x_2 \cdot (c_{12}, c_{22}, c_{32}, \ldots)$, ..., $x_1 x_1 \cdot (c_{111}, c_{211}, c_{311}, \ldots)$, etc.

We evaluate a quadratic system as a matrix-vector multiplication in Sec. 5.3.1. A multivariate quadratic system $\mathcal{P}$ with n variables and m polynomials is usually stored as a column-major matrix with the columns being all monomials up to degree 2 and the rows being the polynomials (See Figure 5.1). Hence, the evaluation of $\mathcal{P}$ can roughly be divided into two steps:

1. the generation of all monomials, i.e., the vector, and
2. computation of the resulting polynomials for known monomials, i.e., the matrix-vector multiplication.

The computation proceeds by accumulating the product of a column vector with a prepared monomial as shown in Fig. 5.1, which is exactly a matrix-vector production.

An alternative evaluation skips the first step and generates the quadratic terms through multiplications by variables (twice). In a degree-reverse-lex order for the monomials of polynomial, the quadratic terms are ordered as

$$(c_{*11}x_1)x_1 + (c_{*12}x_1 + c_{*22}x_2)x_2 + (c_{*13}x_1 + c_{*23}x_2 + c_{*33}x_3)x_3 + \cdots \quad .$$

One can accumulate all the linear terms in one parenthesis and follows with multiplication by the second variable.

The second step is the most computationally intensive part for evaluating $\mathcal{P}$. Since there are $(n++n\cdot\frac{n+1}{2})$ monomials, the second step requires $m\cdot(n+n\cdot\frac{n+1}{2})$ multiplications to multiply the coefficients of $\mathcal{P}$ by the quadratic monomials and almost the same number additions to accumulate results.

Now we consider the complexity of the first step. A straightforward generation of all quadratic monomials requires $n \cdot (n+1)/2$ multiplications. The evaluation without generating quadratic monomials costs $n \cdot m$ extra multiplications for multiplying variables again. One can choose the method of calculation of quadratic terms with the value of n and m for a lower cost of computation.

Optimization and Constant-Time Evaluation

We first discuss the case without the constant-time requirement. For the genuinely public map, the multiplications in $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$ can be done by (1) loading

the multiplication tables with the value of the variables and (2) performing the multiplication with Algo. 1 simultaneously. We can also omit some computations based on the value of variables x_i. However, we can not load the multiplication tables or skip computations based on the secret values for crypto-safe evaluations.

For the constant-time evaluation, we use the strategy of generating multiplication tables since we can reuse the tables while evaluating quadratic terms. In other words, we have to use the approach of multiplying quadratic terms with multiplications of the coefficient and two linear variables while evaluating $\mathcal{P}$. We first generating all multiplication tables for all variable $x_0, \ldots, x_{n-1}$ with Algo. 2, Eq. (3.6), and Eq. (3.7. Then we load the tables with *the indexes* of variables instead of their values to prevent side-channel leakage.

Table 5.2 shows the benchmarks of our implementations for evaluating quadratic systems. We can see only about a 5% difference between constant-time and general evaluations over $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$. Hence, we conclude that the extra cost for generating n multiplication tables is low comparing to the evaluation.

Table 5.2: Benchmarks on evaluations of quadratic polynomials

system	size k byte	const. time k cycles	general k cycles
$\mathbb{F}_{16}, n = 64, m = 64$	65	9.6	9.1
$\mathbb{F}_{256}, n = 64, m = 64$	130	16.2	15.6

Benchmarking in CPU cycles on Intel XEON E3-1245 v3 @ 3.40GHz with AVX2 instruction set.

5.3.3 Solving Linear Equations

We present the constant-time solver from modifying the Gaussian Elimination in this section. Constant-time Gaussian Elimination was initially introduced in [BCS13] for matrices over $\mathbb{F}_2$. Their idea can be easily extended to the case of $\mathbb{F}_{256}$.

The constant-time solver is actually a "worse-case Gaussian elimination" (Algorithm 5) since changing the control flow according to the "value" of pivots is the undesired property. To avoid the timing difference from the swapping for zero pivots, we perform every possible row-swap in the Algo. 5. It is a so-called "conditional move" at line 8 with a predicate as the current pivot(line 7). The conditional move is implemented as multiplication by a value in $\{0, 1\}$ or as an `AND` operation with a bit mask. We perform the row operations, a row-vector multiplying by a scalar(line 14), with Log/Exp tables for time constancy.

We use worse-case Gaussian elimination in the signing process of Rainbow and report the timing for solving the system in the signing process in Tab. 5.3. Solving linear equations (Eq. (5.1)) takes up much of the time during the signing process of Rainbow as seen in Sec. 5.2.3. From the table, it is clear that the constant-time Gaussian elimination is slower than plain version, but it is still an $O(n^3)$ operation.

Algorithm 5: The worse-case Gaussian elimination

```
WorseCaseGaussian( A) :
  input : A : an n × m matrix for m > n over F_256 .
  output: A : an n × m matrix with all 1's for diagonal entries if success.
for i ← 0 to n − 1 do
    Let A_i be the i-th row of A.
    Let a_ii be the i-th entry of the row vector A_i.
    // Swap pivots.
    for j ← i + 1 to n − 1 do
        Compute m ← NOT(OR all bits of a_ii).
        A_i ← A_i + m · A_j.
    end
    A_i ← Inverse(a_ii) · A_i.
    // Forward and backward Eliminations.
    for j ← 0 to n − 1 do
        if i = j then continue.
        A_j ← A_j + a_ji · A_i.
    end
end
return A.
```

Table 5.3: Benchmarks of linear solvers with Gaussian elimination

system	plain elimination	constant version
32×32 over $\mathbb{F}_{16}$	6,610	9,539
20×20 over $\mathbb{F}_{256}$	4,702	9,901

Benchmarking in CPU cycles on Intel XEON E3-1245 v3 @ 3.40GHz.

5.4 Benchmarks

In this section, we give comparisons of benchmarks among Rainbows over binary fields and some widely used schemes (though not post-quantum ones). Almost all the schemes in the comparisons are parameterized at a 128-bit security level, besides the RSA-2048 is in the 112-bit security level. Tab. 5.4 lists the specific parameters for the schemes under comparisons.

Table 5.4: Parameters of signature schemes

schemes	public key kbyte	secret key kbyte	digest bit	signature bit
Rainbow(16,32,32,32)	145.5	100.2	256	384
Rainbow(256,28,20,20)	94.3	62.9	320	544
ECDSA(NIST P256)	0.064	0.096	256	512
Ed25519	0.032	0.064	256	512
RSA-2048 [a]	0.256	2.048	2048	2048
RSA-3072	0.384	3.072	3072	3072

[a] 112-bit security.

5.4.1 The Benchmarks

We list the results of benchmarking in Tab. 5.5. Our implementations of MPKCs[1] were tested in the following environment:

- CPU: Intel XEON E3-1245 v3 (Haswell) @ 3.40GHz, turbo boost disabled.
- memory: 32 GB ECC.
- OS: ubutnu 1604, Linux version 4.4.0-78-generic.
- gcc: 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.4).

We collect other benchmarks under the same Intel Haswell architecture.

Since the signing and verifying processes are the most commonly used functions, they are the primary targets of our comparison with other mainstream pre-quantum systems. For verifying signatures, the results show the Rainbow is indeed very efficient in general. For generating signatures, we can observe the Rainbow over $\mathbb{F}_{256}$ is the most efficient among all schemes in comparisons and all instances of Rainbows are comparable with Ed25519 [BDL+11], which is the most efficient pre-quantum signature in our comparisons.

Table 5.5: Benchmarks of Signature Schemes on Intel Haswell Archiecture.

schemes	gen-key() M cycles	sign() k cycles	verify() k cycles
Rainbow(16,32,32,32)	1,359.7	68.1	22.8
Rainbow(256,28,20,20)	328.9	47.8	18.3
ECDSA(NIST P256) [b]	0.286	377.1	901.5
Ed25519 [b]	0.066	61.0	185.1
RSA-2048 [a,b]	233.7	5,240.2	66.4
RSA-3072 [b]	844.4	15,400.9	119.3

[a] 112-bit security.
[b] [BL16] benchmarked ECC and RSA on Intel Xeon E3-1275 v3 (Haswell) at 3.5GHz.

5.5 Summary

We have reviewed the Rainbow signature at the 128-bits security level and analyzed the main components of Rainbow signatures including evaluating $\mathcal{MQ}$ equations and solving linear equations. We present techniques for implementing these main components in x86 platforms using SIMD instructions with side-channel resilience. The implementations are based on the following methods for multiplying field elements securely:

1. We use SIMD table lookup and log/exp tables for preventing cache-time attacks.

[1] The software for MPKC experiments can be downloaded from `https://github.com/fast-crypto-lab/mpkc-128bit` .

2. For the private evaluation of instances of $\mathcal{MQ}$ over $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$, we generate instead of load the multiplication tables with the values of multipliers and thus obtain a constant-time evaluation of quadratic systems nearly as fast as a public evaluation.

From the benchmarks, we conclude that MPKC signatures remain competitive speed-wise under crypto-safe requirements in current mainstream instruction sets.

Chapter 6

The Additive FFT and its Implementation in Binary Fields

We review the additive FFT and describe its implementation in this chapter. These are the preliminaries for Chapter 7. The additive FFT here was developed by Cantor [Can89], Gao and Mateer [GM10], Lin, Chung, and Han [LCH14], and Lin, Al-Naffouri, and Han [LANH16]. The implementations are based on the joint work with Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. The preprint version can be found in [CCK+17].

6.1 Introduction

A fast Fourier transformation(FFT) is an algorithm evaluating the values of polynomials at particular points efficiently. It is useful with many applications including Reed-Solomon Code [LCH14, LANH16], message authentication codes in cryptographies [BC14], and even the multiplication of Boolean polynomials [BGTZ08, HvdHL16, CCK+17, vdHLL17, CCK+17, LCK+18], a fundamental problem in computer science. In this chapter, we discuss the additive FFT and its implementations.

6.1.1 FFTs over Binary Fields

Evaluating univariate polynomials over binary fields attracts great research interests since the FFTs over binary fields are more complicated compared with the FFT over real numbers. The standard "multiplicative" FFTs evaluates polynomials at multiplicative subgroups formed by roots of unity. Unfortunately, in the binary fields, a multiplicative subgroup formed by roots of unity consisted of elements of odd order. The simplest divide-and-conquer strategy, breaking down the problem into two sub-problems, need not exist in the multiplicative FFTs. Therefore constructing the desired subgroups may induce an extra burden for multiplicative FFTs. For example, the Schönhage [Sch77] FFT evaluates polynomials at points formed by a "virtual" root of unity with the order 3^n. The

Cooley-Tuckey FFT works on the subgroups which orders are factors of the original order of the multiplicative group.

The Additive FFT over Binary Fields

The additive FFTs evaluate polynomials in binary fields at points forming an *additive* subgroup, in which the size is 2^i for $i \in \mathbb{N}$ and thus can be divided into two subgroups of size 2^{i-1} easily. Hence, it fits the most straightforward divide-and-conquer strategy and leads to efficient implementation in practice.

In general, for evaluating a polynomial

$$f = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1} \quad , n = 2^l \text{ for } l \in \mathbb{N} \text{ and } f_i \in \widetilde{\mathbb{F}}$$

at points $\{0, 1, \omega_{\widetilde{\mathbb{F}}}(2), \ldots, \omega_{\widetilde{\mathbb{F}}}(n-1)\}$, the additive FFT evaluates the polynomial f in two steps. The first step converts the polynomial basis of f from the monomial basis $(1, x, x^2, x^3, \ldots)$ to a *novelpoly* basis which will be described in Sec. 6.3.1. We name the second step "butterfly network" in this thesis. At the step, given the polynomial f in the *novelpoly* basis, one recursively divides the evaluation of f of length n at n points into the evaluations of two shorter polynomials of length $\frac{n}{2}$ at two sets $\{0, 1, \ldots, \omega_{\widetilde{\mathbb{F}}}(\frac{n}{2} - 1)\}$ and $\omega_{\widetilde{\mathbb{F}}}(\frac{n}{2}) + \{0, 1, \ldots, \omega_{\widetilde{\mathbb{F}}}(\frac{n}{2} - 1)\}$. In comparison of complexities, the basis conversion takes $O(n \log n \log \log n)$ operations of field addition. and the butterfly network takes $O(n \log n)$ operations of field addition and multiplication.

6.1.2 The Development of Additive FFTs

We review the research of additive FFTs in this section.

In 1989, Cantor [Can89] developed the techniques to evaluate polynomials at points that form an *additive* subgroup. Although the additive FFT applies to general fields, the FFT is particularly useful in binary fields. The additive FFT was presented over a particular representation of fields, termed "Cantor basis" later in the literature. Cantor's FFT works with the complexity of $O(n \log n)$ multiplications and $O(n \log^{\log_2 3} n)$ additions(`XOR`) for evaluating $n = 2^l$ elements.

In 2010, Gao and Mateer [GM10] presented an additive FFT (heretofore "GM FFT") over $\mathbb{F}_{2^m}$, where the evaluation points are an additive subgroup of size 2^l in the underlying field. The complexity in GM FFT is $O(n \log^2 n)$ `XOR` operations for evaluating a polynomial at $n = 2^l$ points in general. However, it can be optimized to $O(n \log n \log \log n)$ `XOR` and $O(n \log n)$ field multiplications when $l = m$ and m is a power of 2. In the optimized case, the fields are represented in the Cantor basis.

In 2014, Lin, Chung, and Han [LCH14] proposed the *novelpoly* basis for polynomials. They shifted the problem to target on evaluating polynomials in the *novelpoly* basis instead of the usual monomial basis. In the *novelpoly* basis, they can evaluate the polynomials with the only process of the butterfly network and result in an FFT of $O(n \log n)$ complexities for both field additions and multiplications. In the subsequent work [LANH16], motivated by GM FFT, they presented a method for converting a polynomial between *novelpoly* and monomial basis. The complexity for the conversion is $O(n \log n \log \log n)$ `XOR` operations.

In the these, we follow the additive FFT developed from [LCH14] and [LANH16]. The additive FFT clearly separates the basis conversion and butterfly network into two stags (instead of interleaved basis conversion and butterfly stages of GM FFT) and works in the best-known complexity of $O(n \log n \log \log n)$ `XOR` and $O(n \log n)$ field multiplications for the polynomials in monomial basis.

The Implementations

In 2014, Bernstein and Chou [BC14] presented an efficient implementation of the GM FFT for evaluating polynomials in $\widetilde{\mathbb{F}}_{256}[x]$. For further optimization of GM FFT, they omitted the basis conversion step as one variant of their additive FFT. They applied their implementations to field multiplications in $\mathbb{F}_{2^{256}}$ for a message authentication code(MAC) in cryptography. However, their implementation employed only bit-operations without considering the powerful instruction sets on modern CPUs.

We can not find software based on additive FFTs before the work of [CCK+17] for the problem of multiplying Boolean polynomials. Hence, we discuss the implementation of the additive FFT on modern CPUs in this chapter.

6.1.3 Overview of this chapter

After the introduction, Section 6.2 discusses some properties of subspace polynomials. Section 6.3 describes the additive FFT algorithm, mainly divided into the butterfly network in Sec. 6.3.2 and basis conversion in Sec. 6.3.3. We show the techniques for implementing the additive FFT in Sec. 6.4.

6.2 Subspace Polynomials

We review subspace polynomials defined in Eq. (2.15) and (2.16) in this section. The development of additive FFTs relies heavily on subspace polynomials. The section may contain some redundant information for the sake of self-completeness of this chapter.

Cantor [Can89] defined the subspace polynomials over $\widetilde{\mathbb{F}}$ as

$$s_1(x) = x^2 - x = \prod_{a \in \mathbb{F}_2} (x - a) \ , \qquad \text{(2.15 revisited)}$$

and inductively

$$s_{i+1}(x) = s_1(s_i(x)) \ , \quad i = 1, 2, \ldots \qquad \text{(2.16 revisited)}$$

The starting few polynomials of $s_i(x)$ are $s_0(x) = x$, $s_1(x) = x^2 - x$, $s_2(x) = x^4 + x$, etc. We enumerate more polynomials of $s_i(x)$ in Tab. 6.1.

With respect to Cantor basis (v_i) in Eq. (2.19), subspace polynomials vanish the subspaces of $\widetilde{\mathbb{F}}$ as

$$s_i(x) := \prod_{a \in W_i} (x - a) \ . \qquad (6.1)$$

One can see the kernel space of $s_i(x)$ is W_i and $\deg s_i(x) = |W_i| = 2^i$ since $\dim(W_i) = i$.

Cantor *et al.* [Can89] [GM10] showed the following useful properties for s_i:

- $s_i(x)$ is linear, i.e., $s_i(x+y) = s_i(x) + s_i(y)$.
- $s_i(x) = x^{2^i} + x$ iff i is a power of 2, i.e., W_i is a field.
- $s_i(x) = s_{i-1}^2(x) + s_{i-1}(x) = s_1(s_{i-1}(x))$; $s_{i+j}(x) = s_i(s_j(x))$.

With $s_0(x) = x$ and $s_{i+1} = s_i^2 + s_i$, by induction, we know s_i contains only terms with coefficients 1 and monomials x^{2^j}. Moreover, if $i = 2^{k_0} + 2^{k_1} + \cdots + 2^{k_j}$, where $2^{k_0} < 2^{k_1} < \cdots < 2^{k_j}$, then we can write

$$s_i(x) = s_{2^{k_0}}(s_{2^{k_1}}(\cdots(s_{2^{k_j}}(x))\cdots)) \ . \tag{6.2}$$

Therefore every s_i can be a composition of polynomial functions with only two terms.

The evaluation of $s_i(x)$ at points in the Cantor basis is fast. From Eq. (6.1),

$$\forall a \in W_i, \quad s_i(a) = 0 \ .$$

For computing $s_i(v_j)$ for $j \geq i$, we can generalize Eq. (2.20) to

$$s_i(v_j) = s_{i-1}(s_1(v_j)) = s_{i-1}(v_j^2 + v_j) = s_{i-1}(v_{j-1}) = \cdots = v_{j-i} \ . \tag{6.3}$$

Hence, the effect of the linear operator s_i is to shift the corresponding binary expansion of elements in the Cantor basis to the right by i bits, or

$$s_i(\alpha) = \omega(\omega^{-1}(\alpha) \gg i) \ . \tag{6.4}$$

For example, we have $s_i(v_i) = v_0 = 1$.

6.3 The Additive FFT

We describe the additive FFT in this section. Roughly speaking, the additive FFT first converts a polynomial into the *novelpoly* basis and then evaluates the polynomial in a butterfly network. Section 6.3.1 describes the *novelpoly* basis for polynomials. Section 6.3.2 describes the butterfly network. i.e., the FFT of polynomials in the *novelpoly* basis. Section 6.3.3 gives the algorithm for converting polynomial bases.

6.3.1 The *novelpoly* Basis w.r.t. Subspace Polynomials

In [LCH14], Lin *et al.* proposed the *novelpoly* basis for polynomials. For polynomials of degree $< n$ in the *novelpoly* basis, they can evaluate the polynomials in $O(n \log n)$ field operations. Note that the *novelpoly* basis for *polynomials* must be distinguished from the Cantor basis for the *fields*.

Let $(1, X_1(x), X_2(x), \ldots)$ be the *novelpoly* basis for polynomials. The element of the *novelpoly* basis is defined as

$$X_k(x) := \prod (s_i(x))^{b_i} \quad \text{where } k = \sum b_i \cdot 2^i \text{ with } b_i \in \{0, 1\} \ . \tag{6.5}$$

In other words, $X_k(x)$ is the product of all $s_i(x)$ where the i-th bit of k is set. Clearly, for k is a power of 2, $X_k(x) = X_{2^i}(x) = s_i(x)$. Since $\deg s_i(x) = 2^i$, $\deg X_k(x) = k$.

6.3.2 Evaluating Polynomials in the *novelpoly* Basis

For a polynomial $f(x)$ in the *novelpoly* basis

$$f(x) = g(X) = g_0 + g_1 X_1(x) + \ldots + g_{n-1} X_{n-1}(x) \in \widetilde{\mathbb{F}}_{2^m}[x]_{<n} \text{ and } n = 2^l, \quad (6.6)$$

we can evaluate $f(x)$ at the set of points $\alpha + W_l$, where $\alpha \in \widetilde{\mathbb{F}}_{2^m}$, in $O(n \log n)$ field operations with a butterfly network, denoted as `Butterfly`. From [LCH14], the `Butterfly` is an FFT for polynomials in the *novelpoly* basis.

The Butterflies

Algorithm 6: The `Butterfly` for polynomials in the *novelpoly* basis.

1 `Butterfly(` $g(X) = f(x) \in \widetilde{\mathbb{F}}_{2^m}[x]_{<n}, \alpha \in \widetilde{\mathbb{F}}_{2^m}$ `)` :
  input : $g(X) = g_0 + g_1 X_1(x) + ... + g_{n-1} X_{n-1}(x) \in \widetilde{\mathbb{F}}_{2^m}[x]_{<n}$.
  an extra scalar: $\alpha \in \widetilde{\mathbb{F}}_{2^m}$.
  output: $[f(0+\alpha), f(1+\alpha), \ldots, f((\omega(n-1))+\alpha)]$, which is a list of values of f at points $\alpha + W_l$.
2 **if** $\deg(g(X)) = 0$ **then** return $[g_0]$
3 Let $i \leftarrow \lceil \log_2 n \rceil - 1$ s.t. $\deg(s_i(x)) = \deg(X_{2^i}(x)) = 2^i < n$.
4 Let $g(X) = p_0(X) + X_{2^i} \cdot p_1(X) = p_0(X) + s_i(x) \cdot p_1(X)$.
5 Compute $h_0(X) \leftarrow p_0(X) + s_i(\alpha) \cdot p_1(X)$.
6 Compute $h_1(X) \leftarrow h_0(X) + s_i(v_i) \cdot p_1(X)$.
7 **return** [`Butterfly(`$h_0(X), \alpha$`)`,`Butterfly(`$h_1(X), v_i + \alpha$`)`]

Algorithm 6 details the process of the `Butterfly`. It is a typical divide-and-conquer process transforming the current problem into 2 small sub-problems. Line 4 expresses the polynomial $f = g(X)$ as two half-sized polynomials $p_0(X)$ and $p_1(X)$ with

$$g(X) = p_0(X) + X_{2^i}(x) \cdot p_1(X) = p_0(X) + s_i(x) \cdot p_1(X) \quad , l = i + 1 \ .$$

For evaluating $f(x)$ at a set $\alpha + W_l = \alpha + W_{i+1}$, We can divide the points in W_{i+1} into the two half-sized sets

$$W_l = W_{i+1} = W_i \cup (W_{i+1} \backslash W_i) = W_i \cup (v_i + W_i) \ .$$

Since $s_i(x)$ vanishes W_i, all values of $s_i(x)$ at $\alpha + W_i$ are the same $s_i(\alpha)$, and the values of $f(x)$ at $\alpha + W_i$ become the values of the polynomial

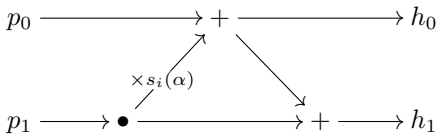
$$h_0(X) = p_0(X) + s_i(\alpha) \cdot p_1(X) \quad (6.7)$$

at $\alpha + W_i$. Similarly, the values of $f(x)$ at $\alpha + v_i + W_i$ become the values of the polynomial

$$h_1(X) = p_0(X) + (s_i(\alpha) + s_i(v_i)) \cdot p_1(X) = h_0(X) + s_i(v_i) \cdot p_1(X) \quad (6.8)$$

at $\alpha + v_i + W_i$. Note that the polynomials $h_0(X)$ and $h_1(X)$ are half length of original polynomial $g(X)$.

Figure 6.1: The butterfly unit.



Line 5 and 6 in the Algo. 6 perform the actual computations for generating two new sub-polynomials $h_0(X)$ and $h_1(X)$. Figure 6.1 shows the compuatation pattern for calculating one term of polynomials $h_0(X)$ and $h_1(X)$. The pattern is named a butterfly. There are two multipliers $s_i(\alpha)$ and $s_i(v_i)$ in each butterfly from Eq. (6.7) and (6.8). Since $s_i(v_i) = 1$ in the Cantor basis, one butterfly comprises two field additions and only one field multiplication.

Although line 7 indicates the use of recursion, we actually program iteratively in many layers of butterflies(See Fig. 6.2). One layer of butterflies corresponds to one recursion. It turns out that there are l layers, corresponding to the depth of recursion and $n/2$ butterflies in each layer. Through the iterative programming style, it is possible to optimize data movement in the `Butterfly` by processing several layers at the same time. Inverse `Butterfly` simply performs the butterflies in reverse.

We remark at last that the length of the input polynomial is equal to the number of evaluating points in Algo. 6. We will evaluate polynomials of high degree at a set of smaller size in Sec. 7.3.2.

The butterflies: An Example

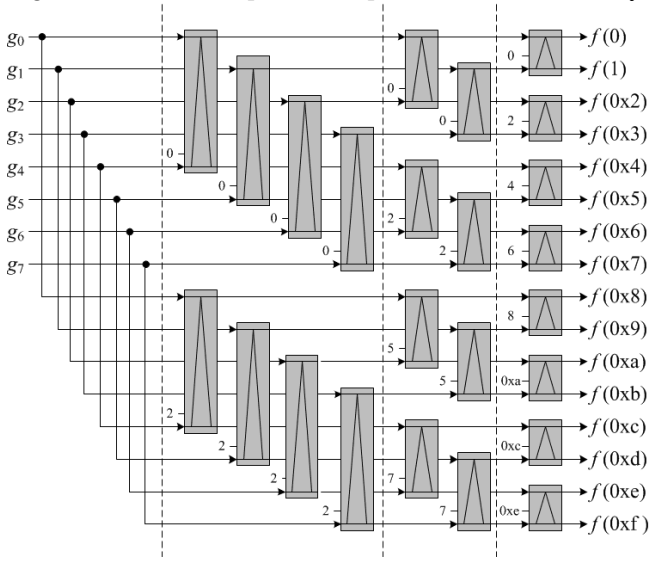
Figure 6.2 shows an example of evaluating a degree-7 polynomial $f(x) = g(X) = g_0 + \cdots + g_7 X_7 \in \widetilde{\mathbb{F}}[x]$ in *novelpoly* basis at points $W_4 = \{0, 1, \ldots, \omega_{\widetilde{\mathbb{F}}}(\texttt{0xf})\}$. It actually calls the `Butterfly` twice depicted in upper and lower parts respectively in the figure. The upper part is $\texttt{Butterfly}(g(X), 0)$ for evaluating $f(x)$ at points W_3 and the lower part is $\texttt{Butterfly}(g(X), \omega_{\widetilde{\mathbb{F}}}(\texttt{0x8}))$ for points $\omega_{\widetilde{\mathbb{F}}}(\texttt{0x8}) + W_3$. In this case, one `Butterfly` consists of 3 layers of butterflies, and each layer contains 4 butterflies.

We can observe that the constants in butterflies. Since the constants are values of $s_i(x)$ at the corresponding points, they are usually *smaller* than the actual values of the evaluating points. The multipliers in first layer are calculated by evaluating the degree-4 $s_2(x)$ at two $\alpha \in \{0, \omega_{\widetilde{\mathbb{F}}}(\texttt{0x8}) \in \widetilde{\mathbb{F}}_{16}\}$, resulting in the *small* multipliers $\{0, \omega_{\widetilde{\mathbb{F}}}(\texttt{0x2}) \in \mathbb{F}_4\}$. The second layer evaluates the degree-2 $s_1(x)$ at 4 points $\alpha \in \{0, \omega_{\widetilde{\mathbb{F}}}(\texttt{0x4}), \omega_{\widetilde{\mathbb{F}}}(\texttt{0x8}), \omega_{\widetilde{\mathbb{F}}}(\texttt{0xc})\}$ and results in the multipliers $\{0, \omega_{\widetilde{\mathbb{F}}}(\texttt{0x2}), \omega_{\widetilde{\mathbb{F}}}(\texttt{0x5}), \omega_{\widetilde{\mathbb{F}}}(\texttt{0x7})\}$. In the last layer, the multipliers are the elements corresponding to the particular positions $\{0, \omega_{\widetilde{\mathbb{F}}}(\texttt{0x2}), \omega_{\widetilde{\mathbb{F}}}(\texttt{0x4}), \omega_{\widetilde{\mathbb{F}}}(\texttt{0x6}), \ldots, \omega_{\widetilde{\mathbb{F}}}(\texttt{0xe})\}$ because $s_0(\alpha) = \alpha$.

6.3.3 Converting Polynomial Bases

In this section, we review the methods converting a polynomial $f(x)$ in the monomial basis to $g(X)$ in the *novelpoly* basis.

Figure 6.2: An example of computations in `Butterfly`.



The forward butterfly units for evaluating a degree-7 polynomial $f(x) = g(X) = g_0 + \cdots + g_7 X_7$ at 16 points $\{0, 1, \ldots, \omega_{\widetilde{\mathbb{F}}}(\texttt{0xf})\}$.

Since the element X_k of the *novelpoly* basis comprises products of $s_i(x)$'s, the straightforward conversion proceeds with continuing divisions of $f(x)$ by $s_i(x)$'s. One first finds the largest i such that $\deg(s_i) = 2^i < \deg(f)$ and then divide $f(x)$ by $s_i(x)$ to find f_0 and f_1 such that

$$f(x) = f_0(x) + s_i(x) \cdot f_1(x) \ .$$

Recursively divide f_0 and f_1 by s_{i-1} of lower degrees and eventually express $f(x)$ as a sum of non-repetitive products of the $s_i(x)$'s, which is the desired form for $g(X)$. Since the coefficients of $s_i(x)$ are always 1 in the Cantor basis, the division is performed with only additions. Therefore the complexity of division by one s_i depends on the number of terms of s_i and the conversion takes $O(n(\log n)^2)$ field additions(`XOR`).

Table 6.1: Variable Substitution of $s_i(x)$

$s_0(x)$	x		
$s_1(x)$	$x^2 + x$		
$s_2(x)$	$x^4 + x$	$= s_2(x) = y$	
$s_3(x)$	$x^8 + x^4 + x^2 + x$	$= s_1(y) = y^2 + y$	
$s_4(x)$	$x^{16} + x$	$= s_4(x) = z$	
$s_5(x)$	$x^{32} + x^{16} + x^2 + x$	$= s_1(z) = z^2 + z$	
$s_6(x)$	$x^{64} + x^{16} + x^4 + x$	$= s_2(z) = z^4 + z$	$= s_6(x) = w$
$s_7(x)$	$x^{128} + x^{64} + x^{32} + x^{16} + x^8 + x^4 + x^2 + x$	$= s_3(z) = z^8 + z^4 + z^2 + z$	$= s_1(w) = w^2 + w$

Lin *et al.* [LANH16] presented a basis conversion of fewer field operations by dividing $f(x)$ by $s_i(x)$ where i is only power of 2. From Eq. (6.2)

$$s_i(x) = s_{2^{k_0}}(s_{2^{k_1}}(\cdots(s_{2^{k_j}}(x))\cdots)) \ ,$$

any particular $s_i(x)$ can be express as compositions of $s_{2^k}(x)$'s, which is a polynomial of two terms. By the technique of variable substitution(see Tab. 6.1 and Algo. 7), one can finish the basis conversion with division of $f(x)$ by $s_{2^k}(x)$'s. The complexity for the conversion is $O(n \log n \log\log n)$ field additions for $f(x) \in \widetilde{\mathbb{F}}_{2^m}[x]_{<n}$.

Algorithm 7: Variable Substitution

`VarSubs(` $f(x), y$ `)` :
input : $f(x) = f_0 + f_1 x + ... + f_{n-1}x^{n-1} \in R[x]$
$y = s_i(x) = x^{2^i} + x$.
output: $h(y) = h_0(x) + h_1(x)y + \cdots + h_{m-1}(x)y^{m-1} \in R[x][y]$.
if $\deg(f(x)) < 2^i$ **then** return $h(y) \leftarrow f(x)$
Let $k \leftarrow \text{Max}(2^j)$ where $j \in \mathbb{Z}$ s.t. $\deg((x^{2^i} + x)^{2^j}) \leq \deg(f(x))$.
Let $y^k \leftarrow x^{k2^i} + x^k$.
Compute $f_0(x) + y^k \cdot f_1(x) = f(x)$ by dividing $f(x)$ by $x^{k2^i} + x^k$.
// Note the division is done by only `XOR` operations.
return `VarSubs(` $f_0(x), y$ `)` $+y^k \cdot$ `VarSubs(` $f_1(x), y$ `)` .

Algorithm 8 shows the details of basis conversion. The algorithm proceeds with

1. finding the largest $i = 2^k$ such that $2^i < \deg f$ and then performing Algo. 7(variable substitution) to express f as a power series of s_i. Note that $s_i(x) = x^{2^i} + x$ for $i = 2^k$.

2. Recursively express the series in s_i as a series in $X_j(s_i)$, where $j < 2^i$.

3. Recursively express each coefficient of $X_j(s_i)$ (which is a polynomial in x of degree $< 2^i$) as a series in X_k, where $k < 2^i$.

Note that the algorithms relies on the simple form of $s_i(x) = x^{2^i} + x$ for $i = 2^k$ instead of the representation of coefficients.

Basis Conversion: An Example

Figure 6.3 shows an example of converting a degree-15 polynomial to the *novelpoly* basis. For a straightforward conversion, one has to divide by 3 different $s_i(x)$'s, namely $s_1(x), s_2(x)$, and $s_3(x)$, which has 4 terms. However, by applying Algo. 8, one can see there are actually 4 layers of division and the number of `XOR`'s are the same in all layers.

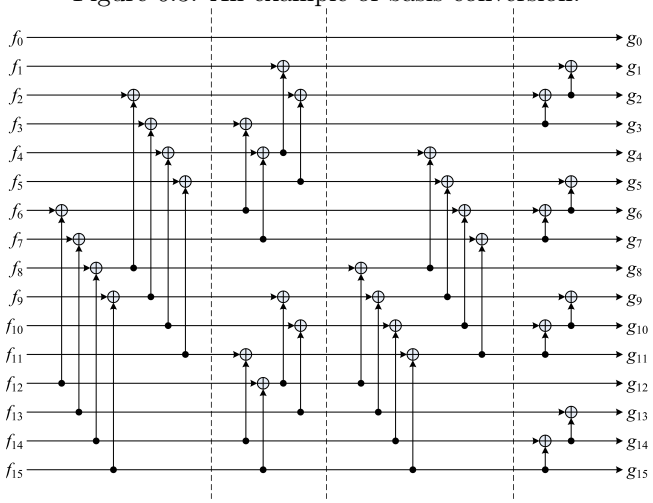
We can clarify the layers in the Fig. 6.3. The first 2 layers substitute variables into terms of $y = s_2(x) = x^4 + x$ by the Algo. 7. The first layer divides the input by $s_2(x)^2 = x^8 + x^2$ and the second layer divides the two polynomials(the quotient and the reminder) by $s_2(x) = x^4 + x$ on the high degree and low degree polynomials from the first layer. The third layer divides *one input polynomial* by $s_3(x) = y^2 + y$ *by adding between coefficients of terms differing by a factor of* $s_2(x)$*, and 4 positions apart* (see 3rd column of Table 6.1). The last layer divides by $s_1(x) = x^2 + x$ for 4 short polynomials (Last loop in Algo. 8). Note that we only do division by two terms polynomials in the conversion.

Algorithm 8: Basis conversion: monomial to *novelpoly* basis.

`BasisCvt(`$f(x)$`)` :
input : $f(x) = f_0 + f_1 x + ... + f_{n-1}x^{n-1} \in R[x]_{<n}$ in monomial basis.
output: $g(X) = g_0 + g_1 X_1 + ... + g_{n-1}X_{n-1} \in R[x]_{<n}$ in *novelpoly* basis.

if $\deg(f(x)) \leq 1$ **then** return $g(X) \leftarrow f_0 + X_1 f_1$
Let $i \leftarrow \text{Max}(2^k)$ where $k \in \mathbb{N}$ s.t. $\deg(s_i(x)) \leq \deg(f(x))$.
Let $y \leftarrow s_i(x)$.
$h(y) = h_0(x) + \cdots + h_{m-1}(x)y^{m-1} \leftarrow$ `VarSubs(` $f(x), y$`)` $\in R[x][y]$.
// s.t. $f(x) = h(s_i(x))$ and $h_j(x) \in R[x]_{<2^i}$ for $j = 0, \ldots, m-1$.
$h'(Y) = q_0(x) + q_1(x)X_{2^k} + \cdots + q_{m-1}(x)X_{(m-1)\cdot 2^k} \leftarrow$ `BasisCvt(` $h(y)$`)` .
foreach *coefficient* $q_i(x)$ *of* $h'(Y)$ **do**
  $g_i(X) \leftarrow$ `BasisCvt(` $q_i(x)$`)` .
end
return $g(X) = g_0(X) + g_1(X)X_{2^k} + ... + g_{n-1}(X)X_{(m-1)\cdot 2^k}$

Figure 6.3: An example of basis conversion.



Converting $f(x) = f_0 + \cdots + f_{15}x^{15}$ to $g(X) = g_0 + \cdots + g_{15}X_{15}$ in Algorithm 8.

6.3.4 The `addFFT` Algorithm

Algorithm 9 shows the `addFFT` algorithm for evaluating a polynomial $\underline{f} \in \widetilde{\mathbb{F}}_{2^m}[x]_{<n}$ in monomial basis at n points $\alpha + W_l$, where $n = 2^l$ and $\alpha \in \mathbb{F}_{2^m}$. The algorithm first calls the `BasisCvt` to convert the basis of polynomials and then uses the `Butterfly` to evaluate polynomials in the *novelpoly* basis. Inverse additive FFT, or `iaddFFT`, simply performs the `Butterfly` and the `BasisCvt` in reverse.

Algorithm 9: The Additive FFT Algorithm

```
1 addFFT( f(x), α) :
  input : f(x) = f_0 + f_1 x + ... + f_{n-1} x^{n-1} ∈ F̃_{2^m}[x]_{<n} .
          an extra scalar: α ∈ F̃_{2^m} .
  output: The values of f at α + W_l, where n = 2^l .
2 Compute g(X) ← BasisCvt(f(x)).
3 return Butterfly(g(X), α).
```

6.4 Implementing the Additive FFT

We discuss the techniques of implementations about the butterfly network and the basis conversion in this section.

6.4.1 Performing the Butterflies

Computing the constants in the Butterflies

The central computation of the `Butterfly` is to generate the two polynomials

$$h_0(X) \leftarrow p_0(X) + s_i(\alpha) \cdot p_1(X) \qquad \text{(6.7 revisited)}$$

and

$$h_1(X) \leftarrow h_0(X) + s_i(v_i) \cdot p_1(X) \ . \qquad \text{(6.8 revisited)}$$

Here $s_i(v_i) = 1$ for the Cantor basis. Hence, we have to identify the value of the constant $s_i(\alpha)$ for a particular butterfly. For a small butterfly network, we can hard-code the values or query a table in the software implementation. For a large butterfly network, we have to calculate the values in run-time.

By observing the Fig. 6.2, we can see the value of α corresponds its *index* of the particular position in the butterflies. The outputs of the butterfly network correspond to the values of a polynomial at $(0, 1, \omega_{\widetilde{\mathbb{F}}}(\texttt{0x2}), \omega_{\widetilde{\mathbb{F}}}(\texttt{0x3}), \ldots)$ which are precisely the *indexes* of the particular wires. We can also observe the constants of the last layer in the Fig. 6.2. The constants are $(0, \omega_{\widetilde{\mathbb{F}}}(\texttt{0x2}),$ $\omega_{\widetilde{\mathbb{F}}}(\texttt{0x4}), \ldots)$ in the last layer, which equals to $s_0(\alpha) = \alpha$.

While evaluating a polynomial $f(x) \in \widetilde{\mathbb{F}}_{2^m}[x]$ in the tower field, we have to figure out the value of the representative of α in the tower field although calculating the value $s_i(\alpha)$ for α in the Cantor basis is efficient by Eq. (6.4)

$$s_i(\alpha) = \omega(\omega^{-1}(\alpha) \gg i) \ ,$$

It requires the calculation of the field isomorphism for the value of $s_i(\alpha)$. More precisely, for performing the `addFFT` over the tower fields, we first calculate the value of $s_i(\alpha)$ in the Cantor basis and then perform a field isomorphism for the value of $s_i(\alpha)$ in the tower field. The field isomorphism is a linear transformation with the matrix in Tab. 2.1, which can be performed with the method of four Russian efficiently. In this case, we index the wires in the Cantor basis, but output values of the polynomial in the representation of the tower field at points $(\omega_{(v_i)}(0), \omega_{(v_i)}(1), \omega_{(v_i)}(\texttt{0x2}), \ldots)$. This is equal to reorder the outputs of the `Butterfly`.

We can accelerate the field isomorphism by calculating the isomorphism of the differences for the consecutive α's. This empirical trick works especially when the binary value of the α is large. The difference of two large α's are usually smaller than their original values which reducing the operations in the linear transformation when the vector contains many 0's.

Multiplications by Subfield Elements in the Butterflies

To accelerate the multiplications in butterflies, we can evaluate the polynomials at subfields, and thus the multiplication can be performed with the subfield multiplications in Sec. 3.2. Given the value of $s_i(\alpha)$, the multiplication in the butterfly are

$$h_0(X) = p_0(X) + s_i(\alpha) \cdot p_1(X) \ .$$

Given $p_1(X) \in \widetilde{\mathbb{F}}_{2^m}[x]$, we can have the subfield multiplication if $s_i(\alpha) \in \widetilde{\mathbb{F}}_{2^{m-1}}$. The situation always occurs when $\omega^{-1}(\alpha) < 2^{m/2}$. Even when the value of $\omega^{-1}(\alpha)$ is "large", the evaluation of $s_i(\alpha)$ will shorten the value by i bits in the Cantor basis. By observing Tab. 2.1, we know the most significant bit of elements in tower fields are the same with Cantor basis. Hence, the s_i shorten the values in the tower representation by i bits as well.

6.4.2 Performing the Basis Conversion

Skipping the Computation

The rule of thumb in the implementation of the `BasisCvt` is always to skip the computation by defining polynomials in *novelpoly* basis naturally. For example, Chou and Berstein had omitted the "radix conversion" in one variant of the "auth256", a message authenticated code, in [BC14]. Lin *et al.* also got rid of the basis conversion for the en/decoder of the Reed-Solomon code by defining polynomials in *novelpoly* basis in [LCH14]. The optimization relies on considering the subject for FFT carefully although it is definitely the most efficient optimization.

Reducing the memory access for block data

We focus on reducing the number of memory access for optimizing the `BasisCvt`. The memory access model is critical for optimizing algorithms on computers. The typical design of an algorithm usually assumes on free access of data while it's usually not the fact on computers. In practice, loading data from memory to CPUs takes time, and the access time has to be covered by the computation in CPUs. Otherwise, the running time for waiting data might greater than the actual computation time. The number of memory access is more critical for the process of only simple `XOR` operations, reported by Albrecht *et al.* [ABH10] for multiplying matrices over $\mathbb{F}_2$. The `BasisCvt` faces the same situation.

For reducing the number of memory access, we can combine several layers of operations in Algo. 8. It is possible since the Algo. 8 always `XOR` coefficients of higher degree to coefficients of lower degree. While same coefficients of lower degree gather coefficients from higher degree among several layers, we can combine the accumulations among layers. This optimization effectively reduces the number of memory write. For dividing polynomials whose degree greater than

$s_8(x) = x^{256} + x$, the data are moved in 256-bits units, and the effect of this optimization is more noticeable than small memory blocks.

We note that the structures between layers are even regular in the `Butterfly`. Hence, we can also perform several layers of butterflies together to increase the performance.

Techniques for bit-level data

We have different strategies for dividing $s_i(x)$'s that degree < 256.

For dividing polynomials by

$$s_7(x) = x^{128} + x^{64} + x^{32} + x^{16} + x^8 + x^4 + x^2 + x \ ,$$

we perform the division with the `PCLMULQDQ` instruction. Given $f(x) \in \mathbb{F}_2[x]_{<256}$, we want to express $f(x)$ in the expansion of $s_7(x)$. Let

$$f(x) = f_3 \cdot x^{192} + f_2 \cdot x^{128} + f_1 \cdot x^{64} + f_0 \quad \text{for } f_0, f_1, f_2, f_3 \in \mathbb{F}_2[x]_{<64} \ .$$

We have

$$\begin{aligned} f(x) &= f_3 \cdot x^{64} \cdot x^{128} + f_2 \cdot x^{128} + f_1 \cdot x^{64} + f_0 \\ &= f_3 \cdot x^{64} \cdot s_7(x) + \left[f_2 \cdot x^{64} + f_1 + f_3 \times (x^{128} - s_7(x))\right] x^{64} + f_0 \end{aligned}$$

Let $g_1 \cdot x^{64} + g_0 = f_2 \cdot x^{64} + f_1 + f_3 \cdot (x^{128} - s_7(x))$ for $g_1, g_2 \in \mathbb{F}_2[x]_{<64}$, then

$$\begin{aligned} f(x) &= f_3 \cdot x^{64} \cdot s_7(x) + g_1 \cdot x^{128} + g_0 \cdot x^{64} + f_0 \\ &= \left[f_3 \cdot x^{64} + g_1\right] \cdot s_7(x) + g_1 \times (x^{128} - s_7(x)) + g_0 \cdot x^{64} + f_0 \ . \end{aligned}$$

Hence , we can perform the calculation with 2 `PCLMULQDQ` instructions at the "$\times$" symbols.

The Technique of Bit-slice

For dividing by $s_i(x)$'s of degree < 64, we perform the basis conversion in the bit-slice manner. More precisely, we first collect 128 or 256 copies of the computation and perform the bit-level transpose for the data with the techniques in Sec. 2.4.1. After the transpose, we can perform the division with `XOR` operations in 128 or 256 parallels.

Chapter 7

Multiplication of Boolean Polynomials

The last application targets on multiplying Boolean polynomials of high degree. This chapter is based on the joint work with Wen-Ding Li, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang published in [LCK+18].

7.1 Introduction

Multiplying polynomials in the ring $\mathbb{F}_2[x]$ (Boolean polynomials) is a fundamental problem in computer science. The operation is so essential that modern CPUs even dedicate a hardware instruction to the carryless multiplication of polynomials.

To the best of our knowledge, all current fast algorithms for multiplying high-degree Boolean polynomials are based on a fast Fourier transform (FFT) algorithm. An FFT is capable of evaluating polynomials at a set of points in the underlying field efficiently. As to multiplying Boolean polynomials with FFTs, the multiplication proceeds by evaluating polynomials at these specific points, multiplying values of two polynomials point-wise, and interpolating the values back to a polynomial with an inverse FFT. Note that one has to perform the FFTs in an extension field $\mathbb{F}_{2^m}$ of $\mathbb{F}_2$ since there are only two possible points in $\mathbb{F}_2$.

In 2017, van der Hoeven and Larrieu [vdHL17] showed a new technique, named Frobenius FFT, to evaluate a polynomial in $\mathbb{F}_2[x]$ at points in $\mathbb{F}_{2^m}$ with Frobenius automorphism. The Frobenius FFT roughly runs m times faster than evaluating a polynomial in $\mathbb{F}_{2^m}[x]$ by evaluating at fewer points and deducing other values of the polynomial with Frobenius map. In their subsequent work [vdHLL17], they applied the Frobenius FFT to multiplying Boolean polynomials and resulted in a multiplier with 2 time faster than their original multiplier [HvdHL16].

In this chapter, we will show the technique of the Frobenius FFT can cooperate with the additive FFT as well. The adaption becomes to truncate some butterflies in the additive FFT. As a result, we can achieve an implementation with the best-known performance for multiplying Boolean polynomials.

7.1.1 Previous Multiplications for Boolean Polynomials

We introduce two methods for multiplying Boolean polynomials with FFTs here. In general, the multiplication algorithms convert the problem from performing an FFT in $\mathbb{F}_2[x]$ to performing in $\mathbb{F}_{2^m}[x]$ because the field $\mathbb{F}_{2^m}$ provides more points than $\mathbb{F}_2$ for evaluating polynomials.

Kronecker Substitution (KS) of Coefficients of Polynomials

Most previous works for multiplying Boolean polynomials based on Kronecker Substitution or Segmentation (KS) [GG13, Chap. 8]. Let $\mathbb{F}_2[x]_{<n}$ denotes the Boolean polynomials of degree $< n$. For computing $a \cdot b \to c \in \mathbb{F}_2[x]_{<n}$ with KS, we first partition the polynomials into $2n/m$ blocks of size $(m/2)$-bits, i.e., write

$$a = \sum_{i=0}^{n-1} a_i x^i \to \hat{a} = \sum_{i=0}^{(2\cdot n/m)-1} \hat{a}_i x^{i\cdot(m/2)} \quad \text{where } \hat{a}_i \in \mathbb{F}_2[x]_{<m/2} \ .$$

We consider each $\hat{a}_i$ as a field element in $\mathbb{F}_{2^m}$ and $\hat{a} \in \mathbb{F}_{2^m}[y]$ such that $a = \hat{a}(x^{m/2})$. Then we can perform a standard polynomial multiplication with FFTs over $\mathbb{F}_{2^m}[y]$. Note that the method prevents "overflow" from splitting the polynomials down to blocks of size $(m/2)$.

For multiplicative FFT implementations with KS, Brent *et al.* [BGTZ08] implemented mainly the Schönhage [Sch77] algorithm in the library `gf2x`. Harvey, van der Hoeven, and Lecerf [HvdHL16] presented multiplication using DFTs over the field $\mathbb{F}_{2^{60}}$, whose size of elements is close to a machine word and the size of the field allows abundant multiplicative subgroups. For additive FFT implementations, Chen *et al.* [CCK+17] presented a multiplication based on an additive FFT over binary fields with Cantor bases [Can89] [GM10].

Frobenius Cross-Sections of the Set of Evaluated Points

In 2017, van der Hoeven *et al.* [vdHLL17] presented a new multiplier with a two times improvement over their previous KS multiplier [HvdHL16]. Instead of partitioning the polynomials in $\mathbb{F}_2[x]_{<n}$ into blocks, they directly run an FFT to evaluate a polynomial in $\mathbb{F}_2[x]$ at points in $\mathbb{F}_{2^m}$ (specifically, $\mathbb{F}_{2^{60}}$ for $m = 60$).

However, they only compute values at a subset Σ_ω of size n/m, from which they can derive the values at a larger set $\Omega_n \subset \mathbb{F}_{2^m}$ (where $|\Omega_n| = n$), using the Frobenius map of $\mathbb{F}_{2^m}$ (i.e., squaring, since this is a binary field). For multiplying polynomials in $\mathbb{F}_2[x]_{<n}$ with an FFT over the field $\mathbb{F}_{2^m}$, the new method works on an FFT of size n/m instead of $2n/m$ (when using Kronecker Segmentation).

In this thesis, we use the term "Frobenius cross-section" for the set Σ_ω which induces a partition of the larger set Ω_n under the Frobenius map. We will present a new Frobenius cross-section, and one can evaluate Boolean polynomials at the corss-section efficiently with a truncated additive FFT.

7.1.2 The Practical Complexity Model

When implementing Boolean polynomial multiplication on modern computers, the software usually works on a structure with multiple bits (e.g., a binary field

of m-bits, denoted as $\mathbb{F}_{2^m}$). It is a natural result of high-performance implementations since the computer works with instructions on machine words instead on a single bit. Hence, instead of the bit complexity model, we use the algebraic complexity model, which analyzes the complexity of algebraic operations, for evaluating the complexity of algorithms.

In the algebraic complexity model, from Harvey *et al.* [HvdHL17], the best complexity for multiplying polynomials of degree $< n$ is $O(n \log n)$ *field* multiplications and $O(n \log n \log \log n)$ *field* additions by Cantor and Kaltofen [CK91]. The multiplication algorithm presented in this chapter works with the known best complexity as well.

However, our implementation only supports a *practical* length of polynomials, i.e., $n < 2^{64}$ bits. The restriction comes from the specific underlying fields, which are $\mathbb{F}_{2^{64}}$ or $\mathbb{F}_{2^{128}}$, instead of an arbitrary field.

7.1.3 Our Contributions

[vdHLL17] introduces this problem: Using a particular FFT, can we find a Frobenius cross-section that results in an efficient multiplier for Boolean polynomials, in particular, what about for additive FFTs?

We answer the question by deriving a Frobenius cross-section for additive FFT over binary fields and applying it to multiplying Boolean polynomials. More specifically, after presenting the cross-section, we show it provides sufficient points for evaluation/interpolation and how to fit the proposed cross-section into an additive FFT as well as the implementation techniques for a practically fast polynomial multiplier.

7.1.4 Chapter Overview

After the introduction, section 7.2 reviews the method of multiplying polynomials with FFTs and the Frobenius cross-section. Section 7.3 describes the cooperation between the additive FFT and Frobenius cross-section. Section 7.4 shows the benchmarks of our implementations and comparisons with previous works. Section 7.5 summaries.

7.2 Preliminaries

In this section, we review the method for multiplying Boolean polynomials of high degree and the concept of Frobenius cross-sections [vdHL17]. Although it is well known that multiplication can be done with an FFT for evaluating polynomials [CLRS09] [GG13], van der Hoven *et al.* [vdHL17] can perform the multiplication by evaluating the polynomial at a set smaller than the number of coefficients. The particular set is termed Frobenius corss-section with respect to underlying FFTs.

7.2.1 The FFT Based Multiplication of Polynomials

Given two polynomials $a(x) = a_0 + \cdots + a_{\frac{n}{2}-1}x^{\frac{n}{2}-1}$ and $b(x) = b_0 + \cdots + b_{\frac{n}{2}-1}x^{\frac{n}{2}-1} \in \mathbb{F}_2[x]_{<n}$, represented as Boolean sequences of length $\frac{n}{2}$ and n is

a power of 2 (zero-padding if necessary), we can calculate the product $c(x) = a(x) \cdot b(x) \in \mathbb{F}_2[x]_{<n}$ by evaluation and interpolation as follows:

1. Evaluate $a(x)$ and $b(x)$ at n points in $\mathbb{F}_{2^m}$ with FFTs.
2. Perform pointwise multiplications for the evaluated values.
3. Interpolate the values back to the result $c(x) \in \mathbb{F}_2[x]_{<n}$.

The complexity of the polynomial multiplication is the same as the FFT used.

7.2.2 The Frobenius cross-section

In 2017, Van der Hoeven and Larrieu [vdHL17] noted that knowing the value of a Boolean polynomial at any point in a binary field also determines the value of the said polynomial at many other points because of the Frobenius map commuting with the polynomial. In other words, let $c(x) \in \mathbb{F}_2[x]_{<n}$ and the Frobenius map

$$\phi_2 : e \mapsto e^2 \quad \text{for } e \in \mathbb{F}_{2^m} \ ,$$

then

$$c(\phi_2(e)) = \phi_2(c(e)) \quad \text{for } e \in \mathbb{F}_{2^m} \text{ and } c(x) \in \mathbb{F}_2[x] \ . \tag{7.1}$$

Thus, given the values of $c(x)$ at a set Σ, we can determine the values at the set $\phi_2(\Sigma)$. Similarly for evaluations at $\phi_2^{\circ j}(\Sigma)$, where $\phi_2^{\circ j}$ means applying ϕ_2 serially j times.

Definition 3. The *order* of v under ϕ_2 is

$$\mathrm{Ord}_{\phi_2}(v) := \min\{j : j > 0, \phi_2^{\circ j}(v) = v\} \ .$$

Similarly,

$$\mathrm{Ord}_{\phi_2}(\Sigma) := \min\{j : j > 0, \phi_2^{\circ j}(\Sigma) = \Sigma\} = \mathrm{lcm}\{\mathrm{Ord}_{\phi_2}(v) : v \in \Sigma\} \ .$$

Definition 4. (The Frobenius Cross-Section) We say a set Σ is a *Frobenius cross-section* if $\mathrm{Ord}_{\phi_2}(\Sigma) = j \in \mathbb{N}$ and Σ partitions its superset Ω under Frobenius map, i.e.,

$$\Omega = \Sigma \uplus \phi_2(\Sigma) \uplus \cdots \uplus \phi_2^{\circ(j-1)}(\Sigma)$$

where $\uplus$ denote a disjoint union, i.e., all of $\Sigma, \phi_2(\Sigma), \ldots,$ and $\phi_2^{\circ(j-1)}(\Sigma)$ are disjoint sets.

By (7.1), the value of $c(x)$ at any point in Ω can be derived from one of values at Σ.

7.3 The Multiplication with Frobenius Cross-section and Additive FFT

In this section, we present an efficient algorithm for multiplying Boolean polynomials by evaluating polynomials at a particular Frobenius cross-section with additive FFTs.

7.3.1 The Partition of Evaluating Points

Given a polynomial $a(x) \in \mathbb{F}_2[x]$ for $\deg(a) = n-1$ and $n = 2^{\ell_n}$, we present a Frobenius cross-section $\Sigma \subset \mathbb{F}_{2^m}$ for $m = 2^{\ell_m}$ in the Cantor basis for deriving n values of $a(x)$ from the values at Σ in this section.

Before we define the cross-section Σ, we first discuss the order of basis elements under ϕ_2 in the Cantor basis. Given ϕ_2 is the square operation over $\mathbb{F}_{2^m}$ and the Cantor basis $(v_i)_{i=0,\ldots,m-1}$, we have

$$\phi_2(v_0) = v_0$$

and

$$\phi_2(v_i) = v_i^2 = v_i + v_{i-1} \quad \text{for } i > 0 \ .$$

In [LCK+18], Li *et al.* showed the order of the basis element v_i under ϕ_2

$$\mathrm{Ord}_{\phi_2}(v_i) = 2 \cdot 2^{\lfloor \log_2 i \rfloor} \quad \text{for } i > 0 \ . \tag{7.2}$$

In other words, $\mathrm{Ord}_{\phi_2}(v_i) = m$ if v_i is in the set $\mathbb{F}_{2^m} \setminus \mathbb{F}_{2^{m/2}}$ of a minimum m. Hence, the maximum order for elements in $\mathbb{F}_{2^m}$ is

$$m = \mathrm{Ord}_{\phi_2}(v_i) \quad \text{for } v_i \in \{v_{m/2}, \ldots, v_{m-1}\} \ .$$

We define the set Σ with the maximum order under ϕ_2

$$\Sigma := v_{l+m/2} + W_l \quad \text{where } l = (\ell_n - \ell_m) < m/2 \text{ and } l \geq 0 \tag{7.3}$$

and its superset

$$\Omega := \Sigma \cup \phi_2(\Sigma) \cup \phi_2^{\circ 2}(\Sigma) \cup \cdots \cup \phi_2^{\circ (m-1)}(\Sigma) \ . \tag{7.4}$$

Our goal is to show the following proposition.

Proposition 2. *Σ is a Frobenius cross-section of Ω and $|\Omega| = n$.*

Proof. We start by counting the order of Σ. First, by induction, we show ϕ_2 maps W_l to itself, i.e.,

$$\phi_2(W_l) = W_l \ .$$

Clearly, $\phi_2(W_0) = W_0$. Given $\phi_2(W_i) = W_i$, we have

$$\phi_2(W_{i+1}) = \phi_2(v_i + W_i) \cup \phi_2(W_i) = ((v_i + v_{i-1}) + W_i) \cup W_i = (v_i + W_i) \cup W_i = W_{i+1} \ .$$

Hence, the order of Σ is decided mainly by the element $v_{l+m/2}$. At first glance, we have $\mathrm{Ord}_{\phi_2}(v_{l+m/2}) = m$ from Eq. 7.2. However, since $\Sigma = v_{l+m/2} + W_l$, we have to deal with effect from W_l. To analyze the effect of W_l, we split the vector representation of elements in Σ into high- and low-bits parts. While applying ϕ_2 to $v_{l+m/2}$ for j times, let the vector

$$\phi_2^{\circ j}(v_{l+m/2}) = a + b$$

where $a \in W_l$ is equal to least l dimensions(bits) of $\phi_2^{\circ j}(v_{l+m/2})$ and b is the remainder corresponding to the high-bits part. Then $W_l + a = W_l$ since $a \in W_l$.

And, by omitting the least l dimensions of b, the order for the high-bits part of $\phi_2^{\circ j}(v_{l+m/2})$ is still m

$$\mathrm{Ord}_{\phi_2}(\omega(\omega^{-1}(b) \gg l)) = \mathrm{Ord}_{\phi_2}(v_{m/2}) = m \ .$$

Hence, $\mathrm{Ord}_{\phi_2}(\Sigma) = m$.

Last we argue about the disjoint sets. While continuously applying ϕ_2 to Σ, from the discussion of the order for Σ, the W_l absorbs the a parts of $\phi_2^{\circ j}(v_{l+m/2})$, and only the b parts of $\phi_2^{\circ j}(v_{l+m/2})$ changes. Hence, $\phi_2^{\circ j}(\Sigma)$ are disjoint sets for $j < m$, and Σ is a Frobenius cross-section of Ω.

Since the size of Σ is clearly

$$n_p := |\Sigma| = |W_l| = 2^l = n/m \ , \tag{7.5}$$

we know the size of Ω

$$|\Omega| = \mathrm{Ord}_{\phi_2}(\Sigma) \cdot |\Sigma| = m \cdot \frac{n}{m} = n \ .$$

□

Now we define a linear map $E_\Sigma : \mathbb{F}_2[x]_{<n} \to \mathbb{F}_{2^m}^{n_p}$, evaluating a polynomial $a \in \mathbb{F}_2[x]_{<n}$ at the cross-section Σ, i.e.,

$$E_\Sigma : a(x) \mapsto \{a(v_{l+m/2} + u) : u \in W_l\} \ . \tag{7.6}$$

Clearly, E_Σ evaluates $a(x)$ at $n_p = n/m$ points which are fewer than the number of coefficients n. However, since the points are in $\mathbb{F}_{2^m}$, the size of input and output space are the same $n = m \cdot n/m$ bits.

The following proposition summarizes that the evaluation E_Σ contains sufficient information for interpolation.

Proposition 3. *E_Σ is a bijection between $\mathbb{F}_2[x]_{<n}$ and $\mathbb{F}_{2^m}^{n_p}$.*

Proof. Let E_Ω be the map that evaluates polynomials at the set Ω. Since E_Ω evaluates polynomials at n points, it is clearly a bijection between $\mathbb{F}_2[x]_{<n}$ and $\mathbb{F}_{2^m}^n$. Σ can derive the full Ω with the linear operator ϕ_2 and vice versa. By Eq.(7.1), E_Ω can be derived from E_Σ with the linear operator ϕ_2. □

Last, we discuss how the maximum length n of polynomials is bound by the size of Ω and the underlying field $\mathbb{F}_{2^m}$. From Prop. 2, $|\Omega| = \mathrm{Ord}_{\phi_2}(\Sigma) \cdot |\Sigma| = m \cdot |\Sigma|$. Since the maximum

$$|\Sigma| = |W_{\frac{m}{2}-1}| = 2^{\frac{m}{2}-1} \quad \text{for } l < m/2$$

in Eq. 7.3, the maximum

$$|\Omega| = m \cdot 2^{\frac{m}{2}-1} = \frac{m}{2} \cdot 2^{m/2} \ .$$

Therefore, given a particular field $\mathbb{F}_{2^m}$ and the specific cross-section Σ, the maximum supported length of polynomials is

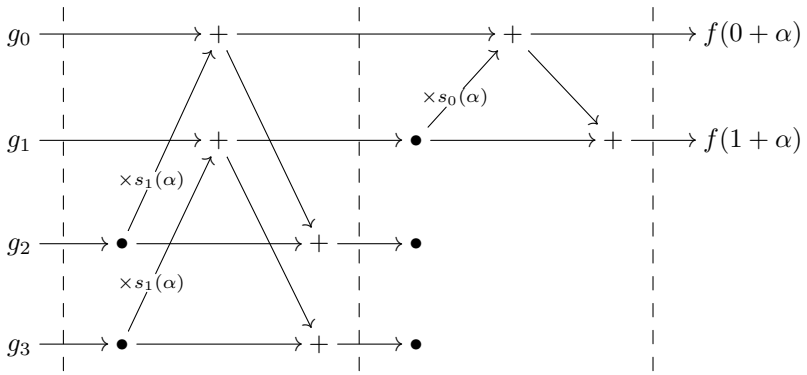
$$n \leq \frac{m}{2} \cdot 2^{m/2} \ .$$

7.3.2 Truncated Additive FFT

In this section, we show how to evaluate a polynomial $a(x) \in \mathbb{F}_2[x]_{<n}$ at Σ, i.e., performing E_Σ with the additive FFT.

To perform E_Σ, we simply call the `addFFT`(Algo. 9) with two inputs $a(x)$ and $\alpha = v_{l+m/2}$. However, since we presume the degree of $a(x)$ is $n-1$, the `addFFT` outputs the values of $a(x)$ at $v_{l+m/2} + W_{\ell_n}$, containing n points, instead of Σ of n_p points. Since the Σ is a subset of $v_{l+m/2} + W_{\ell_n}$, the desired values of $a(x)$ at Σ is a subset of outputs. Hence, we can collect the desired values from the outputs of the `addFFT`. For more efficiency, however, we have to truncate the undesired computations as well in the `addFFT` — the so-called truncated FFT. Among the two main components of the `addFFT`, the computations in `BasisCvt` are indispensable since it converts the basis of the polynomial $a(x)$. Hence, we seek the `Butterfly` for the possible savings.

Figure 7.1: An example of truncated `Butterfly`.



Half contents are truncated in a butterfly network after the first layer.

Figure 7.1 shows an example of the truncated `Butterfly` for evaluating a degree-3 polynomial $f(x) = g_0 + g_1X_1 + g_2X_2 + g_3X_3$ at 2 points $\{\alpha, \alpha+1\}$. There are two layers(recursions) of butterflies in the computation of `Butterfly`. We can truncate the half contents after the first layer since only 2 values are required.

Similar to the example, to perform E_Σ, we pretend to evaluate $a(x)$ at a larger set $v_{l+m/2} + W_{\ell_n}$ with the `Butterfly`. However, after ℓ_m layers of butterflies, the computations for the values at Σ aggregates to the first n/m parts of the following layers, and we can thus truncate the rest.

Last, we note that $v_{l+m/2} + W_{\ell_n} \neq \Omega$ although the size of the two sets is the same.

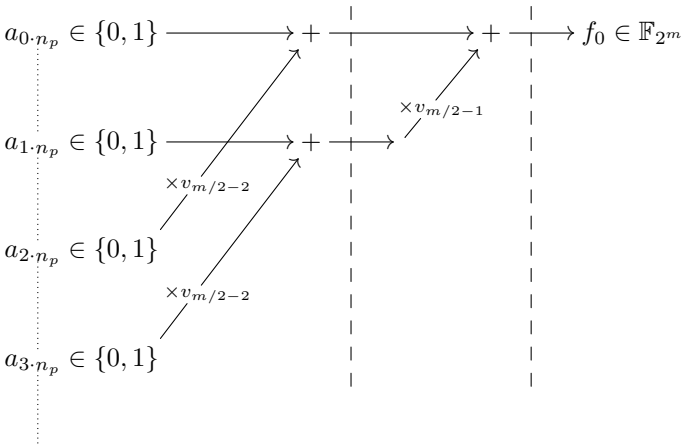
7.3.3 Encoding: the First ℓ_m Layers of the Truncated `Butterfly`

In this section, we show how to perform the ℓ_m layers of the truncated `Butterfly` with a linear transformation.

After performing the first ℓ_m layers of the `Butterfly` in E_Σ, the temporary results will expand by m times because the inputs are 1-bit data and the mul-

tiplication constants in butterflies are m-bits elements of $\mathbb{F}_{2^m}$. However, since we truncate the temporary results to the factor of $1/m$, the space requirement remains unchanged after the data truncation. Hence, we design a process, the $\texttt{Encode}_m$[1], to prevent the data expansion while performing the first ℓ_m layers of butterflies.

Figure 7.2: An example for the first temporary result of 2-layers butterflies



The multiplication constants in the butterflies are $s_{l+2}(v_{l+m/2}) = v_{m/2-2}$ and $s_{l+1}(v_{l+m/2}) = v_{m/2-1}$ for the first and second layers respectively.

Before defining the $\texttt{Encode}_m$, we can show some intuitions of the $\texttt{Encode}_m$ through an example. Figure 7.2 illustrates the the computations of the first temporary result for 2 layers of the `Butterfly`. The `Butterfly` evaluates a polynomial $a(X) = a_0 + a_1X_1 + \cdots + a_{n-1}X_{n-1} \in \mathbb{F}_2[x]_{<n}$ at the set $v_{l+m/2} + W_{\ell_n}$. We can see the first temporary value f_0 is a linear combination of 4 particular bits in inputs

$$\begin{aligned} f_0 &= a_0 + s_{l+1}(v_{l+m/2}) \cdot a_{n_p} + s_{l+2}(v_{l+m/2}) \cdot a_{2\cdot n_p} + s_{l+1}(v_{l+m/2}) \cdot s_{l+2}(v_{l+m/2}) \cdot a_{3\cdot n_p} \\ &= a_0 + v_{m/2-1} \cdot a_{n_p} + v_{m/2-2} \cdot a_{2\cdot n_p} + v_{m/2-1} \cdot v_{m/2-2} \cdot a_{3\cdot n_p} \ . \end{aligned}$$

The scalars are $1, v_{m/2-1}, v_{m/2-2}$, and $v_{m/2-1} \cdot v_{m/2-2}$ for $a_0, a_{n_p}, a_{2\cdot n_p}$, and $a_{3\cdot n_p}$ respectively. As for the storage space, if $m = 4$, we have $f_0 \in \mathbb{F}_{2^4}$ requiring 4-bits storage which equals to the storage of its 4 contributors $(a_0, a_{n_p}, a_{2\cdot n_p}, a_{3\cdot n_p})$.

We can generalize the example to the case of ℓ_m layers. For the first ℓ_m layers, the multiplication constants in the butterflies are the evaluation of $(s_{l+\ell_m}, \ldots, s_{l+1})$ at the same point $v_{l+m/2}$. By Eq. (6.3), we can list the constants in reverse order of layers as

$$(s_{l+1}(v_{l+m/2}), \ldots, s_{l+\ell_m}(v_{l+m/2})) = (v_{m/2-1}, \ldots, v_{m/2-\ell_m}) \ .$$

We note that the constants are independent of the size of inputs n. In other words, for a given m, the constants are always $(v_{m/2-\ell_m}, \ldots, v_{m/2-1})$ with respect to consecutive layers of butterflies.

[1] and its reverse process, the $\texttt{Decode}_m$.

We can further analyze the multiplication constants for distinct inputs. The constants for j-th input is

$$r_j = \prod_{k=0}^{\ell_m - 1} (v_{m/2-1-k})^{j_k} \quad \text{, for } j = j_0 + j_1 \cdot 2 + \cdots + j_{\ell_m - 1} \cdot 2^{\ell_m - 1} \ . \tag{7.7}$$

Hence, we define the $\texttt{Encode}_m$ to be the linear map for the first ℓ_m layers of the truncated `Butterfly` as

$$\texttt{Encode}_m : \begin{cases} \mathbb{F}_2[x]_{<n} & \to & \mathbb{F}_{2^m}[x]_{<n_p} \\ a(X) & \mapsto & f(X) = f_0 + \cdots + f_{n_p-1} X_{n_p-1} \ , \end{cases} \tag{7.8}$$

where

$$f_i = \sum_{j=0}^{m-1} r_j \cdot a_{j \cdot n_p + i} = \begin{bmatrix} r_0 \in \mathbb{F}_2^m, & r_1 \in \mathbb{F}_2^m, & \ldots, & r_{m-1} \in \mathbb{F}_2^m \end{bmatrix} \cdot \begin{bmatrix} a_{0 \cdot n_p + i} \\ a_{1 \cdot n_p + i} \\ \vdots \\ a_{(m-1) \cdot n_p + i} \end{bmatrix}$$

is a linear combination of inputs $(a_{0 \cdot n_p + i}, \ldots, a_{(m-1) \cdot n_p + i})$ with the constants $(r_0, \ldots, r_{m-1})$. We can calculate the linear combination in a form of $m \times m$ matrix-vector product with the algorithm in Sec. 2.4.2.

The details of the $\texttt{Encode}_m$ is listed in Algo. 10.

Algorithm 10: The $\texttt{Encode}_m$ algorithm

$\texttt{Encode}_m(\ a(X))$:
input : $a(X) = a_0 + a_1 X_1 + \cdots + a_{n-1} X_{n-1} \in \mathbb{F}_2[x]_{<n}$.
output: $f(X) = f_0 + \cdots + f_{n_p-1} X_{n_p-1} \in \mathbb{F}_{2^m}[x]_{<n_p}$ where $n_p = n/m$.
for $i \in \{0, \ldots, n_p - 1\}$ **do**
  Collect $(a_{i+0 \cdot n_p}, a_{i+1 \cdot n_p}, \ldots, a_{i+(m-1) \cdot n_p})$.
  Compute $f_i \leftarrow \sum_{j=0}^{m-1} r_j \cdot a_{j \cdot n_p + i}$,where r_j is defined in Eq. (7.7) .
end
return $(f_0, f_1, \ldots, f_{n_p-1})$.

7.3.4 Multiplying Boolean polynomials

In this section, we sum up the processes for multiplying Boolean polynomials. Algorithm 11 lists the details for multiplying Boolean polynomials. It is basically the general multiplication in Sec. 7.2 with a modified `addFFT` comprising a `BasisCvt`, an $\texttt{Encode}_m$, and a `Butterfly`.

The modified `addFFT` process performs E_Σ for the input polynomials. To evaluate a polynomial $a(x) \in \mathbb{F}_2[x]_{<n}$ at n_p points Σ in $\mathbb{F}_{2^m}$, we first perform the `BasisCvt` for converting $a(x)$ to $a(X)$ in *novelpoly* basis. Then we treat each coefficient of $a(X)$ as an element in $\mathbb{F}_{2^m}$ and pretend to perform a virtual `Butterfly` at points $v_{l+m/2} + V_{\ell_n}$. The $\texttt{Encode}_m$ process performs the first ℓ_m layers of the virtual `Butterfly` and truncates temporary results to the first $1/m$ fraction. We start a real `Butterfly` on the outputs of the $\texttt{Encode}_m$. The `Butterfly` evaluates a polynomial in $\mathbb{F}_{2^m}[x]_{<n_p}$ at Σ.

Algorithm 11: The multiplication of Boolean polynomials

`Polymul` $(a(x), b(x))$:
input : $a(x) = a_0 + \cdots + a_{n/2-1}x^{n/2-1} \in \mathbb{F}_2[x]_{<n/2}$.
$b(x) = b_0 + \cdots + b_{n/2-1}x^{n/2-1} \in \mathbb{F}_2[x]_{<n/2}$.
output: $c(x) = a(x) \cdot b(x) = c_0 + \cdots + c_{n-1}x^{n-1} \in \mathbb{F}_2[x]_{<n}$
Compute $a(X) \in \mathbb{F}_2[x]_{<n/2} \leftarrow$ `BasisCvt`$(a(x))$.
Compute $f(X) \in \mathbb{F}_{2^m}[x]_{<n_p} \leftarrow$ `Encode`$_m(a(X))$.
Compute $[\hat{a}_1, \ldots, \hat{a}_{n_p}] \in \mathbb{F}_{2^m}^{n_p} \leftarrow$ `Butterfly`$(f(X), v_{l+m/2})$.
Compute $b(X) \in \mathbb{F}_2[x]_{<n/2} \leftarrow$ `BasisCvt`$(b(x))$.
Compute $g(X) \in \mathbb{F}_{2^m}[x]_{<n_p} \leftarrow$ `Encode`$_m(b(X))$.
Compute $[\hat{b}_1, \ldots, \hat{b}_{n_p}] \in \mathbb{F}_{2^m}^{n_p} \leftarrow$ `Butterfly`$(g(X), v_{l+m/2})$.
Compute $\hat{c} = [\hat{c}_1 \leftarrow \hat{a}_1 \cdot \hat{b}_1, \ldots, \hat{c}_{n_p} \leftarrow \hat{a}_{n_p} \cdot \hat{b}_{n_p}]$.
Compute $h(X) \in \mathbb{F}_{2^m}[x]_{<n_p} \leftarrow$ `iButterfly`$(\hat{c}, v_{l+m/2})$.
Compute $c(X) \in \mathbb{F}_2[x]_{<n} \leftarrow$ `Decode`$_m(h(X))$.
Compute $c(x) \in \mathbb{F}_2[x]_{<n} \leftarrow$ `iBasisCvt`$(c(X))$.
return $c(x)$.

7.4 The Implementation and Benchmarks

7.4.1 The Implementation

In this section, we describe the implementation of the `Polymul`(Algo. 11).

The `Butterfly` and the `BasisCvt`

The implementations of the `Butterfly` and `BasisCvt` are similar with Sec. 6.4 except the underlying fields. In this chapter, we choose the width of fields m to be 64 and 128 for supporting the products of 2^{32}- and 2^{64}-bits polynomials. The corresponding fields are represented as $\mathbb{F}_{2^{64}} := \mathbb{F}_2[x]/\left(x^{64} + x^4 + x^3 + x + 1\right)$ and $\mathbb{F}_{2^{128}} := \mathbb{F}_2[x]/\left(x^{128} + x^7 + x^2 + x + 1\right)$ respectively, and the multiplications are implemented with the `PCLMULQDQ` instructions in Algo. 3 and 4 in Sec. 3.4.

Since the working fields are in different representations from the Cantor basis, we have to perform the field isomorphism for converting the constants from the Cantor basis to the values in the desired representations of the $\mathbb{F}_{2^{64}}$ and $\mathbb{F}_{2^{128}}$. Following the technique in Sec. 6.4.1, we also use the method of the difference of constants in the `Butterfly` to accelerate the computation of the field isomorphism.

The implementation of the `BasisCvt` is the same as Sec. 6.4.2. We note that the conversion manipulates the bit-level data. Hence, after the `BasisCvt`, the data are in a bit-sliced form which accelerates the data collection in the implementation of the `Encode`$_m$.

Implementing the `Encode`$_m$

The two central operations of the `Encode`$_m$ are to (1) collect the coefficients of inputs $(a_{j \cdot n_p + i})_{j=0,\ldots,m-1}$ and (2) compute $f_i \leftarrow \sum_{j=0}^{m-1} r_j \cdot a_{j \cdot n_p + i}$.

For collecting the m-bits inputs $(a_{j\cdot n_p+i})_{j=0,\ldots,m-1}$ efficiently on computers, we fetch m machine words of length w-bits instead of m separated bits. The component $a_{j\cdot n_p+i}$, for example, locates in the i-th bit of the j-th word. With an $m \times w$ matrix transpose, we can collect the inputs(Line 3) for w continuous indexes of the loop. The matrix transpose are implemented with the method in Sec. 2.4.1.

For computing coefficients $f_i \in \mathbb{F}_{2^m}$ at Line 4 in Algo. 10, the SIMD matrix-vector multiplication in Sec. 2.4.2 are applied to parallelize the processes.

7.4.2 Benchmarks

We benchmark our software[2] with experiments on multiplying random Boolean polynomials for various lengths. Although the software is actually a constant-time implementation, i.e., the running time is independent of input data, we report the average time of 100 executions. The experiments are performed on the Intel Haswell architecture, which is our targeting platform. Our hardware is Intel Xeon E3-1245 v3 @3.40GHz with turbo boost disabled and 32 GB DDR3@1600MHz memory. The OS is ubuntu version 1604, Linux version 4.4.0-78-generic and the compiler is gcc: 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.4).

Figure 7.3: Benchmarks of multiplications in $\mathbb{F}_2[x]$ on Intel Xeon E3-1245 v3 @ 3.40GHz

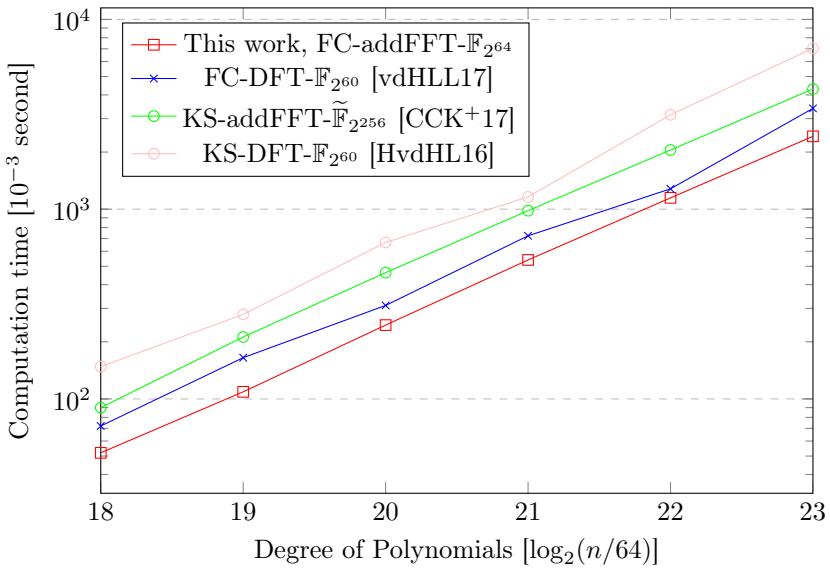


Figure 7.3 shows the results of our experiments and the comparisons with previous implementations. The figure shows the running time verse degree of polynomials both in logarithm scales. The "FC" and "KS" stands for Frobenius

[2]The software is in `https://github.com/fast-crypto-lab/bitpolymul2` .

Table 7.1: Benchmarks of multiplications in $\mathbb{F}_2[x]$ on Intel Xeon E3-1245 v3 @ 3.40GHz (10^{-3} sec.)

$\log_2 n/64$	16	17	18	19	20	21	22	23
This work, $\mathbb{F}_{2^{64}}$ [a]	12	25	52	109	245	540	1147	2420
This work, $\mathbb{F}_{2^{128}}$ [a]	13	28	58	123	273	589	1248	2641
DFT, $\mathbb{F}_{2^{60}}$ [vdHLL17] [b]	15	32	72	165	311	724	1280	3397
KS-$\widetilde{\mathbb{F}}_{2^{256}}$ [CCK+17] [c]	20	41	93	216	465	987	2054	4297
KS-$\mathbb{F}_{2^{128}}$ [CCK+17] [c]	25	53	115	252	533	1147	2415	5115
KS-$\mathbb{F}_{2^{60}}$ [HvdHL16] [d]	29	64	148	279	668	1160	3142	7040
`gf2x` [BGTZ08] [e]	26	59	123	285	586	1371	3653	7364

[a] Version 1656d5e. `https://github.com/fast-crypto-lab/bitpolymul2`
[b] SVN r10681. Available from `svn://scm.gforge.inria.fr/svn/mmx`
[c] Version c13769d. `https://github.com/fast-crypto-lab/bitpolymul`
[d] SVN r10663. Available from `svn://scm.gforge.inria.fr/svn/mmx`
[e] Version 1.2. Available from `http://gf2x.gforge.inria.fr/`

The implementations in upper table base on Frobenius cross-sections and the lower implementations are with Kronecker substitution.

cross-section and Kronecker substitution respectively. More details about the results can be found in Tab. 7.1.

The result shows that our implementations outperform all previous implementations. Among our implementations, the version of $\mathbb{F}_{2^{64}}$ is faster than $\mathbb{F}_{2^{128}}$ for more efficient multiplications over underlying fields. However, the version of $\mathbb{F}_{2^{128}}$ supports polynomials of higher degree. From the figure, we can see the same tendency among all data. It suggests that these algorithms work roughly in the same complexity level while our implementation, however, works with lowest hidden constant. We can also see the straight lines for additive FFT based algorithms, but the line turns slightly for the multiplicative algorithms. It is caused from that polynomials with terms of 2 powers are not optimal for particular sizes of multiplicative groups. Lastly, from the values in the table, we can see the FP implementations lead KS implementations about the factor of two, which is consistent with the conclusion of [vdHLL17].

7.5 Summary

We have shown the new algorithm for multiplying Boolean polynomials of high degree as well as its implementation with SIMD instructions. The new algorithm is based on evaluating polynomials at the Frobenius cross-section $\Sigma = v_{l+m/2} + W_l$ with the truncated additive FFT. This form of the cross-section fits the additive FFT particularly well. A new process $\texttt{Encode}_m$ accelerates the `Butterfly` by performing the ℓ_m layers of butterflies as matrix-vector multiplications and truncating the undesired results for the `Butterfly`.

For implementing the algorithm, we use the efficient memory access models for the `Butterfly` and the `BasisCvt` and the SIMD implementation of the critical components(e.g., bit-matrix transpose and bit-matrix multiplication in the $\texttt{Encode}_m$). The multiplication in underlying fields is also designed to utilize the

PCLMULQDQ instruction. At last, the experiments show our software outperforms all previous implementations to the best of our knowledge.

Bibliography

[ABH10] Martin R. Albrecht, Gregory V. Bard, and William Hart. Algorithm 898: Efficient multiplication of dense matrices over GF(2). *ACM Trans. Math. Softw.*, 37(1):9:1–9:14, 2010.

[AH74] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[AJ86] Leonard M. Adleman and Hendrik W. Lenstra Jr. Finding irreducible polynomials over finite fields. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 350–355. ACM, 1986.

[Anv11] H. Peter Anvin. The mathematics of RAID-6, 2011. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`.

[ASI08] A.A. Al-Shaikhi and J. Ilow. Packet loss recovery codes based on vandermonde matrices and shift operators. In *Information Theory, 2008. ISIT 2008. IEEE International Symposium on*, pages 1058–1062, July 2008.

[BC14] Daniel J. Bernstein and Tung Chou. Faster binary-field multiplication and faster binary-field macs. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 2014.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastian Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2013. Document ID: e801a97c500b3ac879d77bcecf054ce5, `http://cryptojedi.org/papers/#mcbits`.

[BD08] Johannes Buchmann and Jintai Ding, editors. *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*. Springer, 2008.

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[Ber08] Daniel J Bernstein. Fast multiplication and its applications. *Algorithmic number theory*, 44:325–384, 2008.

[BFSY05] M. Bardet, J.-C. Faugère, B. Salvy, and B.-Y. Yang. Asymptotic expansion of the degree of regularity for semi-regular systems of equations. In P. Gianni, editor, *MEGA 2005 Sardinia (Italy)*, 2005.

[BGP06] Côme Berbain, Henri Gilbert, and Jacques Patarin. QUAD: A practical stream cipher with provable security. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2006.

[BGTZ08] Richard P Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in gf (2)(x). *Lecture Notes in Computer Science*, 5011:153–166, 2008.

[BL16] Daniel J. Bernstein and Tanja Lange. eBACS: Ecrypt benchmarking of cryptographic systems. `http://bench.cr.yp.to`, July 2016. Accessed May 10, 2017.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.

[Can89] David G. Cantor. On arithmetical algorithms over finite fields. *J. Comb. Theory Ser. A*, 50(2):285–300, March 1989.

[CCC+08] Anna Inn-Tung Chen, Chia-Hsin Owen Chen, Ming-Shing Chen, Chen-Mou Cheng, and Bo-Yin Yang. Practical-sized instances of multivariate PKCs: Rainbow, TTS, and ℓIC-derivatives. In Buchmann and Ding [BD08], pages 95–108.

[CCC+09] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate PKCs on modern x86 CPUs. In *CHES 2009*, pages 33–48, Lausanne, Switzerland, September 2009.

[CCK+17] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Faster multiplication for long binary polynomials. *CoRR*, abs/1708.09746, 2017.

[CJL+16] L. Chen, S. Jordan, Y.K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography. `https://doi.org/10.6028/NIST.IR.8105`, 2016.

[CK91] David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991.

[CKPS00] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Bart Preneel, ed., Springer, 2000. Extended Version: `http://www.minrank.org/xlfull.pdf`.

[CLP+18] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang, and Chen-Mou Cheng. Implementing 128-bit secure MPKC signatures. *IEICE Transactions*, 101-A(3):553–569, 2018.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[CYC13] Ming-Shing Chen, Bo-Yin Yang, and Chen-Mou Cheng. Raidq: A software-friendly, multiple-parity raid. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, Berkeley, CA, 2013. USENIX.

[DS05] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *Conference on Applied Cryptography and Network Security — ACNS 2005*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2005.

[DY08] Jintai Ding and Bo-Yin Yang. Multivariate public key cryptography. In *Post Quantum Cryptography* (Daniel J. Bernstein, Johannes Buchmann, Erik Dahmen, eds.), pages 193–241. Springer-Verlag Berlin, 1st edition, 2008. ISBN 3-540-88701-6.

[DYC+08] Jintai Ding, Bo-Yin Yang, Chia-Hsin Owen Chen, Ming-Shing Chen, and Chen-Mou Cheng. New differential-algebraic attacks and reparametrization of rainbow. In *Applied Cryptography and Network Security*, volume 5037 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2008. cf. `http://eprint.iacr.org/2008/108`.

[Fau02] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *International Symposium on Symbolic and Algebraic Computation — ISSAC 2002*, pages 75–83. ACM Press, July 2002.

[GG13] Joachim von zur Gathen and Jrgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 3rd edition, 2013.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979. ISBN 0-7167-1044-7 or 0-7167-1045-5.

[GK14] Shay Gueron and Michael E. Kounavis. Intel(r) carry-less multiplication instruction and its usage for computing the gcm mode(rev.2.02), April 2014. `https://software.intel.com/sites/default/files/managed/72/cc/clmul-wp-rev-2.02-2014-04-20.pdf`.

[GM10] Shuhong Gao and Todd D. Mateer. Additive fast fourier transforms over finite fields. *IEEE Trans. Information Theory*, 56(12):6265–6272, 2010.

[GRU14] S.M. Gunther, M. Riemensberger, and W. Utschick. Efficient gf arithmetic for linear network coding using hardware simd extensions. In *Network Coding (NetCod), 2014 International Symposium on*, pages 1–6, June 2014.

[HvdHL16] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Fast polynomial multiplication over $\mathbb{F}_{2^{60}}$. In Sergei A. Abramov, Eugene V. Zima, and Xiao-Shan Gao, editors, *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016*, pages 255–262. ACM, 2016.

[HvdHL17] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Faster polynomial multiplication over finite fields. *J. ACM*, 63(6):52:1–52:23, 2017.

[Int15] Intel. Intel architecture instruction set extensions programming reference, August 2015. `https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf`.

[LANH16] Sian-Jheng Lin, Tareq Y. Al-Naffouri, and Yunghsiang S. Han. Fft algorithm for binary extension finite fields and its application to reed–solomon codes. *IEEE Trans. Inf. Theor.*, 62(10):5343–5358, October 2016.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to reed-solomon erasure codes. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 316–325. IEEE Computer Society, 2014.

[LCK+18] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast fourier transform. In Manuel Kauers, Alexey Ovchinnikov, and Éric Schost, editors, *Proceedings of the 2018 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2018, New York, NY, USA, July 16-19, 2018*, pages 263–270. ACM, 2018.

[LF04] Jérome Lacan and Jérome Fimes. Systematic mds erasure codes based on vandermonde matrices. *IEEE Communications Letters*, 8(9):570–572, 2004.

[LK16] Daniel Lemire and Owen Kaser. Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptographic Engineering*, 6(3):171–185, 2016.

[LLY08] Feng-Hao Michael Liu, Chi-Jen Lu, and Bo-Yin Yang. Secure PRNGs from specialized polynomial maps over any GF(q). In Buchmann and Ding [BD08], pages 95–106.

[LN86] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, New York, NY, USA, 1986.

[Nat01] National Institute of Standards and Technology. Announcing the advanced encryption standard (aes), 2001. federal information processing standards publication 197. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2001.

[oST16] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf`.

[PBB10] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. Selecting parameters for the rainbow signature scheme. In Nicolas Sendrier, editor, *PQCrypto*, volume 6061 of *Lecture Notes in Computer Science*, pages 218–240. Springer, 2010.

[PD05] James S. Plank and Ying Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software: Practice and Experience*, 35(2):189–194, February 2005.

[PGM13] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *FAST 2013*, San Jose, CA, USA, February 2013.

[Pla97] James S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software: Practice and Experience*, 27(9):995–1012, September 1997.

[Riz97] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Comput. Commun. Rev.*, 27(2):24–36, April 1997.

[Sch77] Arnold Schönhage. Schnelle multiplikation von polynomen über körpern der charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977.

[Sch11] Peter Schwabe. *High-Speed Cryptography and Cryptanalysis*. PhD thesis, Eindhoven University of Technology, 2011. `http://cryptojedi.org/users/peter/thesis/`.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[vdHL17] Joris van der Hoeven and Robin Larrieu. The frobenius FFT. In Michael A. Burr, Chee K. Yap, and Mohab Safey El Din, editors, *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017*, pages 437–444. ACM, 2017.

[vdHLL17] Joris van der Hoeven, Robin Larrieu, and Grégoire Lecerf. Implementing fast carryless multiplication. In Johannes Blömer, Ilias S. Kotsireas, Temur Kutsia, and Dimitris E. Simos, editors, *Mathematical Aspects of Computer and Information Sciences - 7th International Conference, MACIS 2017, Vienna, Austria, November 15-17, 2017, Proceedings*, volume 10693 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2017.

[War12] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[Wol04] Christopher Wolf. Efficient public key generation for HFE and variations. In Ed Dawson and Wolfgang Klemm, editors, *Cryptographic Algorithms and their Uses - 2004, International Workshop, Gold Coast, Australia, July 5-6, 2004, Proceedings*, pages 78–93. Queensland University of Technology, 2004.

[YC05] Bo-Yin Yang and Jiun-Ming Chen. Building secure tame-like multivariate public-key cryptosystems: The new TTS. In *ACISP 2005*, volume 3574 of *Lecture Notes in Computer Science*, pages 518–531. Springer, July 2005.

[YCBC07] Bo-Yin Yang, Owen Chia-Hsin Chen, Daniel J. Bernstein, and Jiun-Ming Chen. Analysis of `QUAD`. In Alex Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 290–307. Springer, 2007.

[YCC04] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas Courtois. On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In *ICICS 2004*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer, Oct. 2004.