Frobenius Additive Fast Fourier Transform

Wen-Ding Li
Research Center for Information Technology Innovation, Academia Sinica, Taiwan
thekev@crypto.tw

Ming-Shing Chen
Research Center for Information Technology Innovation, Academia Sinica, Taiwan
mschen@crypto.tw

Po-Chun Kuo
Department of Electrical Engineering, National Taiwan University, Taiwan
kbj@crypto.tw

Chen-Mou Cheng
Graduate School of Engineering, Osaka University, Japan
ccheng@cy2sec.comm.eng.osaka-u.ac.jp

Bo-Yin Yang
Institute of Information Science, Academia Sinica, Taiwan
by@crypto.tw

ABSTRACT

In ISSAC 2017, van der Hoeven and Larrieu showed that evaluating a polynomial $P \in \mathbb{F}_q[x]$ of degree $< n$ at all n-th roots of unity in $\mathbb{F}_{q^d}$ can essentially be computed d times faster than evaluating $Q \in \mathbb{F}_{q^d}[x]$ at all these roots, assuming $\mathbb{F}_{q^d}$ contains a primitive n-th root of unity [18]. Termed the Frobenius FFT, this discovery has a profound impact on polynomial multiplication, especially for multiplying binary polynomials, which finds ample application in coding theory and cryptography. In this paper, we show that the theory of Frobenius FFT beautifully generalizes to a class of additive FFT developed by Cantor and Gao-Mateer [5, 11]. Furthermore, we demonstrate the power of Frobenius additive FFT for $q = 2$: to multiply two binary polynomials whose product is of degree < 256, the new technique requires only 29,005 bit operations, while the best result previously reported was 33,397 [1]. To the best of our knowledge, this is the first time that FFT-based multiplication outperforms Karatsuba and the like at such a low degree in terms of bit-operation count.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; **Computations in finite fields**;

KEYWORDS

Fast Fourier Transform, additive FFT, Frobenius FFT, Frobenius additive FFT, polynomial multiplication, finite field, complexity bound, Frobenius automorphism

ACM Reference Format:
Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. 2018. Frobenius Additive Fast Fourier Transform. In *ISSAC '18: 2018 ACM International Symposium on Symbolic and Algebraic Computation, July 16–19, 2018, New York, NY, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3208976.3208998

ACM ISBN 978-1-4503-5550-6/18/07.
https://doi.org/10.1145/3208976.3208998

1 INTRODUCTION

Let $\mathbb{F}_{q^d}$ be the finite field of q^d elements, and let $\xi \in \mathbb{F}_{q^d}$ be a primitive n-th root of unity. Let $\mathbb{F}_{q^d}[x]_{<n}$ denote the set of polynomials in $\mathbb{F}_{q^d}[x]$ with degree $< n$. The (discrete) Fourier transform of a polynomial $P \in \mathbb{F}_{q^d}[x]_{<n}$ is $(P(1), P(\xi), P(\xi^2), \ldots, P(\xi^{n-1}))$, namely, evaluating P at all n-th roots of unity. How to efficiently compute the Fourier transform not only is an important problem in its own right but also finds a wide variety of applications. As a result, there is a long line of research aiming to find what is termed "fast" Fourier tranform, or FFT for short, for various situations.

Arguably, one of the most important applications of FFT over finite fields is fast polynomial multiplication. In particular, the case of $q = 2$ has received a lot of attention from the research communities due to its wide-ranging application, e.g., in coding theory and cryptography. Obviously, here we need to go to an appropriate extension field $\mathbb{F}_{2^d}$ in order to obtain a primitive n-th root of unity for any practically meaningful n. In this case, we can use the well-known Kronecker method to efficiently compute binary polynomial multiplication [12]. Such FFT-based techniques have better asymptotic complexity compared with school-book and Karatsuba algorithms. However, it is a conventional wisdom that FFT may not be suitable for polynomial multiplication of small degrees because of the large hidden constant in the big-O notation [10].

In ISSAC 2017, van der Hoeven and Larrieu showed how to use the Frobenius map $x \mapsto x^q$ to speed up the Fourier transform of $P \in \mathbb{F}_q[x]$ essentially by a factor of d over $Q \in \mathbb{F}_{q^d}[x]$ and hence avoid the factor-of-two loss as in the Kronecker method [18]. However, the Frobenius FFT is complicated, especially when the Cooley-Tukey algorithm is used for a (highly) composite n. One of the reasons behind might be that the Galois group of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ is generated by the Frobenius map and isomorphic to a cyclic subgroup of the *multiplicative* group of units of $\mathbb{Z}/n\mathbb{Z}$, whereas the Cooley-Tukey algorithms works by decomposing the *additive* group $\mathbb{Z}/n\mathbb{Z}$. The complicated interplay between these two group structures can bring a lot of headaches to implementers.

Can we hope for a better alignment between the Frobenius map and FFT-style algorithms? In his seminal work, Cantor showed how to evaluate a polynomial on some additive subgroup of size n of a tower of Artin-Schreier extensions of a finite field of characteristic p and gave an $O(n(\log n)^{1+\log_p((p+1)/2)})$ FFT-like algorithm based on polynomial division [5]. Based on Cantor's construction, Gao

and Mateer gave a Cooley-Tukey-style algorithm whose complexity is $O(n \lg(n) \lg\lg(n))$ for the specialized case of $n = 2^{2^r}$ [11], using which Chen *et al.* achieved a competitive performance among the state of the art in binary polynomial multiplication algorithms [9]. As will become clear later in this paper, the theory of Frobenius FFT beautifully generalizes to such additive FFT techniques developed by Cantor and Gao-Mateer because the group it works on comes from the same Frobenius map. In the rest of this paper, we shall refer to such a generalization as *the Frobenius additive FFT*, or FAFFT for short, and although we will restrict our discussion to the case of $p = 2$, most of our results can be extended to the case of general p.

We will use binary polynomial multiplication as a vehicle to demonstrate the power of FAFFT, for which we will use the bit-operation—mainly AND and XOR, as we are working in $\mathbb{F}_2$—count as the main measure for computational complexity. Although this may not be the most accurate performance indicator on modern CPUs, it is still a useful metric for complexity estimation, especially for hardware implementation using digital circuits or "bitsliced" software implementation widely used in embedded systems. In such a performance metric, today's most competitive techniques for multiplying binary polynomials of small degrees are almost all based on the Karatsuba algorithm or its generalization to n-way split [1, 6–8, 21]. As we will see, FAFFT techniques can break this monopoly and outperform Karatsuba-like algorithms in multiplying binary polynomials of degree as small as 231. To the best of our knowledge, this is the first time FFT-based techniques are shown to be competitive for multiplying polynomials of such small degrees.

Last but not least, we have released to the general public our software that generates computational procedures for polynomial multiplication:

https://github.com/fast-crypto-lab/Frobenius_AFFT .

The rest of this paper is organized as follows. In Section 2, we will review the theory and some important techniques for additive FFT; then in Section 3, we will develop those for FAFFT. Finally in Section 4, we will show how we can set a few new speed records, both theoretically and in practice, for binary polynomial multiplication using FAFFT.

2 REVIEW OF ADDITIVE FFT

2.1 Cantor's construction

An Artin-Schreier extension of a finite field of characteristic p is a degree-p Galois extension. Let $\mathbb{F}_p$ be the field of p elements, and $\mathbb{F}$ an Artin-Schreier extension field containing $\mathbb{F}_p$. The Artin-Schreier polynomial $\wp(x) = x^p - x$ is a linear map on the vector space $\mathbb{F}$ over $\mathbb{F}_p$. In other words, it is an additive polynomial: $\wp(x + y) = \wp(x) + \wp(y)$. Furthermore, the composition of i additive Artin-Schreier polynomials

$$s_i(x) := \underbrace{(\wp \circ \wp \circ \cdots \circ \wp)}_{i}(x) = \wp^{\circ i}(x)$$

is again a linear map on $\mathbb{F}$ over $\mathbb{F}_p$ and therefore additive. Following Cantor's seminal work [5], we define $s_0(x) := x$ and let W_i be the kernel of s_i as a linear map. Cantor showed that $\mathbb{F}_p = W_1 \subset W_2 \subset \cdots$, $\dim_{\mathbb{F}_p} W_i = i$, $|W_i| = p^i$, and W_i is a field if and only if $i = p^j$ for some positive integer j. Furthermore, $\deg s_i = p^i$, so the roots of s_i are precisely those in W_i, and hence $s_i(x) = \prod_{\omega \in W_i}(x - \omega)$ is exactly the *vanishing polynomial* for the subspace W_i. Last but not least, $s_i(x)$ is sparse, having at most $i + 1$ nonzero coefficients. More precisely, $s_i(x)$ is a linear combination of the monomials $x, x^p, \ldots, x^{p^i}$; in particular, we have the following property.

Lemma 2.1. *Given a vanishing polynomial $s_i(x)$ as defined above, $s_i(x) = x^{p^i} - x$ if and only if $i = p^j$ and $j \in \mathbb{N} \cup \{0\}$.*

These vanishing polynomials have a nice decomposition:

$$\begin{aligned} s_i(x) &= \prod_{\omega \in W_1} \Big(s_{i-1}(x) - \omega\Big) \\ &= \prod_{\omega \in W_2} \Big(s_{i-2}(x) - \omega\Big) \\ &\vdots \\ &= \prod_{\omega \in W_i} \Big(x - \omega\Big). \end{aligned}$$

Based on this decomposition, Cantor gave an FFT-like algorithm for evaluating $P \in \mathbb{F}[x]$ with $\deg P < n = p^m$ at all $\omega \in W_m$ by iteratively computing the following set of polynomials:

$$\mathcal{A}_m = \left\{ P^{(i)}_\omega(x) := P(x) \bmod \Big(s_i(x) - \omega\Big) : 0 \le i \le m, \omega \in W_{m-i} \right\}. \tag{1}$$

That is, we start from $P^{(m)}_0(x) = P(x)$ and compute $P^{(m-1)}, \ldots$; the constant polynomials $P^{(0)}_\omega \; \forall \omega \in W_m$ are the evaluation results. We note that for any polynomial $P \in \mathbb{F}[x]$, P divides $s(P) = P^p - P$. In particular, for $\omega \in W_{m-i}$, $s_i(x) - \omega$ divides $s(s_i(x) - \omega) = s_{i+1}(x) - s(\omega)$. Therefore, once we have computed $P^{(i+1)}_{s(\omega)}$, we can compute

$$\begin{aligned} P^{(i)}_\omega(x) &= P(x) \bmod \Big(s_i(x) - \omega\Big) \\ &= \Big(P(x) \bmod \big(s_{i+1}(x) - s(\omega)\big)\Big) \bmod \Big(s_i(x) - \omega\Big) \\ &= P^{(i+1)}_{s(\omega)}(x) \bmod \Big(s_i(x) - \omega\Big). \end{aligned}$$

In this paper, we define the additive fourier transform of P as

$$AFT_n(P) = \left\{ P(x) \bmod \Big(s_0(x) - \omega\Big) : \omega \in W_m \right\} = \{P(\omega) : \omega \in W_m\}$$

2.2 Cantor bases

The complexity of Cantor's algorithm is $O(n(\log n)^{1+\log_p((p+1)/2)})$ for general p and n a power of p. For the case of $p = 2$ and $n = 2^{2^r}$ a power of a power of two, Gao and Mateer showed how to reduce the complexity to $O(n \lg(n) \lg\lg(n))$ via a Cooley-Tukey-style algorithm [11]. From now on, we will restrict our discussion to the special case of $p = 2$; nevertheless, we would like to stress that the theory of Frobenius FFT applies to the general additive FFT techniques developed by Cantor. Before we present more details, we need to introduce a sequence of explicit and computationally useful bases for extension fields given by Cantor [5].

Let $\mathbb{F}_{2^d}$ denote an binary extension field, and let

$$\boldsymbol{v} = (v_0, v_1, \ldots, v_{d-1}).$$

We call $\boldsymbol{v}$ a basis for $\mathbb{F}_{2^d}$ if $v_0, v_1, \ldots, v_{d-1}$ are linearly independent over $\mathbb{F}_2$. Throughout this paper, we often represent an element ω_i

of a binary extension field as

$$\omega_i = i_0 v_0 + i_1 v_1 + \cdots + i_{d-1} v_{d-1},$$

where $i = i_0 + 2i_1 + 2^2 i_2 + \cdots + 2^{d-1} i_{d-1}$, $i_j \in \{0, 1\}$ $\forall 0 \le j < d$, with the basis elements $v_0, v_1, \ldots, v_{d-1}$ inferred from the context.

Definition 2.2. Given a sequence $u_0, u_1, u_2, \ldots$ of elements from the algebraic closure of $\mathbb{F}_2$ satisfying

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

where each "monomial of a lower degree" has the form $u_0^{j_0} u_1^{j_1} \cdots u_{i-1}^{j_{i-1}}$ such that $\forall 0 \le k < i, j_k \in \{0, 1\}$ and $\exists k, j_k = 0$. For $d = 2^r$, a Cantor basis $\boldsymbol{v}_d = (v_0, \ldots, v_{d-1})$ for $\mathbb{F}_{2^d}$ is given by

$$v_i = u_0^{i_0} u_1^{i_1} \cdots u_{r-1}^{i_{r-1}},$$

where $i = i_0 + 2i_1 + \cdots + 2^{r-1} i^{r-1}$.

If we fix $\mathbb{F}_{2^{2^k}} = \mathbb{F}_2(u_0, u_1, \ldots, u_{k-1})$ for $k = 1, 2, \ldots$, then we have a tower of Artin-Schreier extension fields containing $\mathbb{F}_2$. As a quick example, the following tower of extension fields of $\mathbb{F}_2$ are one such construction:

$$\begin{aligned}
\mathbb{F}_4 &:= \mathbb{F}_2[u_0]/(u_0^2 + u_0 + 1),\\
\mathbb{F}_{16} &:= \mathbb{F}_4[u_1]/(u_1^2 + u_1 + u_0),\\
\mathbb{F}_{256} &:= \mathbb{F}_{16}[u_2]/(u_2^2 + u_2 + u_1 u_0),\\
\mathbb{F}_{65536} &:= \mathbb{F}_{256}[u_3]/(u_3^2 + u_3 + u_2 u_1 u_0),\\
&\vdots
\end{aligned}$$

In this case, for example, the Cantor basis for $\mathbb{F}_{65536}$ is

$$\boldsymbol{v}_{16} = (1, u_0, u_1, u_0 u_1, u_2, u_0 u_2, u_1 u_2, \ldots, u_0 u_1 u_2 u_3).$$

An important property of a Cantor basis $\boldsymbol{v}_d$ is that, $\forall 0 \le i < d$, the subspace spanned by $(v_0, v_1, \ldots, v_{i-1})$ coincides with the subspace W_i on which $s_i(x)$ vanishes. That is, if we let $W_0 := \{0\}$, then we have

$$W_i = \left\{ \sum_{j=0}^{i-1} a_j v_j : a_j \in \mathbb{F}_2 \right\}.$$

We summarize some essential consequences of this property in the following lemma.

Lemma 2.3. *Given a Cantor basis $\boldsymbol{v}_d$, $\forall 0 \le j \le k \le d$, we have $v_k \in W_{k+1}$, so*

$$s_j(v_k) \in W_{k-j+1} \setminus W_{k-j}, \text{ or } s_j(v_k) + v_{k-j} \in W_{k-j}.$$

In particular, for $j = 1$, we have $s_1(v_k) + v_{k-1} \in W_{k-1}$, and for $j = k$, $s_k(v_k) + v_0 \in W_0 = \{0\}$, or $s_k(v_k) = 1$.

Before leaving our discussion on finite field arithmetic, let us remark briefly on its computational complexity. Unless stated otherwise, we will use the bit complexity model in this paper. We use $A(d)$ to denote the complexity of adding two elements in $\mathbb{F}_{2^d}$; as usual, $A(d) = O(d)$. Now let $M_q(d)$ denote the complexity of multiplying two polynomials of degree $< d$ over $\mathbb{F}_q$. Currently, the best known bound for M_q is $M_q(d) = O(d \log q \log(d \log q) 8^{\log^*(d \log q)})$, where $\log^*(\cdot)$ is the iterated logarithm function [13]. It is conventional to assume that $M_q(d)/d$ is an increasing function of d. We will denote as $M(d)$ the bit complexity of multiplying two elements in $\mathbb{F}_{2^d}$ represented in a Cantor basis. We can use the modular decomposition technique to convert $\mathbb{F}_{2^d}$ to $\mathbb{F}_2[x]$ [20], so it follows that $M(d) = O(M_2(d))$. As a result, we also assume that $M(d)/d$ is an increasing function in d. Finally, we note that in some cases, Cantor's construction allows for more efficient multiplication. For example, given $\alpha, \beta \in \mathbb{F}_{2^{2^k}} := \mathbb{F}_{2^{2^{k-1}}}[u_{k-1}]/(u_{k-1}^2 + u_{k-1} + \zeta)$, if α happens to be in the (proper) subfield $\mathbb{F}_{2^{2^{k-1}}}$, then multiplying α and β can be computed using only two multiplications in $\mathbb{F}_{2^{2^{k-1}}}$. In this case, the cost of multiplication becomes $2M(2^{k-1})$ rather than $M(2^k)$. As we shall see, we often multiply elements from different extension fields of $\mathbb{F}_2$, so Cantor's trick plays an important role in reducing bit complexity.

2.3 Additive FFT on subgroups of $\mathbb{F}_{2^{2^r}}$

In this section, we come back to the Cooley-Tukey-style algorithm given by Gao and Mateer [11]. Here we will follow the exposition due to Lin, Chung, and Han [15]. In computing the polynomials in $\mathcal{A}_m$ in Eq. (1), one needs to compute remainders of division by polynomials of the form $s_i(x) - \omega$. Thus we can replace division by substitution if we write the dividend polynomials as linear combinations of monomials of the form $\prod s_i^{b_i}$ [15]:

Definition 2.4. Given a (Cantor) basis $\boldsymbol{v}_d$ and its subspace vanishing polynomials $s_0, s_1, \ldots, s_{d-1}$, we define its corresponding *novel polynomial basis* as the polynomials X_k $\forall 0 \le k < n = 2^d$, where

$$X_k(x) := \prod (s_i(x))^{b_i} \text{ for } k = \sum_{i=0}^{d-1} 2^i b_i \text{ with } b_i \in \{0, 1\}.$$

It is easy to see that, as $\deg s_i = 2^i$, $\deg X_k = k$. Thus, a polynomial $P \in \mathbb{F}_{2^d}[x]$ with $\deg P < n$ can be represented in novel polynomial basis:

$$P(x) = p_0 X_0(x) + p_1 X_1(x) + \cdots + p_{n-1} X_{n-1}(x) \text{ for } p_i \in \mathbb{F}_{2^d}.$$

We note that the notion of a novel polynomial basis can be defined for d not a power of two. However, there is no gain in this case in terms of complexity because conversion from the usual monomial basis to novel polynomial basis has the same complexity $O(n \lg n (\lg n)^2)$ as Cantor's iterative division algorithm. When d is a power of two, on the other hand, both Gao-Mateer and Lin *et al.* gave $O(n \lg n \lg \lg n)$ algorithms for basis conversion; in Algorithm 1, we present the version given by the latter [16]. We also note that for a polynomial that admits coefficients from $\mathbb{F}_2$, we can easily gain a factor of d because addition in $\mathbb{F}_2$ costs $A(1)$ rather than $A(d)$.

Finally, we are ready to define the additive FFT of a polynomial $P \in \mathbb{F}_{2^d}[x]$ with $\deg P < 2^k$ as

$$\texttt{AFFT}(k, P, \alpha) := \Big(P(\omega_i + \alpha)\Big)_{i=0}^{2^k - 1}.$$

Now if $n_1 = 2^{k-1}$, then

$$\begin{aligned}
&P(\omega_i + \alpha) = P(\omega_{n_1 \cdot i_1 + i_2} + \alpha)\\
&= \sum_{0 \le j_2 < n_1} \sum_{0 \le j_1 < 2} p_{n_1 \cdot j_1 + j_2} X_{n_1 \cdot j_1 + j_2}(\omega_{n_1 \cdot i_1 + i_2} + \alpha)\\
&= \sum_{0 \le j_2 < n_1} \Big(p_{j_2} + s_{k-1}(\omega_{n_1 \cdot i_1 + i_2} + \alpha) \cdot p_{n_1 + j_2}\Big) X_{j_2}(\omega_{n_1 \cdot i_1 + i_2} + \alpha)\\
&= \sum_{0 \le j_2 < n_1} \Big(p_{j_2} + s_{k-1}(\omega_{n_1 \cdot i_1} + \alpha) \cdot p_{n_1 + j_2}\Big) X_{j_2}(\omega_{i_2} + (\alpha + \omega_{n_1 \cdot i_1})).
\end{aligned}$$

BasisConversion($f(x)$) :
input : $f(x) = f_0 + f_1 x + ... + f_{n-1}x^{n-1}$
output: $g(X) = g_0 + g_1 X_1(x) + ... + g_{n-1}X_{n-1}(x) = f(x)$
if $\deg f(x) \le 1$ **then** return $g(X) = f_0 + X_1 f_1$;
Let $k = \max\{2^i : \deg s_{2^i}(x) \le \deg f(x)\}$.
Compute $h'(y) = h'_0(x) + h'_1(x)y + h'_2(x)y^2 + \cdots$ such that $h'(s_k(x)) = f(x)$ and all $h'_i(x)$ has degree $< 2^k$.
$h(Y) \leftarrow$ BasisConversion($h'(y)$)
// we have $h'(y) = h(Y) = h_0(x) + h_1(x)Y_1 + h_2(x)Y_2 + \cdots$
// $h'(s_k(x)) = h(X) = h_0(x) + h_1(x)X_{2^k} + h_2(x)X_{2^{k+1}} + \cdots$
$g_i(X) \leftarrow$ BasisConversion($h_i(x)$) for all $h_i(x)$.
return $g_0(X) + g_1(X)X_{2^k} + g_2(X)X_{2^{k+1}} + \cdots$

Algorithm 1: Converting from monomial to novel polynomial basis

We can see that the AFFT with input polynomial degree of $2^k - 1$ can be computed using two AFFT with input polynomial of degree $2^{k-1} - 1$ corresponding to $i_1 = 0$ and 1, which leads us to Algorithm 2.

AFFT($k, P(x), \alpha$) :
input : $P(x) = p_0X_0(x) + p_1X_1(x) + ... + p_{2^k-1}X_{2^k-1}(x)$, all $p_i \in \mathbb{F}_{2^d}$
$\alpha \in \mathbb{F}_{2^d}, k \le d$
output: $(P(\omega_0 + \alpha), P(\omega_1 + \alpha), \ldots, P(\omega_{2^k-1} + \alpha))$.
if $k = 0$ **then** return p_0 ;
// Decompose $P(x) = P_0(x) + s_{k-1}(x) \cdot P_1(x)$.
$P_0(x) \leftarrow p_0X_0(x) + p_1X_1(x) + \ldots p_{2^{k-1}-1}X_{2^{k-1}-1}(x)$
$P_1(x) \leftarrow p_{2^{k-1}}X_0(x) + p_{2^{k-1}+1}X_1(x) + \ldots p_{2^k-1}X_{2^{k-1}-1}(x)$
$Q_0(x) \leftarrow P_0(x) + s_{k-1}(\alpha) \cdot P_1(x)$.
$Q_1(x) \leftarrow Q_0(x) + s_{k-1}(v_{k-1}) \cdot P_1(x)$.
return AFFT($k-1, Q_0(x), \alpha$)‖AFFT($k-1, Q_1(x), v_{k-1} + \alpha$)

Algorithm 2: Addtive FFT in novel polynomial basis [15]

We note that for a Cantor basis, $s_{k-1}(\omega_{n_1}) = s_{k-1}(v_{k-1}) = 1$ from Lemma 2.3. Given $P \in \mathbb{F}_{2^d}[x]$ with $\deg P < n$ represented in monomial basis and $n = 2^m$, its additive Fourier transform $AFT_n(P)$ can be computed as follow. We first perform basis conversion to get p_i such that $P(x) = p_0X_0(x) + p_1X_1(x) + \ldots + p_{2^m-1}X_{2^m-1}(x)$. Then we perform AFFT($m, P(x), 0$). Thus, to compute AFT_n(P) using AFFT, the maximum depth of recursion is m, and the algorithm performs total $\frac{1}{2}n$ multiplications and n additions in each depth of recursion. Therefore the cost of the algorithm is $\frac{1}{2}n\lg(n)(M(d) + 2A(d))$, where $n = 2^m$ is the number of terms.

3 FROBENIUS ADDITIVE FOURIER TRANSFORM

Let P be a polynomial in $\mathbb{F}_2[x]$ and $\boldsymbol{v}_d$, a Cantor basis for $\mathbb{F}_{2^d}$. Recall that for all $\alpha \in \mathbb{F}_{2^d}$, $P(\phi(\alpha)) = \phi(P(\alpha))$, where ϕ is the Frobenius map that sends x to x^2. The core idea of the Frobenius Fourier transform is to evaluate P at a minimal number of points such that the values of P at other points can be derived through the Frobenius map ϕ when $P \in \mathbb{F}_2[x] \subset \mathbb{F}_{2^d}[x]$. This minimal set of points is called a *cross section* [18]. Formally, given a set $W \subseteq \mathbb{F}_{2^d}$, a subset $\Sigma \subseteq W$ is called a cross section of W if for every $w \in W$, there exists exactly one $\sigma \in \Sigma$ such that $\phi^{\circ j}(\sigma) = w$ for some j.

Let $\boldsymbol{v}_d$ denote a basis of $\mathbb{F}_{2^d}$. Given a polynomial $P \in \mathbb{F}_2[x]$ with $\deg P < n = 2^m$, the $AFT_n(P)$ is the evaluation of the points in $W_m = \{\omega_0, \omega_1, \omega_2, \ldots, \omega_{2^m-1}\}$. To perform a Frobenius additive Fourier transform, we partition W_m into disjoint orbits under the action of ϕ. If there exists a subset Σ of W_m that contains exactly one element in each orbit, then Σ is a cross section of W_m, and the Frobenius map allows us to recover $AFT_n(P)$ from the values of P on Σ. We denote

$$\left\{P(\sigma) : \sigma \in \Sigma\right\}$$

the *Frobenius additive Fourier transform* (*FAFT*) of polynomial P. In the rest of this section, we will show how to generalize the theory of Frobenius FFT to additive FFT by explicitly constructing the cross sections.

3.1 Frobenius maps and Cantor bases

We recall that the Frobenius map ϕ on $\mathbb{F}_{2^d}$ generates the (cyclic) Galois group $\mathrm{Gal}(\mathbb{F}_{2^d}/\mathbb{F}_2)$ of order $[\mathbb{F}_{2^d} : \mathbb{F}_2] = d$, which naturally acts on $\mathbb{F}_{2^d}$ by taking $\alpha \in \mathbb{F}_{2^d}$ to $\phi(\alpha)$. The orbit of α under this action is thus

$$\mathrm{Orb}_\alpha = \left\{\sigma(\alpha) : \sigma \in \mathrm{Gal}(\mathbb{F}_{2^d}/\mathbb{F}_2)\right\}.$$

Lemma 3.1. *Given a Cantor basis $\boldsymbol{v}_d$, $\forall k > 0$, $\forall w \in W_{k+1} \setminus W_k$,*

$$\left|\mathrm{Orb}_w\right| = 2^{\lfloor \lg k \rfloor + 1}.$$

Proof. Let $\ell = \lfloor \lg k \rfloor$. In this case, $2^\ell \le k < 2^{\ell+1}$, and $v_k = u_\ell u_{\ell-1}^{j_{\ell-1}} \cdots u_0^{j_0}$, $j_i \in \{0, 1\} \forall 0 \le i < \ell$. Since $w \in W_{k+1} \setminus W_k$, we can write

$$w = v_k + \alpha = u_\ell u_{\ell-1}^{j_{\ell-1}} \cdots u_0^{j_0} + \alpha$$

for some $\alpha \in W_k$. Obviously the smallest field containing w is $\mathbb{F}_{2^{2^{\ell+1}}} = \mathbb{F}_2(u_0, u_1, \ldots, u_\ell)$, so the stabilizer of w is the subgroup of $\mathrm{Gal}(\mathbb{F}_{2^d}/\mathbb{F}_2)$ generated by $\phi^{2^{\ell+1}}$. It follows immediately from the orbit-stabilizer theorem and Lagrange's theorem that

$$\left|\mathrm{Orb}_w\right| = 2^{\lfloor \lg k \rfloor + 1}.$$

□

Consider the field $\mathbb{F}_{2^d}$ with a Cantor basis $\boldsymbol{v}_d$ for d a power of two. From Lemma 2.3, we have $\phi(v_0) = v_0$, and $\phi(v_i) = s(v_i) + v_i = v_i + v_{i-1} + \alpha$, where $\alpha \in W_{i-1}$ for $i > 0$. Based on this, we can further characterize the orbit of $w \in W_{k+1} \setminus W_k$ using the following lemma.

Lemma 3.2. *Given a Cantor basis $\boldsymbol{v}_d$, $\forall k > 0$, consider the orbit of $w \in W_{k+1} \setminus W_k$ under the action of* $\mathrm{Gal}(\mathbb{F}_{2^d}/\mathbb{F}_2)$. *For each $i = 1, 2, 4, \ldots$, let j_i be in $\{0, 1\}$, there is precisely one element $w' \in \mathrm{Orb}_w$ such that $w' = v_k + j'_1 v_{k-1} + \cdots + j'_k v_0 \in W_{k+1} \setminus W_k$, $\forall j'_i \in \{0, 1\}$, and $j'_i = j_i$ for $i = 1, 2, 4, \ldots, 2^{\lfloor \lg k \rfloor}$.*

Proof. Let ℓ be a power of two. From Lemma 2.1, we have $\phi^{\circ\ell}(x) = x^{2^\ell} = s_\ell(x) + x$. From Lemma 2.3, we see that $\phi^{\circ\ell}(w) + w = s_\ell(w) \in W_{k-\ell+1} \setminus W_{k-\ell}$. That is, $\phi^{\circ\ell}(w) + w \in v_{k-\ell} + W_{k-\ell}$. In other words, $\phi^{\circ\ell}$ allows us to flip j_ℓ, while $j_1, j_2, \ldots, j_{\ell-1}$ remain the same. E.g., for $\ell = 1$, we see that one of w and $\phi(w)$ has $j'_1 = 0$ and the other has $j'_1 = 1$ for any $w \in W_{k+1} \setminus W_k$. Similarly for $\ell = 2$, one of w and $\phi^{\circ 2}(w)$ has $j'_2 = 0$ and the other has $j'_2 = 1$, while

both w and $\phi^{\circ 2}(w)$ have the same j_1'. The same argument holds for $\phi(w)$ and $\phi^{\circ 2}(\phi(w)) = \phi^{\circ 3}(w)$. Thus $w, \phi(w), \phi^{\circ 2}(w), \phi^{\circ 3}(w)$ cover all 4 combinations of j_1 and j_2. Continuing, we see that w and $\phi^{\circ 4}(w)$ have different j_4' but the same j_1' and j_2'. Proceeding thusly, $w, \phi(w), \ldots, \phi^{\circ 2^{\lfloor \lg k \rfloor + 1}-1}(w)$ cover all combinations of $j_i' \in \{0,1\}$, for $i = 1, 2, 4, \ldots, 2^{\lfloor \lg k \rfloor}$. Finally, as $|\mathrm{Orb}_w| = 2^{\lfloor \lg k \rfloor + 1}$, we see that for each such combination, there is precisely one corresponding element in Orb_w combination due to the pigeonhole principle. □

Now we can explicitly construct a cross section. Let $\Sigma_0 = \{0\}$, and $\forall k > 0$, let

$$\Sigma_k = \left\{ v_{k-1} + j_1 v_{k-2} + \cdots + j_{k-1} v_0 : \begin{matrix} j_i = 0 \text{ if } i \text{ is a power of 2,} \\ j_i \in \{0,1\} \text{ otherwise.} \end{matrix} \right\}$$

Theorem 3.3. *Σ_k is a cross section of $W_k \setminus W_{k-1}$. That is, $\forall k > 0$, $\forall w \in W_k \setminus W_{k-1}$, there exists exactly one $\sigma \in \Sigma_k$ such that $\phi^{\circ j}(\sigma) = w$ for some j.*

Proof. First, any two elements of Σ_k are in different orbits for any k; this is a corollary of Lemma 3.2. Next, we know that $\forall w \in W_k \setminus W_{k-1}$, $\left|\mathrm{Orb}_w\right| = 2^{\lfloor \lg (k-1) \rfloor + 1}$, and $\phi^{\circ j}(w) \in W_k \setminus W_{k-1}$, $\forall j$. So each orbit generated by the element in Σ_k has the size $2^{\lfloor \lg (k-1) \rfloor + 1}$, and $2^{\lfloor \lg (k-1) \rfloor + 1} \cdot |\Sigma_k| = 2^{k-1} = |W_k \setminus W_{k-1}|$. By the pigeonhole principle, each element in $W_k \setminus W_{k-1}$ must be in an orbit generated by exactly one element in Σ_k. □

3.2 Frobenius additive Fast Fourier transform

By Theorem 3.3, a cross section of W_m is

$$\Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_m \ .$$

Given $P(x) \in \mathbb{F}_2[x]$ with $\deg P < n$ represented with novel polynomial basis and Cantor basis $\boldsymbol{v}_d$ of field $\mathbb{F}_{2^d}$ where $n = 2^m$, instead of computing $AFT_n(P) = (P(\omega_0), P(\omega_1), \ldots, P(\omega_{2^m-1}))$, we only need to compute $FAFT_n(P) = \{P(\sigma) : \sigma \in \Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_m\}$ and then use Frobenius map ϕ to get the rest.

Due to the structure of the additive FFT, we can simply "truncate" to those points. In the original additive FFT (Algorithm 2), each FAFFT calls two FAFFT routines recursively. Those two recursive calls correspond to evaluating points in $\alpha + W_{k-1}$ and $\alpha + v_{k-1} + W_{k-1}$. We can omit one of the two calls and only compute on $\alpha + W_{k-1}$, so we will not evaluate the points not in the cross section Σ, as $\Sigma \cap (\alpha + v_{k-1} + W_{k-1}) = \emptyset$. This is how we arrive at Algorithm 3, in which we define $\rho(l)$:

$$\rho(l) = \begin{cases} 1 & \text{if } l = 0, \\ 2^{\lceil \lg l \rceil} & \text{otherwise.} \end{cases}$$

It is easy to see that $\mathtt{FAFFT}(m, 1, P(x), v_m)$ computes $\{P(x) : x \in \Sigma_m\}$ because truncation happens when the v_{m-l-1} component is zero for all points in Σ_m, i.e., l is a power of two. To compute $FAFT(\mathrm{P})$, we call $\mathtt{FAFFT}(m, 0, P(x), 0)$.

Figure 1 is a graphical illustration of the $\mathtt{FAFFT}(5, 0, f, 0)$ routine, which computes $FAFT_{32}(f)$ for $f = g_0 X_0(x) + g_1 X_1(x) + g_2 X_2(x) + \ldots + g_{31} X_{31}(x)$. It consists of five layers, each corresponding to one level of recursion in the pseudocode. Each grey box is a "butterfly unit" that performs a multiplication and an addition. A butterfly unit has two inputs $a, b \in \mathbb{F}_{2^d}$. For normal butterfly unit with two output a', b', it performs

$$a' \leftarrow a + b \cdot s_k(\alpha),$$
$$b' \leftarrow a' + b,$$

while the truncated one only outputs a'. In the figure, we denote the $s_k(\alpha)$ in each butterfly unit $c_{i,j}$. Butterflies are also labelled with the value l coressponding to each recursive calls FAFFT. We can see that truncated butterflies happen when l is a power of 2. Initially, the inputs to butterfly units, $g_0, g_1, \ldots, g_{31}$, are all in $\mathbb{F}_2$. But as it goes through the layers, the bit size of the input to the following butterfly unit grows larger, as the multiplicands $c_{i,j}$ may be in extension fields. For example, after the second layer, the lower half of the input are in $\mathbb{F}_{2^2}$ because $c_{3,1}$ are in $(W_2 \setminus W_1) \subset \mathbb{F}_{2^2}$. Then they go through butterfly unit with $c_{2,2} \in (W_3 \setminus W_2) \subset \mathbb{F}_{2^4}$ and finally arrive in $\mathbb{F}_{2^4}$.

```
FAFFT(k, l, P(x), α) :
input  : k ∈ ℕ, l ∈ ℕ,
         P(x) = p_0 X_0(x) + p_1 X_1(x) + ... + p_{2^k-1} X_{2^k-1}(x)
           where p_i ∈ F_{2^ρ(l)},
         α ∈ W_{k+l} \ W_k if l > 0 , otherwise α = 0.
output : P(σ)_{σ∈Σ} where Σ = (Σ_0 ∪ Σ_1 ∪ ... ∪ Σ_{k+l}) ∩ (α + W_k),
if k = 0 then return p_0 ;
Decompose P(x) = P_0(x) + s_{k-1}(x) · P_1(x).
Q_0(x) ← P_0(x) + s_{k-1}(α) · P_1(x).
Q_1(x) ← Q_0(x) + P_1(x).
if l = 0 then
    return FAFFT(k − 1, 0, Q_0(x), α) ||
           FAFFT(k − 1, 1, Q_1(x), v_{k-1} + α)
else if l is a power of two then
    return FAFFT(k − 1, l + 1, Q_0(x), α)
else
    return FAFFT(k − 1, l + 1, Q_0(x), α) ||
           FAFFT(k − 1, l + 1, Q_1(x), v_{k-1} + α)
end
```

Algorithm 3: Frobenius Additive FFT in novel polynomial basis.

3.3 Complexity Analysis

In this section, we analyze the complexity of FAFFT in Algorithm 3. Let $F(k,l)$ and $F_A(k,l)$ denote the cost of multiplication and addition to compute $\mathtt{FAFFT}(k, l, P(x), \alpha)$ for $P \in \mathbb{F}_2[x]$ with $\deg P < 2^k$ and $\alpha \in W_{k+l} \setminus W_k$.

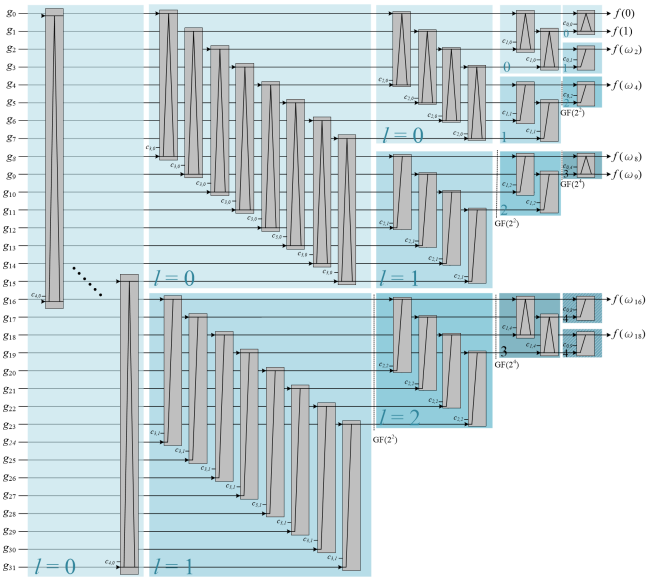
First, it is straightforward to verify that for all $\mathtt{FAFFT}(k', l', P'(x), \alpha')$ calls during the recursion:

- $\alpha' \in W_{k'+l'} \setminus W_{k'}$ if $l' > 0$, otherwise $\alpha' = 0$;
- $P'(x) = \sum p_i' X_i(x)$, $p_i' \in \mathbb{F}_{2^{\rho(l')}}$;
- $s_{k-1}(\alpha') \in (W_{l'+1} \setminus W_{l'}) \quad \subset \begin{cases} u_{\lg l'} + \mathbb{F}_{2^{l'}} & \text{if } l' \text{ is a power of two,} \\ \mathbb{F}_{2^{2^{\lceil \lg l' \rceil}}} & \text{otherwise.} \end{cases}$

Then we have $F(k,l)$

$$\leq \begin{cases} F(k-1,l) + F(k-1,l+1) + 2^{k-1}(M(1)) & \text{if } l = 0, \\ F(k-1,l+1) + 2^{k-1}(M(l)) & \text{if } l \text{ is a power of two,} \\ 2 \cdot F(k-1,l+1) + 2^{k-1}(M(2^{\lceil \lg l \rceil})) & \text{otherwise.} \end{cases}$$

Figure 1: Illustration of the butterfly network with $n = 32$. $f(0), f(1) \in \mathbb{F}_2, f(\omega_2) \in \mathbb{F}_{2^2}, f(\omega_4), f(\omega_8), f(\omega_9) \in \mathbb{F}_{2^4}$ and $f(\omega_{16}), f(\omega_{18}) \in \mathbb{F}_{2^8}$



Theorem 3.4. (multiplication complexity) *Given $n = 2^m$, for $m + l \le d$, d a power of two, we have*

$$F(m,l) \le \frac{1}{2} n \lg n \frac{M(d)}{d} \rho(l),$$

assuming that $\frac{M(l)}{l}$ is increasing in l.

Proof. We prove by induction. Consider $m = 1$: $F(1,l) = M(l) \le \frac{M(d)}{d} l$. Assume the statement holds for $m = k-1$ and for any $l \le d - m$,

$$F(m,l) \le \frac{1}{2} 2^m m \frac{M(d)}{d} \rho(l).$$

We then check three cases: first, $m = k$ and $l = 0$:

$$\begin{aligned} F(k,l) &\le F(k-1,0) + F(k-1,1) + 2^{k-1} \cdot M(1) \\ &= \frac{1}{2}(k-1)2^k \frac{M(d)}{d} + 2^k \cdot M(1) \\ &\le \frac{1}{2} k 2^k \frac{M(d)}{d}. \end{aligned}$$

Second, $m = k$ and l is a power of two:

$$\begin{aligned} F(k,l) &\le F(k-1,l+1) + \cdot 2^{k-1} \cdot M(l) \\ &= (k-1)2^k \frac{M(d)}{d} l + 2^{k-1} \cdot M(l) \\ &\le \frac{1}{2}(k-1)2^k \frac{M(d)}{d} l + 2^{k-1} \frac{M(d)}{d} l \\ &= \frac{1}{2} k 2^k \frac{M(d)}{d} l. \end{aligned}$$

Finally, $l > 0$ and is not a power of two:

$$\begin{aligned} F(k,l) &\le 2 \cdot F(k-1,l+1) + 2^{k-1} \cdot M(2^{\lceil \lg l \rceil}) \\ &= \frac{1}{2}(k-1)2^k \frac{M(d)}{d} 2^{\lceil \lg l+1 \rceil} + 2^{k-1} \cdot M(2^{\lceil \lg l \rceil}) \\ &\le \frac{1}{2}(k-1)2^k \frac{M(d)}{d} 2^{\lceil \lg l \rceil} + 2^{k-1} \cdot \frac{M(2^{\lceil \lg l \rceil})}{2^{\lceil \lg l \rceil}} 2^{\lceil \lg l \rceil} \\ &\le \frac{1}{2} k 2^k \frac{M(d)}{d} l. \end{aligned}$$

□

For the cost of addition, it can be proven following the same procedure as above, substituting $M(d)$ with $2A(d)$ and noting that $\frac{A(d)}{d}$ is constant.

Theorem 3.5. (addition complexity) *Given $n = 2^m$, for $m + l \le d$, d a power of two, we have*

$$F_A(m,l) \le n \lg n \frac{A(d)}{d} \rho(l).$$

Given $P \in \mathbb{F}_2[x]$ with $\deg P < n$, a power of two, to compute $FAFT_n(P)$, we call $\texttt{FAFFT}(\lg(n), 0, P, 0)$. Thus, the cost to compute $FAFT_n(P)$ is $\frac{1}{2} n \lg(n) \frac{M(d)}{d} + n \lg(n) \frac{A(d)}{d}$. Comparing with the cost of additive FFT $\frac{1}{2}(n \lg(n)(M(d) + 2A(d))$, we indeed gain a speed-up factor of d.

3.4 Inverse Frobenius additive FFT

The inverse Frobenius additive FFT is straightforward because for the butterfly unit with two output, it is easy to find its inverse. However, due to the truncation, it is not obvious how to perform inversion when l is a power of two. Here we show that it is always invertible. In the Algorithm 3, when l is a power of two, it truncates and only computes *FAFT* of $Q_0(x) = P_0(x) + s_{k-1}(\alpha) \cdot P_1(x)$. To be able to invert, we need to recover $P_0(x)$ and $P_1(x)$ from $Q_0(x)$. Note that $s_{k-1}(\alpha) \in (W_{l+1} \setminus W_l) = v_l + W_l$ because $\alpha \in W_{k+l+1} \setminus W_{k+l}$ and Lemma 2.3. Since we use a Cantor basis, c.f. Definition 2.2, $v_l = u_{\lg l}$ when l is a power of two. We can rewrite the equation from the point of $\mathbb{F}_{2^l}[u_{\lg l}][x]$. Let $s_{k-1}(\alpha) = u_{\lg l} + c$ and $c \in \mathbb{F}_{2^l}$,

$$Q_0(x) = R_0(x) + R_1(x) u_{\lg l} = P_0(x) + (c + u_{\lg l}) \cdot P_1(x),$$

where $R_0(x), R_1(x) \in \mathbb{F}_{2^l}[x]$. Then we have

$$\begin{aligned} P_0(x) &= R_0(x) + R_1(x) \cdot c, \\ P_1(x) &= R_1(x). \end{aligned}$$

Thus we can always recover $P_0(x)$ and $P_1(x)$ from $Q(x)$. The full inverse Frobenius additive FFT algorithm is shown in Algorithm 4. It can be shown that the complexity of the inverse FAFFT is also bounded by $\frac{1}{2} n \lg(n) \frac{M(d)+2A(d)}{d}$ following the analysis in the previous section.

4 APPLICATION TO $\mathbb{F}_2[x]$-MULTIPLICATION

In this section, we will show that FAFFT is competitive both theoretically (measured by the number of bit operations involved) and in practice (measured by actual speeds on modern CPUs) when applied to multiplying two polynomials in $\mathbb{F}_2[x]$.

IFAFFT(k, l, A, α) :
input : $A = P(\sigma)_{\sigma \in \Sigma}$
where $\Sigma = (\Sigma_0 \cup \Sigma_1 \cup \ldots \cup \Sigma_{k+l}) \cap (\alpha + W_k)$,
$\alpha \in W_{k+l} \setminus W_k$ if $l > 0$, otherwise $\alpha = 0$.
output : $P(x) = p_0 X_0(x) + p_1 X_1(x) + \ldots + p_{2^k-1} X_{2^k-1}(x)$
where $p_i \in \mathbb{F}_{2^{\rho(l)}}$.

if $k = 0$ **then return** the only element in A ;
if $l = 0$ **then**
  Divide the set A to A_0, A_1
  $Q_0(x) \leftarrow$ IFAFFT$(k-1, 0, A_0, \alpha)$
  $Q_1(x) \leftarrow$ IFAFFT$(k-1, 1, A_1, v_{k-1} + \alpha)$
  $P_1(x) \leftarrow (Q_0(x) + Q_1(x))$
  $P_0(x) \leftarrow Q_0(x) + s_{k-1}(\alpha) \cdot P_1(x)$
else if l *is a power of 2* **then**
  $Q(x) \leftarrow$IFAFFT$(k-1, l+1, A, \alpha)$
  Let $s_{k-1}(\alpha) = c + u_{\lg(l)}$
  Let $Q(x) = R_0(x) + u_{\lg(l)} \cdot R_1(x)$
  $P_0(x) \leftarrow R_0(x) + R_1(x) \cdot c$
  $P_1(x) \leftarrow R_1(x)$
else
  Divide the set A to A_0, A_1
  $Q_0(x) \leftarrow$ IFAFFT$(k-1, l+1, A_0, \alpha)$
  $Q_1(x) \leftarrow$ IFAFFT$(k-1, l+1, A_1, v_{k-1} + \alpha)$
  $P_1(x) \leftarrow Q_0(x) + Q_1(x)$
  $P_0(x) \leftarrow Q_0(x) + s_{k-1}(\alpha) \cdot P_1(x)$
end
return $P_0(x) + P_1(x) \cdot s_{k-1}(x)$

Algorithm 4: Inverse FAFFT in novel polynomial basis

4.1 New speed records in terms of bit-operation count

One of our motivating applications is to multiply binary polynomials, which finds ample application in, e.g., implementation of elliptic-curve cryptography (ECC). For example, Bernstein pointed out that ECC over a binary field can be faster than ECC over a prime field at the same security level [1]. Thus, multiplications of binary polynomials of some specific sizes are of interest to the cryptographic engineering community [1, 6]. The previous speed records in terms of bit-operation count for multiplying two binary polynomials of small degrees, say, < 1000, were set by Bernstein [1] and Cenk-Hasan [6], both of which are based on Karatsuba-like algorithms.

With FAFFT and its inverse, we can now efficiently multiply two polynomials $P, Q \in \mathbb{F}_2[x]$ without Kronecker segmentation. First, we apply forward FAFFT on P and Q; we then pointwise-multiply the results and apply an inverse FAFFT to get the product PQ. To further reduce the bit-operation count, we also eliminate some common subexpressions [17].

Figure 2 shows the comparison against two best results previously known in the literature: Bernstein set a comprehensive set of speed records for polynomials of degree up to 1000 [1], whereas Cenk and Hasan selectively improved some of his results up to 4.5% [6]. Since FAFFT works with polynomials whose size is a power of two, we apply FAFFT to multiplying polynomials of size 256, 512, and 1024; such a small disadvantage apparently did not stop FAFFT from setting new speed records. Specifically, when compared

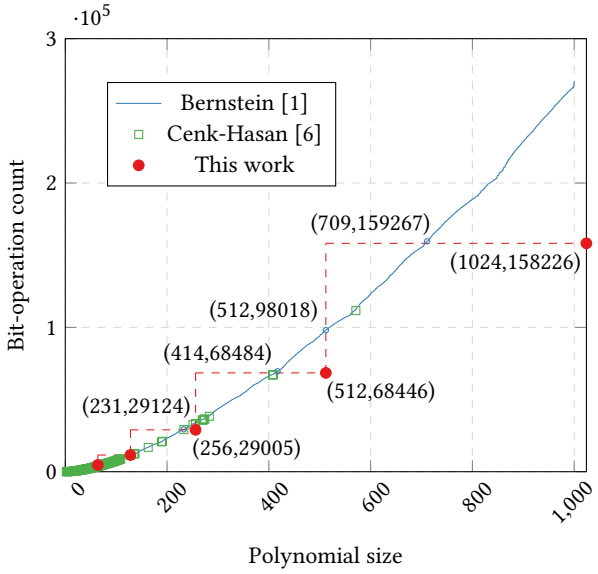


Figure 2: Complexity for multiplication in $\mathbb{F}_2[x]$

against Karatsuba-like algorithms for multiplying polynomials of size 256, 512, and 1000, FAFFT is faster by 19.1%, 29.7%, and 41.1%, respectively. To the best of our knowledge, it is the first time FFT-based method outperforms Karatsuba-like algorithm in such low degree in terms of bit-operation count.

We also try to push the limit by applying FAFFT to multiplying polynomials of size as small as 128, a degree way too small for FFT-based techniques to prevail according to the traditional wisdom. However, we are able to achieve a bit-operation count of 11,556 using FAFFT, which is only slightly worse than 11,466 achieved by Cenk and Hasan [6]. We note that this would not be possible without Frobenius FFT. Using Kronecker segmentation and a highly optimized implementation of additive FFT by [2], one would achieve a bit-operation count of 22,292. As expected, we gain a factor of two speed-up here using FAFFT.

4.2 New speed records on modern CPUs

Now we will show that FAFFT is also competitive in practice when applied to multiplying two polynomials in $\mathbb{F}_2[x]$. On modern CPUs, we can use the carryless multiplication instruction PCLMULQDQ for achieving further speed-up [14]. Similar to van der Hoeven, Larrieu, and Lecerf [19, Prop. 1], here we need a bijection for multiplying polynomials in $\mathbb{F}_2[x]_{<2^{m-1}}$.

Proposition 4.1. *Let $P \in \mathbb{F}_2[x]_{<2^m}$, $7 < m \leq 71$, $57 + m < k \leq 128$ and $S = v_{k-1} + W_{m-7}$ where v_{k-1} and W_{m-7} are defined as in Section 2 w.r.t. $\boldsymbol{v}_{128}$, a Cantor basis for $\mathbb{F}_{2^{128}}$. Then the map $P \in \mathbb{F}_2[x]_{<2^m} \mapsto (P(w))_{w \in S} \in \mathbb{F}_{2^{128}}^{2^{m-7}}$ is a bijection.*

Proof. $\forall w \in S$, $\left|\mathrm{Orb}_w\right| = 128$ (Lemma 3.1). Every two points in S are in different orbits (Lemma 3.2). Let $\mathrm{Orb}_S := \cup_{w \in S} \mathrm{Orb}_w$. Then $\left|\mathrm{Orb}_S\right| = 128 \cdot \left|S\right| = 2^m$. Since $(P(w))_{w \in S}$ determines $(P(w))_{w \in \mathrm{Orb}_S}$ via the Frobenius map, and the latter is P on 2^m points, $(P(w))_{w \in S}$

Table 1: Timing of multiplications in $\mathbb{F}_2[x]_{<n}$ on Intel Skylake Xeon E3-1275 v5 @ 3.60GHz (10^{-3} sec.)

$\log_2(n/64)$	16	17	18	19	20	21	22	23
This work, $\mathbb{F}_{2^{128}}$	9	20	41	88	192	418	889	1865
FDFT [19] [c]	11	24	56	127	239	574	958	2465
ADFT[9]	16	34	74	175	382	817	1734	3666
$\mathbb{F}_{2^{60}}$[12][b]	22	51	116	217	533	885	2286	5301
gf2x [3] [a]	23	51	111	250	507	1182	2614	6195

[a] Version 1.2. Available from http://gf2x.gforge.inria.fr/
[b] SVN r10663. Available from svn://scm.gforge.inria.fr/svn/mmx
[c] SVN r10681. Available from svn://scm.gforge.inria.fr/svn/mmx

determines P. So $P \in \mathbb{F}_2[x]_{<2^m} \mapsto (P(w))_{w\in S} \in \mathbb{F}_{2^{128}}^{2^{m-7}}$ is bijective. □

To compute $(P(w))_{w\in S}$, we perform $\mathtt{AFFT}(m, P(x), v_{k-1}) = (P(w))_{w\in v_{k-1}+W_m}$. As we only need the points $S = v_{k-1} + W_{m-7}$, we further truncate the first 7 butterfly layers of the AFFT to only evaluate points in S. The composition of these 7 layers can be written as matrix-vector products over $\mathbb{F}_2$. The multiplications in subsequent layers are all in $\mathbb{F}_{2^{128}}$. This lets us use PCLMULQDQ to multiply in $\mathbb{F}_{2^{128}}$ (analogously to [19]).

We report our benchmark on Intel Skylake architecture in Table 1. [3], [12] and [9] use Kronecker segmentation with triadic variants of Schönhage-Strassen, DFT over $\mathbb{F}_{2^{60}}$ and additive FFT over $\mathbb{F}_{2^{256}}$, respectively. Instead of Kronecker segmentation, [19] applied Frobenius FFT to improve [12] by a factor of 2 and achieved the best result. We use the variant of Frobenius additive FFT to improve the result of [9] by about a factor of 2 and outperform [19]. Thus, for polynomial of size n where $\log_2(n/64) = 16, 17, \ldots, 23$, our implementation outperforms the previous best results in the literature.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Tung Chou for his valuable suggestions to improve the article, as well as the anonymous reviewers for their insightful comments and helpful suggestions. The work is supported by the Ministry of Science and Technology, Taiwan, R.O.C. under Grant No. 105-2923-E-001-003-MY3.

REFERENCES

[1] Daniel J. Bernstein. 2009. Batch Binary Edwards. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*. 317–336. https://doi.org/10.1007/978-3-642-03356-8_19

[2] Daniel J. Bernstein and Tung Chou. 2014. Faster Binary-Field Multiplication and Faster Binary-Field MACs. In *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers (Lecture Notes in Computer Science)*, Antoine Joux and Amr M. Youssef (Eds.), Vol. 8781. Springer, 92–111. https://doi.org/10.1007/978-3-319-13051-4_6

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

[4] Michael A. Burr, Chee K. Yap, and Mohab Safey El Din (Eds.). 2017. *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017*. ACM. https://doi.org/10.1145/3087604

[5] David G. Cantor. 1989. On Arithmetical Algorithms over Finite Fields. *J. Comb. Theory Ser. A* 50, 2 (March 1989), 285–300. https://doi.org/10.1016/0097-3165(89)90020-4

[6] Murat Cenk and M. Anwar Hasan. 2015. Some new results on binary polynomial multiplication. *J. Cryptographic Engineering* 5, 4 (2015), 289–303. https://doi.org/10.1007/s13389-015-0101-6

[7] Murat Cenk, M. Anwar Hasan, and Christophe Nègre. 2014. Efficient Subquadratic Space Complexity Binary Polynomial Multipliers Based on Block Recombination. *IEEE Trans. Computers* 63, 9 (2014), 2273–2287. https://doi.org/10.1109/TC.2013.105

[8] Murat Cenk, Christophe Nègre, and M. Anwar Hasan. 2013. Improved Three-Way Split Formulas for Binary Polynomial and Toeplitz Matrix Vector Products. *IEEE Trans. Computers* 62, 7 (2013), 1345–1361. https://doi.org/10.1109/TC.2012.96

[9] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. 2017. Faster Multiplication for Long Binary Polynomials. *CoRR* abs/1708.09746 (2017). arXiv:1708.09746 http://arxiv.org/abs/1708.09746

[10] Haining Fan and M. Anwar Hasan. 2015. A Survey of Some Recent Bit-parallel GF (2 N) Multipliers. *Finite Fields Appl.* 32, C (March 2015), 5–43. https://doi.org/10.1016/j.ffa.2014.10.008

[11] Shuhong Gao and Todd Mateer. 2010. Additive Fast Fourier Transforms over Finite Fields. *IEEE Trans. Inf. Theor.* 56, 12 (Dec. 2010), 6265–6272. https://doi.org/10.1109/TIT.2010.2079016

[12] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. 2016. Fast Polynomial Multiplication over $\mathbb{F}_{2^{60}}$. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016*, Sergei A. Abramov, Eugene V. Zima, and Xiao-Shan Gao (Eds.). ACM, 255–262. https://doi.org/10.1145/2930889.2930920

[13] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. 2017. Faster Polynomial Multiplication over Finite Fields. *J. ACM* 63, 6 (2017), 52:1–52:23. https://doi.org/10.1145/3005344

[14] Intel Corp. 2008. Carry-Less Multiplication and Its Usage for Computing The GCM Mode. http://http://software.intel.com/en-us/articles/carry-less-multiplication-and-its-usage-for-computing-the-gcm-mode.

[15] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. 2014. 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*. IEEE Computer Society, 316–325. https://doi.org/10.1109/FOCS.2014.41

[16] Sian-Jheng Lin, Tareq Y. Al-Naffouri, and Yunghsiang S. Han. 2016. FFT Algorithm for Binary Extension Finite Fields and Its Application to Reed–Solomon Codes. *IEEE Trans. Inf. Theor.* 62, 10 (Oct. 2016), 5343–5358. https://doi.org/10.1109/TIT.2016.2600417

[17] C. Paar. 1997. Optimized arithmetic for Reed-Solomon encoders. In *Proceedings of IEEE International Symposium on Information Theory*. 250–. https://doi.org/10.1109/ISIT.1997.613165

[18] Joris van der Hoeven and Robin Larrieu. 2017. The Frobenius FFT, See [4], 437–444. https://doi.org/10.1145/3087604.3087633

[19] Joris van der Hoeven, Robin Larrieu, and Grégoire Lecerf. 2017. Implementing Fast Carryless Multiplication. In *Mathematical Aspects of Computer and Information Sciences - 7th International Conference, MACIS 2017, Vienna, Austria, November 15-17, 2017, Proceedings (Lecture Notes in Computer Science)*, Johannes Blömer, Ilias S. Kotsireas, Temur Kutsia, and Dimitris E. Simos (Eds.), Vol. 10693. Springer, 121–136. https://doi.org/10.1007/978-3-319-72453-9_9

[20] Joris van der Hoeven and Grégoire Lecerf. 2017. Composition Modulo Powers of Polynomials, See [4], 445–452. https://doi.org/10.1145/3087604.3087634

[21] Joachim von zur Gathen and Jamshid Shokrollahi. 2005. Efficient FPGA-Based Karatsuba Multipliers for Polynomials over F_2. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers (Lecture Notes in Computer Science)*, Bart Preneel and Stafford E. Tavares (Eds.), Vol. 3897. Springer, 359–369. https://doi.org/10.1007/11693383_25