

Functional Verification of Additive FFT Assembly Implementations in Code-Based PQC

Jiaxiang Liu¹, Kuang-Lin Pan³, Xiaomu Shi¹, Ming-Hsien Tsai², Bow-Yaw Wang³ and Bo-Yin Yang³

¹ Institute of Software, Chinese Academy of Sciences

² Department of Information Management, National Taiwan University of Science and Technology

³ Institute of Information Science, Academia Sinica

Abstract. We present the first formal verification results for the highly optimized additive Fast Fourier Transforms (FFTs) in two different code-based post-quantum cryptosystems: Classic McEliece and Hamming Quasi-Cyclic. Since Classic McEliece is already a standard and HQC is being standardized, they may both see wide adoption in the future. The Gao-Mateer and Frobenius additive FFTs during decoding are clearly the most intricate, error-prone components in the two cryptosystems respectively. They are not only complicated and mathematically intricate, but also aggressively optimized in practical implementations.

A main problem of this verification is the $> 2^{14}$ independent bit-variables in each multiplicand. Aside from adding a vector syntax and improving various syntactic transformations to the open-source tool CRYPTO LINE, we also formulate additive FFTs as Chinese Remainder Theorem decompositions which factors the verification problem into local checks feasible for computer algebra systems like SINGULAR. The workload was also tractable: the calendar time for the formal verification in this paper was less than three months for our very limited workforce.

Keywords: Classic McEliece · HQC · additive FFTs · functional verification

1 Introduction

Code-based cryptography has received a lot less attention than lattice-based cryptography, but two code-based cryptosystems are nevertheless likely to be deployed and to receive corresponding attention: *Classic McEliece* has been standardized by the International Organization for Standardization [fS26] this last June; HQC (short for *Hamming Quasi-Cyclic*) was selected by the U.S. NIST (National Institute of Standards and Technology) [oST25] in 2025 as a future standard, with draft standards to come soon.

The very different codes used — binary Goppa codes for Classic McEliece and quasi-cyclic codes for HQC — share a common computational bottleneck: decoders dominated by polynomial evaluation and interpolation over binary fields, for which the additive FFTs are the method of choice. Additive FFTs are also, by some margin, the most intricate component of high-speed constant-time implementations. Their recursion is built over a sequence of $\mathbb{F}_2$ subspaces rather than the familiar Cooley-Tukey radix-2 split; their hand-optimized, vectorized, and bitsliced schedules also depart substantially from comprehensible textbook FFTs. This combination — arithmetic complexity and aggressive optimization — makes additive FFTs both the component most likely to harbor a bug, and one whose bugs we cannot risk randomized testing to catch before deployment.

Contributions. We present the first functional formal verification of highly optimized additive FFT implementations in code-based cryptography, at the heart of both Gao–Mateer additive FFT in Classic McEliece (mceliece348864 and mceliece8192128 for AVX2) and Frobenius additive FFT in HQC-128 (forward and backward for AVX2 and NEON). These are constant-time state-of-the-art implementations as of July 2026. Concretely:

- We extend the open-source tool CRYPTO_{LINE} with vector variables and instructions, allowing bitsliced and vectorized $\mathbb{F}_2$ -arithmetic to be modeled at the granularity of actually handwritten AVX2/NEON code rather than scalarized by hand. Our extension necessarily includes significant performance improvements in various syntactic transformations in CRYPTO_{LINE} to analyze models with several orders of magnitudes more variables than those in lattice-based models.
- In our Chinese Remainder Theorem-based verification methodology, each additive FFT level is framed as a ring decomposition, factoring an otherwise intractable bit-blasted problem (tens of thousands of independent bit-variables per multiplicand) into a sequence of local checks small enough for a computer algebra system such as SINGULAR to dispatch.
- We report a complete evaluation: all radix conversion routines verify within six minutes, and full forward/backward additive FFTs on both architectures within roughly 0.6–1.4 hours each, with the entire effort completed in under 2.5 months by a small team. Property checking against an inefficient reference model is combined with logic equivalence checking against the optimized implementation. The combined verification lets us verify aggressively optimized code (table-lookup radix conversion, shuffle-based multiplication, embeddings into $\mathbb{F}_{2^{64}}$) whose structure is widely divergent from the underlying algebra.

Related Work. Numerous work about functional verification of cryptographic programs has been published (for a hopelessly incomplete list, see [BBB⁺21, BCH⁺26, CLT⁺25, ABB⁺25, BHK⁺25, AOB⁺24, EPG⁺19]). Lattice-based post-quantum cryptographic programs have also been verified. Various optimized assembly implementations for Number Theoretic Transforms in different lattice-based schemes have been verified in [HLS⁺22, CYW25, CLT⁺25]. Formal verification of a low-level implementation for ML-KEM is reported in [AOB⁺24]. Classical elliptic curve and lattice-based cryptography adopt prime fields in their construction. Almost all prior work hence does not verify computation over finite fields of characteristic 2. In [BCPR23], the reference C implementation of Classic McEliece 348864 is partially verified. The reference implementation is inefficient and not meant for deployment. It hence does not employ additive FFTs. To our knowledge, this is the first work on functional verification of highly optimized implementations of additive FFTs used in Classic McEliece and HQC-128.

The rest of the paper is organized as follows: notations and preliminaries are provided in Section 2. We review Gao–Mateer and Frobenius additive FFTs in Section 3. Section 4 highlights our vector extension to the verification tool CRYPTO_{LINE} utilized in this work, followed by detailed descriptions of functional verification for both additive FFTs (Section 5). Evaluation of our techniques are reported with additional statistics in Section 6.

2 Preliminaries

A *commutative ring* $(R, 0, 1, +, \times)$ with *identity* $1 \neq 0$ is a structure such that (1) $(R, 0, +)$ is an abelian group; (2) $(R, 1, \times)$ is a commutative monoid; and (3) $a \times (b + c) = a \times b + a \times c$. A *field* $\mathbb{F} = (R, 0, 1, +, \times)$ is a commutative ring such that $(R, 1, \times)$ is an abelian group. For instance, $\mathbb{F}_p = (\{0, 1, \dots, p-1\}, 0, 1, +, \times)$ with prime p is a *prime field* where $+$ and

$\times$ are defined modulo p . The *order* of a field is its size. The *characteristic* of a field $\mathbb{F}$ is the least number c such that $c \times 1 = 0$. The order and characteristic of $\mathbb{F}_2$ are 2.

The *polynomial ring* over a commutative ring R with *indeterminate* X (denoted by $R[X]$) consists of polynomials with coefficients in R . We write $R^n[X]$ to denote the set of polynomials with degrees at most $n - 1$. A polynomial in $R[X]$ can have different representations in different radices. The (standard) monomial radix is X .

If $p(z) \in \mathbb{F}_2[z]$ is irreducible, $\mathbb{F}_2[z]/\langle p(z) \rangle$ is a polynomial ring such that $f(z), g(z) \in \mathbb{F}_2[z]$ are equal if $f(z) = g(z) \bmod p(z)$. $\mathbb{F}_2[z]/\langle p(z) \rangle$ is in fact a field of order 2^m where m is the degree of an irreducible polynomial $p(z)$. For instance, $\mathbb{F}_2[z]/\langle z^{12} + z^3 + 1 \rangle$ consists of polynomials of the form $\sum_{i=0}^{11} b_i z^i$ where $b_i \in \mathbb{F}_2$ for $0 \leq i < 12$. Since all fields of the same finite order are isomorphic, we will write $\mathbb{F}_{2^m}$ for the field of order 2^m . If the degree of the irreducible polynomial $p(z)$ is m , $\mathbb{F}_{2^m}$ is also a vector space $V_m = \{\sum_{i=0}^{m-1} b_i z^i \mid b_i \in \mathbb{F}_2\}$ over $\mathbb{F}_2$ of dimension m spanned by the *standard basis* $\gamma_m = \{1, z, z^2, \dots, z^{m-1}\}$.

2.1 Classic McEliece

Classic McEliece [McE78, ABC⁺22] is the venerable, conservative code-based cryptosystem: a Niederreiter-style KEM built on binary Goppa codes, whose security rests on the hardness of decoding a random-looking linear code and which has survived essentially unchanged since McEliece’s 1978 proposal [McE78] and Niederreiter’s slight modification [Nie86]. A parameter set is fixed by a triple (m, n, t) : the code lives in $\mathbb{F}_{2^m}$, has length $n \leq 2^m$, and corrects t errors, with public key the $mt \times (n - mt)$ parity-check matrix—which is why its keys are large, from hundreds of kilobytes to over a megabyte. ISO-standardized sets span NIST security levels 1,3,5: **mcEliece348864** ($m=12$, $n=3488$, $t=64$) at level 1, **mcEliece460896** ($m=13$, $n=4608$, $t=96$) at level 3, and three level-5 sets **mcEliece6688128**, **mcEliece6960119**, and **mcEliece8192128** (all $m=13$, with $n=6688, 6960, 8192$ and $t=128, 119, 128$), of which the **mcEliece6*** sizes are recommended for long-term security. Each comes in an interoperable **f** variant with faster key generation.

The routine we verify is a multipoint evaluation over $\mathbb{F}_{2^{13}}$ (or $\mathbb{F}_{2^{12}}$) during decapsulation. Decoding Goppa codes reduces, via Berlekamp–Massey, to error-locator polynomials over $\mathbb{F}_{2^m}$ whose roots mark the error positions; to find these positions means an evaluation at every element of the support — a full multipoint evaluation over a binary field. Naively this is quadratic; additive FFTs (of Gao–Mateer [GM10]/Bernstein–Chou–Schwabe “McBits” lineage [BCS13, Cho18]) finishes in quasi-linear time by recursing over a sequence of $\mathbb{F}_2$ -subspaces, and it is the single most expensive routine in the decoder.

2.2 HQC

HQC (Hamming Quasi-Cyclic) [AMBD⁺18, AAB⁺22] is a more efficiency-oriented KEM; its security rests on decoding, using structured ring arithmetic, to shrink keys to a few kB at the cost of a more elaborate decoder. Unlike McEliece, its ciphertext errors are removed by a *public, fixed* concatenated code—an inner Reed–Muller [Mul54, Ree54] and an outer Reed–Solomon [RS60] code (“RMRS”)—rather than by a secret trapdoor. Parameter sets HQC-128/-192/-256 are according to design security level, with the ring dimension n being the smallest primitive prime above the concatenated length, the Reed–Muller multiplicity ranging from 3–5, and the shortened Reed–Solomon length increasing accordingly. The public codes mean HQC’s decoder is key-independent, but arithmetic-heavy: decryption is dominated by large binary-polynomial multiplication in $\mathbb{F}_2[X]/(X^n + 1)$ and by the evaluation and interpolation inside Reed–Solomon decoding. The Reed–Solomon stage evaluates and interpolates polynomials over a binary extension field, and Frobenius additive FFT [vdHL17, CCPY26] replaces schoolbook evaluation with

a quasi-linear subspace recursion of the same shape as in McEliece — which is why a single verification methodology reaches both.

3 Additive Fast Fourier Transforms

Fast Fourier Transforms (FFTs) evaluate a polynomial $F(X) \in \mathbb{C}[X]$ at roots of unity where $\mathbb{C}$ denotes the field of complex numbers. Let $\{\zeta^i | 0 \leq i < 2^n\}$ be the roots of $X^{2^n} - 1$. FFTs can be seen as the following ring isomorphism from Chinese Remainder Theorem (see, for instance, Theorem III.2.25 in [Hum12]).

$$\frac{\mathbb{C}[X]}{\langle X^{2^n} - \zeta^0 \rangle} \simeq \frac{\mathbb{C}[X]}{\langle X^{2^{n-1}} - \zeta^0 \rangle} \times \frac{\mathbb{C}[X]}{\langle X^{2^{n-1}} - \zeta^{2^{n-1}} \rangle} \simeq \prod_{i=0}^{2^n-1} \frac{\mathbb{C}[X]}{\langle X - \zeta^{\text{brv}(i)} \rangle}$$

where $\text{brv}(i)$ is the number obtained by reversing bits of i .

The ring isomorphism of course is not restricted to algebraically closed fields like $\mathbb{C}$. FFTs can be applied to polynomials in $\mathbb{F}^{2^n}[X]$ on any field $\mathbb{F}$ with 2^n roots of unity. Number Theoretic Transforms evaluate polynomials in $\mathbb{F}_p^{2^n}[X]$ where $\mathbb{F}_p$ is a prime field with $p > 2$. FFTs however do not apply to fields with characteristic 2 directly. Since $\alpha = -\alpha$ in such fields, evaluation at ζ and $-\zeta$ gives the same result. Algebraically, the polynomial $X^2 - \alpha^2$ is factored into $(X - \alpha)^2$ in fields with characteristic 2. Since $\langle X - \alpha \rangle + \langle X - \alpha \rangle \neq \mathbb{F}[X]$, the Chinese Remainder Theorem is inapplicable. In order to apply Chinese Remainder Theorem, a quadratic polynomial factored to distinct polynomials is needed.

The key idea of *additive* FFTs (*aFFTs*) is to consider the quadratic polynomial $X^2 + X + \alpha^2 + \alpha$. Note that $(X + \alpha)(X + (\alpha + 1)) = X^2 + X + \alpha^2 + \alpha$ and $\langle X + \alpha \rangle + \langle X + \alpha + 1 \rangle = \mathbb{F}[X]$. Chinese Remainder Theorem gives the ring isomorphism

$$\frac{\mathbb{F}_{2^m}[X]}{\langle X^2 + X + \alpha^2 + \alpha \rangle} \simeq \frac{\mathbb{F}_{2^m}[X]}{\langle X + \alpha \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle X + \alpha + 1 \rangle}.$$

Just like FFTs, the ring isomorphism can be applied recursively to further decompose polynomial rings. Since the isomorphism is obtained by polynomials of the form $X^2 + X + \alpha^2 + \alpha$, aFFTs do not evaluate polynomials at roots of unity. They instead evaluate on subspaces over $\mathbb{F}_2$ of size 2^r . Two types of aFFTs have been developed and used in implementations of error correction codes and hence code-based post-quantum cryptography.

3.1 Gao-Mateer Additive Fast Fourier Transforms

Recall that $\mathbb{F}_{2^m} = \mathbb{F}_2[z]/\langle p(z) \rangle$ is a vector space with the standard basis $\gamma_m = \{1, z, z^2, \dots, z^{m-1}\}$ where m is the degree of $p(z)$. Gao-Mateer aFFTs evaluate a polynomial $F(X) \in \mathbb{F}_{2^m}[X]$ on the subspace U_r spanned by $\epsilon_r = \gamma_r$ [GM10]. To see how it works, represent $F(X)$ in radix $X^2 + X$:

$$\begin{aligned} F(X) &= \sum (f_{0,i} + f_{1,i}X)(X^2 + X)^i = \sum f_{0,i}(X^2 + X)^i + X \sum f_{1,i}(X^2 + X)^i \\ &= G_0(X^2 + X) + XG_1(X^2 + X) \end{aligned}$$

where $f_{0,i}, f_{1,i} \in \mathbb{F}_{2^m}$ for all i . Then $F(\alpha) = G_0(\alpha^2 + \alpha) + \alpha G_1(\alpha^2 + \alpha)$ and $F(\alpha + 1) = G_0(\alpha^2 + \alpha) + (\alpha + 1)G_1(\alpha^2 + \alpha) = F(\alpha) + G_1(\alpha^2 + \alpha)$ for $(\alpha + 1)^2 + (\alpha + 1) = \alpha^2 + \alpha$. To compute $F(\alpha)$ and $F(\alpha + 1)$, it suffices to evaluate $G_0(\alpha^2 + \alpha)$ and $G_1(\alpha^2 + \alpha)$. Note that the degrees of $G_0(X)$ and $G_1(X)$ are a half of the degree of $F(X)$.

Define U_r^- to be the space spanned by $\{z, z^2, \dots, z^{r-1}\}$ and $U_r^- + 1 = \{\alpha + 1 | \alpha \in U_r^-\}$. Then $U_r = U_r^- \cup (U_r^- + 1)$. For $\alpha \in U_r^-$, $F(\alpha)$ and $F(\alpha + 1)$ can be computed through

$G_0(\alpha^2 + \alpha)$ and $G_1(\alpha^2 + \alpha)$. It remains to evaluate $G_0(\alpha^2 + \alpha)$ and $G_1(\alpha^2 + \alpha)$ recursively. Define the space $\tilde{U}_{r-1} = \{\alpha^2 + \alpha | \alpha \in U_r^-\}$ spanned by $\delta_{r-1} = \{z^{2^i} + z^i | 1 \leq i < r\}$. We hence wish to evaluate $G_0(X), G_1(X)$ on $\tilde{U}_{r-1}$. But $\tilde{U}_{r-1}$ cannot be decomposed into $\tilde{U}_{r-1}^- \cup (\tilde{U}_{r-1}^- + 1)$ since $1 \notin \delta_{r-1}$. Although the degrees of $G_0(X)$ and $G_1(X)$ are reduced from $F(X)$ by half, the ring isomorphism is not directly applicable to subspaces.

In order to apply the ring isomorphism recursively, Gao-Mateer aFFTs transform the vector space $\tilde{U}_{r-1}$ by twisting (or change of variables). Define $F_i(X) = G_i((z^2 + z)X)$ for $0 \leq i < 2$. Then evaluating $G_i(X)$ on the space spanned by δ_{r-1} becomes computing $F_i(X)$ on the space spanned by $\epsilon_{r-1} = \{(z^2 + z)^{-1}(z^{2^i} + z^i) | 1 \leq i < r\} = \{1, z^2 + z, z^4 + z^3 + z^2, \dots, z^{2^{r-4}} + z^{2^{r-5}} + \dots + z^{r-2}\}$. Define U_{r-1} and U_{r-1}^- to be the spaces spanned by ϵ_{r-1} and $\epsilon_{r-1} \setminus \{1\}$ respectively. Then $U_{r-1} = U_{r-1}^- \cup (U_{r-1}^- + 1)$. The ring isomorphism is applicable to evaluating $F_i(X)$ on U_{r-1} recursively.

In short, Gao-Mateer aFFTs convert polynomials in the monomial radix X to $X^2 + X$ to exploit the ring isomorphism. After each level, the ring isomorphism fails because the basis is changed due to radix conversion. Twisting is then employed to normalize the basis so that the aFFTs can be applied to evaluate twisted polynomials recursively.

3.2 Frobenius Additive Fast Fourier Transforms

A different approach that ensures 1 belongs to bases of subspaces is developed in [vdHL17]. Frobenius aFFTs use a fixed basis for vector spaces but requires a more delicate radix to work. Concretely, consider *Cantor radix* $S_0(X) = X$ and $S_{i+1}(X) = S_i^2(X) + S_i(X)$. Then $S_1(X) = X^2 + X$ and $S_{i+1}(X) = S_1(S_i(X)) = S_i(S_1(X))$ by induction.

Consider the space $W_n = \{\sum_{i=1}^n b_i v_i | b_i \in \mathbb{F}_2\}$ spanned by *Cantor basis* $\beta_n = \{v_1, \dots, v_n\}$ where $v_0 = 0, v_1 = 1$ and $v_i = S_1(v_{i+1}) = v_{i+1}^2 + v_{i+1}$ for $1 \leq i$. We have $S_0(v_0) = S_1(v_1) = 0$ and $S_{i+1}(v_{i+1}) = S_i(S_1(v_{i+1})) = S_i(v_i) = 0$ by induction. Hence $S_i(v_j) = 0$ for $0 \leq j \leq i$. For $f, g \in \mathbb{F}_{2^n}$, $S_0(f+g) = f+g = S_0(f) + S_0(g)$ and $S_{i+1}(f+g) = S_i^2(f+g) + S_i(f+g) = (S_i(f) + S_i(g))^2 + (S_i(f) + S_i(g)) = S_i^2(f) + S_i(f) + S_i^2(g) + S_i(g) = S_{i+1}(f) + S_{i+1}(g)$. $S_i(X)$ is linear for all i . For $f = \sum_{i=1}^n b_i v_i$ with $b_i \in \mathbb{F}_2$, $S_n(f) = S_n(\sum_{i=1}^n b_i v_i) = \sum_{i=1}^n b_i S_n(v_i) = 0$. $S_n(X)$ vanishes on the space W_n spanned by β_n .

Let $F(X) \in \mathbb{F}_{2^m}[X]$. Frobenius aFFTs evaluate $F(X)$ on the vector space W_r . By Chinese Remainder Theorem,

$$\begin{aligned} \frac{\mathbb{F}_{2^m}[X]}{\langle S_r \rangle} &= \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-1}^2 + S_{r-1} + v_1^2 + v_1 \rangle} \simeq \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-1} \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-1} + v_1 \rangle} \\ &= \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-2}^2 + S_{r-2} + v_1^2 + v_1 \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-2}^2 + S_{r-2} + v_2^2 + v_2 \rangle} \\ &\simeq \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-2} \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-2} + v_1 \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-2} + v_2 \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-2} + v_2 + v_1 \rangle} \\ &\simeq \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-3} \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-3} + v_1 \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-3} + v_2 \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-3} + v_2 + v_1 \rangle} \times \\ &\quad \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-3} + v_3 \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-3} + v_3 + v_1 \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-3} + v_3 + v_2 \rangle} \times \frac{\mathbb{F}_{2^m}[X]}{\langle S_{r-3} + v_3 + v_2 + v_1 \rangle} \\ &\vdots \\ &\simeq \prod_{\alpha \in W_r} \frac{\mathbb{F}_{2^m}[X]}{\langle S_0 + \alpha \rangle} = \prod_{\alpha \in W_r} \frac{\mathbb{F}_{2^m}[X]}{\langle X + \alpha \rangle} \end{aligned}$$

Similar to FFTs, each ring isomorphism reduces the degree of polynomials by half. Frobenius aFFTs represent a polynomial in Cantor radix $S_{r-1}, S_{r-2}, \dots, S_0$ whereas the simpler radix

Type τ	::=	bit	Vector Instruction $\bar{t}$	::=	mov $[v_0, v_1, \dots, v_n]$ $\bar{\alpha}$
Vector Type $\bar{\tau}$	::=	$\tau[d]$			mov $\%v$ $\bar{\alpha}$ <i>move</i>
Var v	::=	$\mathbf{x} \mid \mathbf{y} \mid \mathbf{z}$			adds $\%c \ \%v \ \bar{\alpha} \ \bar{\alpha}$ <i>carrying addition</i>
Vector Var $\%v$	::=	$\%x \mid \%y \mid \%z$			and $\%v@{\bar{\tau}} \ \bar{\alpha} \ \bar{\alpha}$ <i>logical and</i>
Atom α	::=	$c@{\tau} \mid v \mid \%v[i]$			assert $\bar{\Delta} \ \bar{\mathbb{M}} \ \bar{\Sigma}$ <i>assertion</i>
Vector Atom $\bar{\alpha}$	::=	$[\alpha, \dots, \alpha] \mid \%v$			assume $\bar{\Delta} \ \bar{\mathbb{M}} \ \bar{\Sigma}$ <i>assumption</i>
		$\mid \bar{\alpha}++\bar{\alpha} \mid \bar{\alpha}++i$			...
		$c \in \mathbb{Z} \text{ and } i \in \mathbb{Z}^{\geq 0}$			

(a) Vector Type and Atom
(b) Vector Instruction

Figure 1: Language Syntax of vCRYPTOLINE

$X^2 + X = S_1$ is used in Gao-Mateer aFFTs. Radix conversion in Frobenius aFFTs is clearly more involved, but twisting is needed in Gao-Mateer aFFTs. Finally, Frobenius aFFTs evaluate on the subspace W_r spanned by Cantor basis $v_1, v_2, \dots, v_r$ while Gao-Mateer aFFTs evaluate on the subspace V_r spanned by the standard basis $1, z, z^2, \dots, z^{r-1}$.

4 Vector CryptoLine

CRYPTOLINE is a formal verification tool designed for low-level cryptographic programs [FLS⁺19, TFL⁺23]. It has verified assembly implementations of field and group operations in elliptic curves [FLS⁺19, ABB⁺25, ZHZ⁺24], hash functions [LLS⁺24], and Number Theoretic Transforms in lattice-based cryptography [HLS⁺22, CLT⁺25]. Elements of fields with characteristic 2 are naturally represented by polynomials with bit coefficients (or *bit* polynomials). Implementations of aFFTs easily perform bit operations over tens of thousands of variables. Extensions to CRYPTOLINE are thus necessary for formally verifying aFFT implementations in code-based cryptography. To this end, we introduce vCRYPTOLINE (short for *Vector CRYPTOLINE*), which adds vector support to CRYPTOLINE.

4.1 Modeling Language

To model bit polynomials, we extend CRYPTOLINE to support vector variables and instructions. In CRYPTOLINE, constants and variables are typed. *Types* in vCRYPTOLINE are extended to specify sizes of vectors as well. For instance, **bit** and **bit[256]** denote the unsigned 1-bit type and bit vector type of size 256 respectively (Figure 1a).

In vCRYPTOLINE, *vector variables* are prefixed with the percentage sign (%). *Atoms* in vCRYPTOLINE include constants $c@{\tau}$, variables v , and elements of a vector variable $\%v[i]$. *Vector atoms* include vector variables $\%v$ and atoms enclosed by square brackets $[\alpha, \dots, \alpha]$. The vector atom $\bar{\alpha}++\bar{\alpha}$ denotes the concatenation of two vector atoms. Similarly, the vector atom $\bar{\alpha}++i$ is the concatenation of i copies of $\bar{\alpha}$.

Scalar instructions are extended to vector instructions straightforwardly (Figure 1b). For instance, the vector move instruction **mov** $\%v \ \bar{\alpha}$ assigns the vector atom $\bar{\alpha}$ to the vector variable $\%v$. Another useful form of the vector move instruction is **mov** $[v_0, v_1, \dots, v_n] \ \bar{\alpha}$. It extracts elements from the vector atom $\bar{\alpha}$ and stores them in scalar variables $v_0, v_1, \dots, v_n$. The carrying addition instruction **adds** $\%c \ \%v \ \bar{\alpha}_0 \ \bar{\alpha}_1$ adds the two source vector atoms by components and stores corresponding carries and sums in the destination vector variables $\%c$ and $\%v$ respectively.

The **assert** and **assume** instructions verify and assume the property $\bar{\Delta} \ \bar{\mathbb{M}} \ \bar{\Sigma}$ respectively.

Expression ϵ	$::= \alpha \mid \text{poly } \epsilon \mid \epsilon + \epsilon \mid \epsilon - \epsilon \mid \epsilon \times \epsilon$
Vector Expression $\bar{\epsilon}$	$::= [\epsilon, \dots, \epsilon] \mid \bar{\alpha} \mid \bar{\epsilon} + \bar{\epsilon} \mid \bar{\epsilon} - \bar{\epsilon} \mid \bar{\epsilon} \times \bar{\epsilon}$
Moduli μ	$::= [\epsilon, \dots, \epsilon]$
Vector Moduli $\bar{\mu}$	$::= [\mu, \dots, \mu]$
Vector Constant $\bar{\nu}$	$::= [c@{\tau}, \dots, c@{\tau}]$
Algebraic Predicate $\bar{\Delta}$	$::= \epsilon = \epsilon \mid \bar{\epsilon} = \bar{\epsilon} \mid \epsilon \equiv \epsilon \bmod \mu \mid \bar{\epsilon} \equiv \bar{\epsilon} \bmod \bar{\mu}$
Bit-vector Predicate $\bar{\Sigma}$	$::= \epsilon = \epsilon \mid \bar{\epsilon} = \bar{\epsilon} \mid \epsilon < \epsilon \mid \bar{\epsilon} < \bar{\epsilon} \mid \epsilon \equiv \epsilon \bmod c@{\tau} \mid \bar{\epsilon} \equiv \bar{\epsilon} \bmod \bar{\nu}$

Figure 2: Atomic Vector Predicate in vCRYPTOLINE

The properties in **assert** are called *assertions* and those in **assume** are *assumptions*. When users wish to add new properties to speed up verification, these properties are often verified and then assumed by the **assert** and **assume** instructions. Since added properties are checked by vCRYPTOLINE, they are safe to be used for verification. The **assert-assume** pattern is quite useful to summarize computation for verification in practice.

vCRYPTOLINE also has its type system to detect simple errors in model construction. For instance, the two source vector atoms in **adds** instruction must have the same type. Adding two vector variables of different sizes induces a type error. If the two source vector atoms are of the same type, the type system infers the destination vector variable is also of the same type automatically. Users need not track sizes of every vector variables themselves.

4.2 Specification Language

The properties $\bar{\Delta} \wedge \bar{\Sigma}$ in **assert** and **assume** instructions are written in the vCRYPTOLINE specification language that is accordingly extended from CRYPTOLINE [FLS⁺19]. A property consists of *algebraic* and *bit-vector* predicates. Algebraic predicates $\bar{\Delta}$ are conjunctive formulas over atomic algebraic predicates. Bit-vector predicates $\bar{\Sigma}$ are quantifier-free formulas over atomic bit-vector predicates. Figure 2 shows atomic algebraic and bit-vector predicates provided by vCRYPTOLINE.

An *expression* is an atom, a polynomial ($\text{poly } \epsilon$), the sum, difference, or product of expressions. A *vector expression* is a vector atom ($\bar{\alpha}$), expressions enclosed by square brackets ($[\epsilon, \dots, \epsilon]$), or the sum, difference, or product of vector expressions. vCRYPTOLINE generalizes addition, subtraction, and multiplication to vector expressions componentwisely.

An *atomic algebraic predicate* is an equation or modular equation over expressions or vector expressions. If an equation or modular equation contains vector expressions, it denote the conjunction of equations or modular equations over corresponding components in vector expressions. For instance, the equation $\%v = [1@bit, 0@bit]$ denotes

$$\%v[0] = 1@bit \quad \wedge \quad \%v[1] = 0@bit.$$

An *atomic bit-vector predicate* is an equation or a comparison. Similar to atomic algebraic predicates, bit-vector predicates over vector expressions denote conjunctions of equations, modular equations, or comparisons over corresponding components.

4.3 Formal Verification with vCryptoLine

A vCRYPTOLINE *program* P is a sequence of vCRYPTOLINE instructions. Typically, a program starts with an **assume** $\bar{\Delta} \wedge \bar{\Sigma}$ and ends with an **assert** $\bar{\Delta}' \wedge \bar{\Sigma}'$ instruction. The starting **assume** $\bar{\Delta} \wedge \bar{\Sigma}$ instruction specifies input assumptions and we say $\bar{\Delta} \wedge \bar{\Sigma}$ is the *pre-condition* of the program. The ending **assert** $\bar{\Delta}' \wedge \bar{\Sigma}'$ instruction specifies output

requirements and we say $\overline{\Delta}' \bowtie \overline{\Sigma}'$ is the *post-condition*. Consider a program run which satisfies all assumptions including pre-condition. The run is *valid* if it also satisfies all assertions including post-condition. A program is *valid* if all runs of the program are valid.

When a program P is sent to vCRYPTOLINE for verification, the vCRYPTOLINE tool reports *PASS* or *FAIL*. Internally, vCRYPTOLINE translates vector constructs into equivalent CRYPTOLINE verification obligations, which are discharged by the underlying CRYPTOLINE verifier. The soundness of vCRYPTOLINE therefore follows from that of CRYPTOLINE, as stated in the following theorem:

Theorem 1. [FLS⁺19, TFL⁺23] *Let P be a vCRYPTOLINE program. If vCRYPTOLINE reports PASS, then P is valid.*

Note that validity of programs depends on their assumptions. If any assumption is not satisfiable by any run, the program is vacuously valid and vCRYPTOLINE will report *PASS* by definition. To avoid vacuous verification results, it is recommended to verify properties by **assert** instructions before making assumptions. The tool does not check whether it reports vacuous results. Assumptions must be made carefully.

In addition to checking properties about programs, vCRYPTOLINE also supports logic equivalence checking [LLS⁺24]. Given two programs with the same input variables, vCRYPTOLINE verifies if the two programs give the same output values on every possible input values. Logic equivalence checking is useful when post-conditions are too tedious to specify [LLS⁺24]. In this paper, we verify several highly optimized implementations with both property and logic equivalence checking. We first construct an inefficient reference model and verify it with property checking. A highly optimized implementation is then compared against the reference model with logic equivalence checking. If both verification tasks pass, the optimized implementation has the same functional behaviors as the reference model and the reference model satisfies its post-conditions. We conclude the optimized implementation must satisfy the same post-conditions. This strategy is useful when implementations are drastically different from underlying algebraic computation.

5 Functional Verification

Optimized implementations of Gao-Mateer and Frobenius aFFTs are available in Classic McEliece [ABC⁺22] and HQC-128 [CCPY26] respectively. Although they are both formulated by Chinese Remainder Theorem, Gao-Mateer and Frobenius aFFTs are different in implementation details and hence have different properties to verify. We give details of both optimized implementations and explain how to verify them.

5.1 Additive FFTs in Classic McEliece

Classic McEliece with parameter 348864 employs Gao-Mateer aFFTs to find roots of any $F(X) \in \mathbb{F}_{2^{12}}^{64}[X]$ by evaluating $F(X)$ on the space V_{12} spanned by $\gamma_{12} = \{1, z, z^2, \dots, z^{11}\}$. Since aFFTs evaluate $F(X)$ at all elements in $\mathbb{F}_{2^{12}} = \mathbb{F}_2[z]/\langle z^{12} + z^3 + 1 \rangle$, a root α of $F(X)$ can be found by inspecting if $F(\alpha) = 0$ [BCS13, Cho18]. Gao-Mateer aFFTs perform 6 levels of radix conversion for polynomials in $\mathbb{F}_{2^{12}}^{64}[X]$. All but the first level need twisting to normalize basis (Section 3). We explain how the optimized AVX2 implementation of Gao-Mateer aFFTs is verified by vCRYPTOLINE.

The AVX2 implementation represents polynomials from $\mathbb{F}_{2^{12}}^{64}[X]$ in the *bit slicing* format. Recall polynomials from $\mathbb{F}_{2^{12}}^{64}[X]$ have 64 coefficients in $\mathbb{F}_{2^{12}}$. A 64-bit word can store one bit of information from all 64 coefficients. A polynomial in $\mathbb{F}_{2^{12}}^{64}[X]$ is hence represented by 12 64-bit words. Concretely, let $F(X) = \sum_{i=0}^{63} f_i X^i$ with $f_i = \sum_{j=0}^{11} b_{i,j} z^j \in \mathbb{F}_{2^{12}}$ for

$0 \leq i < 64$. A 64-bit word $0bb_{63,0}b_{62,0} \cdots b_{1,0}b_{0,0}$ contains the least significant bits $b_{i,0}$ of all 64 coefficients of $F(X)$. Another 64-bit word $0bb_{63,1}b_{62,1} \cdots b_{1,1}b_{0,1}$ contains the least but one significant bits, and so forth. 12 64-bit words suffice to represent $F(X)$.

The bit slicing format enables efficient implementations for parallel field arithmetic operations. For instance, the AVX2 implementation provides vectorized field multiplication `vec256_mul_asm` computing 256 products of 256 pairs of field elements. In our VCRYPTOLINE model, 2×256 input field elements are represented by 2×12 256-bit vector variables $\%a_i$ and $\%b_i$ for $0 \leq i < 12$. Initially, all inputs are stored in memory. Our model thus initializes memory cells by inputs:

```
mov %MEM00 %a0;    mov %MEM01 %a1;    mov %MEM02 %a2;    mov %MEM03 %a3;
mov %MEM04 %a4;    mov %MEM05 %a5;    mov %MEM06 %a6;    mov %MEM07 %a7;
mov %MEM08 %a8;    mov %MEM09 %a9;    mov %MEM0a %a10;   mov %MEM0b %a11;
mov %MEM10 %b0;    mov %MEM11 %b1;    mov %MEM12 %b2;    mov %MEM13 %b3;
mov %MEM14 %b4;    mov %MEM15 %b5;    mov %MEM16 %b6;    mov %MEM17 %b7;
mov %MEM18 %b8;    mov %MEM19 %b9;    mov %MEM1a %b10;   mov %MEM1b %b11;
```

To load inputs to AVX2 256-bit register, we again use the VCRYPTOLINE `mov` instruction:

```
mov %ymm0 %MEM00;    (* vmovupd MEM00,%ymm0 *)
```

The AVX2 `vpxor` and `vpxor` instructions perform addition and multiplication in $\mathbb{F}_2$ respectively. They are hence modeled by VCRYPTOLINE `adds` and `mul` instructions respectively:

```
mul %ymm14 %ymm0 %ymm13;    (* vpxor %ymm13,%ymm0,%ymm14 *)
adds %dc %ymm1 %ymm1 %ymm14;    (* vpxor %ymm14,%ymm1,%ymm1 *)
```

At the end of `vec256_mul_asm`, the VCRYPTOLINE model loads the results to 12 256-bit vector variables $\%r_i$ from memory cells:

```
mov %r0 %MEMOUT0; mov %r1 %MEMOUT1; mov %r2 %MEMOUT2; mov %r3 %MEMOUT3;
mov %r4 %MEMOUT4; mov %r5 %MEMOUT5; mov %r6 %MEMOUT6; mov %r7 %MEMOUT7;
mov %r8 %MEMOUT8; mov %r9 %MEMOUT9; mov %r10 %MEMOUTa; mov %r11 %MEMOUTb;
```

To show the correctness of the AVX2 `vec256_mul_asm` program, VCRYPTOLINE verifies 256 modular equations

$$\begin{aligned} & \%r0[i] + \%r1[i]z + \%r2[i]z^2 + \cdots + \%r11[i]z^{11} \\ & \equiv (\%a0[i] + \%a1[i]z + \cdots + \%a11[i]z^{11}) \times (\%b0[i] + \%b1[i]z + \cdots + \%b11[i]z^{11}) \\ & \quad \text{mod } [2, z^{12} + z^3 + 1] \end{aligned}$$

for $0 \leq i < 256$. In the bit slicing format, the expression $\sum_{j=0}^{11} \%r_j[i]z^j$ denotes the i -th output polynomial. Each modular equation hence verifies if the output is the product of corresponding inputs in $\mathbb{F}_{2^{12}}$. Models for other functions are built and verified similarly.

5.1.1 Radix Conversion

Assume the input polynomial is $F(X) = \sum_{i=0}^{63} f_i X^i$ with $f_i \in \mathbb{F}_2[z]/\langle z^{12} + z^3 + 1 \rangle$ for $0 \leq i < 64$. Gao-Mateer aFFTs aim to evaluate $F(X)$ on the space U_{12} spanned by the basis $\epsilon_{12} = \{1, z, z^2, \dots, z^{11}\}$. Let $X_6 = X$ and $F_{6,0}(X_6) = F(X)$. The AVX2 implementation first converts $F_{6,0}(X_6)$ to radix $Y_5 = X_6^2 + X_6$ and obtains

$$F_{6,0}(X_6) \equiv G_{5,0}(Y_5) + X_6 G_{5,1}(Y_5) \text{ mod } [2, z^{12} + z^3 + 1] \quad (1)$$

where $G_{5,0}(Y_5), G_{5,1}(Y_5) \in \mathbb{F}_{2^{12}}[Y_5]$. The equation (1) is verified by VCRYPTOLINE.

Since $Y_5 = X_6^2 + X_6$, evaluating $F_{6,0}(X_6)$ on U_{12} spanned by ϵ_{12} amounts to evaluating $G_{5,i}(Y_5)$ on $\tilde{U}_{11}$ spanned by $\delta_{11} = \{z^2 + z, z^{2 \cdot 2} + z^2, \dots, z^{11 \cdot 2} + z^{11}\}$. Since $1 \notin \tilde{U}_{11}$, the

radix conversion normalizes δ_{11} by twisting. Concretely, define $X_5 = (z^2 + z)^{-1}Y_5$ and $F_{5,i}(X_5) = G_{5,i}(Y_5)$. We have

$$F_{6,0}(X_6) \equiv F_{5,0}(X_5) + X_6 F_{5,1}(X_5) \pmod{[2, z^{12} + z^3 + 1]} \quad (2)$$

where $F_{5,j}(X_5) \in \mathbb{F}_{2^{12}}^{32}[X_5]$ for $0 \leq j < 2$. Now $X_5 = (z^2 + z)^{-1}Y_5 = (z^2 + z)^{-1}(X_6^2 + X_6)$. Evaluating $F_{5,i}(X_5)$ on the space U_{11} spanned by $\epsilon_{11} = \{1, (z^2 + z)^{-1}(z^4 + z^2), (z^2 + z)^{-1}(z^6 + z^4), \dots, (z^2 + z)^{-1}(z^{22} + z^{11})\} = \{1, z^2 + z, z^4 + z^3 + z^2, \dots, z^9 + z^2 + z + 1\}$ amounts to values of $G_{5,i}(Y_5)$ on $\tilde{U}_{11}$ as wished. The modular equation (2) is verified by vCRYPTOLINE. Verification of the first level of radix conversion is finished.

The AVX2 implementation continues to convert $F_{5,i}(X_5) \in \mathbb{F}_{2^{12}}^{32}[X_5]$ to radix $Y_4 = X_5^2 + X_5$ at the second level. vCRYPTOLINE verifies the following modular equations:

$$F_{5,i}(X_5) \equiv G_{4,2i}(Y_4) + X_5 G_{4,2i+1}(Y_4) \pmod{[2, z^{12} + z^3 + 1]} \quad (0 \leq i < 2)$$

where $G_{4,j}(Y_4) \in \mathbb{F}_{2^{12}}^{16}[Y_4]$ for $0 \leq j < 4$. Evaluating $G_{4,j}(Y_4)$ on $\tilde{U}_{10}$ spanned by $\delta_{10} = \{(z^2 + z)^2 + (z^2 + z), (z^4 + z^3 + z^2)^2 + (z^4 + z^3 + z^2), \dots, (z^9 + z^2 + z + 1)^2 + (z^9 + z^2 + z + 1)\} = \{z^4 + z, z^8 + z^6 + z^3 + z^2, \dots, z^6 + z^4 + z\}$ gives us values of $F_{5,i}(X_5)$ on U_{11} . Define $X_4 = (z^4 + z)^{-1}Y_4$. The following modular equations for $F_{4,j}(X_4)$ are verified:

$$F_{5,i}(X_5) \equiv F_{4,2i}(X_4) + X_5 F_{4,2i+1}(X_4) \pmod{[2, z^{12} + z^3 + 1]} \quad (0 \leq i < 2)$$

where $F_{4,j}(X_4) \in \mathbb{F}_{2^{12}}^{16}[X_4]$ for $0 \leq j < 4$. Now the evaluation of $F_{4,j}(X_4)$ on the space U_{10} spanned by $\epsilon_{10} = \{1, (z^4 + z)^{-1}(z^8 + z^6 + z^3 + z^2), \dots, (z^4 + z)^{-1}(z^6 + z^4 + z)\} = \{1, z^4 + z^2 + z, \dots, z^{11} + z^8 + z^5 + z^2 + 1\}$ gives values of $F_{5,i}(X_5)$ on U_{11} as desired.

After 6 levels of radix conversion, we have additionally $X_3 = (z^8 + z)^{-1}(X_4^2 + X_4)$, $X_2 = (z^{16} + z)^{-1}(X_3^2 + X_3)$, $X_1 = (z^{32} + z)^{-1}(X_2^2 + X_2)$ and obtain

$$F_{1,i}(X_1) \equiv F_{0,2i} + F_{0,2i+1}X_1 \pmod{[2, z^{12} + z^3 + 1]} \quad (0 \leq i < 32)$$

where $F_{0,j} \in \mathbb{F}_{2^{12}}$ for $0 \leq j < 64$. The 32 modular equations are verified by vCRYPTOLINE. The verification of radix conversion is complete.

5.1.2 Butterflies

Butterflies in Classic McEliece start by evaluating $F_{1,i}(X_1) \in \mathbb{F}_{2^{12}}^2[X_1]$ ($0 \leq i < 32$) at the 2^7 values spanned by the basis $\epsilon_7 = \{1, z^9 + z^3, z^8 + z^7 + z^5, z^{11} + z^9 + z^4 + z^2 + z + 1, z^{11} + z^{10} + z^8 + z^7 + z^5 + z^4 + z^3 + z^2 + 1, z^{11} + z^6 + z^5 + z^4 + z^3 + z + 1, z^7 + z^5 + z\}$.

At the end of the first level of butterflies, the AVX2 implementation obtains 32×128 output values $v_{1,i,j} \in \mathbb{F}_{2^{12}}$ for $0 \leq i < 32$ and $0 \leq j < 128$ such that

$$\begin{aligned} v_{1,i,0} &\equiv F_{1,i}(X_1) \pmod{[X_1, z^{12} + z^3 + 1, 2]} \\ v_{1,i,1} &\equiv F_{1,i}(X_1) \pmod{[X_1 + 1, z^{12} + z^3 + 1, 2]} \\ v_{1,i,2} &\equiv F_{1,i}(X_1) \pmod{[X_1 + z^9 + z^3, z^{12} + z^3 + 1, 2]} \\ &\vdots \\ v_{1,i,127} &\equiv F_{1,i}(X_1) \pmod{[X_1 + z^{11} + z^{10} + z^7 + z^6 + z^4 + z^3 + z, z^{12} + z^3 + 1, 2]} \end{aligned}$$

Each modular equation verifies if $F_{1,i}(X_1)$ is evaluated correctly. For instance, the third modular equation checks if the output $v_{1,i,2}$ is equal to $F_{1,i}(X_1)$ when $X_1 = z^9 + z^3$ in $\mathbb{F}_{2^{12}}$. vCRYPTOLINE verifies these $4096 = 32 \times 128$ modular equations successfully.

The 16 polynomials $F_{2,i}(X_2) = F_{1,2i}(X_1) + X_2 F_{1,2i+1}(X_1) \in \mathbb{F}_{2^{12}}^4[X_2]$ ($0 \leq i < 16$) is evaluated by the AVX2 implementation at 2^8 values spanned by the basis $\epsilon_8 = \{1, z^8 +$

$z^7 + z^2 + z, z^{11} + z^{10} + z^7 + z^6 + z^5, z^{11} + z^{10} + z^3 + z^2, z^7 + z^4 + z^3, z^{11} + z^8 + z^4 + z + 1, z^6 + z^4 + z^3 + 1, z^5 + z^2\}$. vCRYPTOLINE verifies $4096 = 16 \times 256$ output values $v_{2,i,j}$ ($0 \leq i < 16$ and $0 \leq j < 256$) satisfy corresponding modular equations:

$$\begin{aligned} v_{2,i,0} &\equiv F_{2,i}(X_2) \pmod{[X_2, z^{12} + z^3 + 1, 2]} \\ v_{2,i,1} &\equiv F_{2,i}(X_2) \pmod{[X_2 + 1, z^{12} + z^3 + 1, 2]} \\ &\vdots \\ v_{2,i,255} &\equiv F_{2,i}(X_2) \pmod{[X_2 + z^{11} + z^{10} + z^7 + z^6 + z^4 + z^3 + z, z^{12} + z^3 + 1, 2]} \end{aligned}$$

At the end of all 6 levels of butterflies, $F_{6,0}(X_6) = F(X)$ is evaluated at all elements of $\mathbb{F}_{2^{12}}$ spanned by $\epsilon_{12} = \{1, z, z^2, \dots, z^{11}\}$. Formally, 4096 output values $v_{6,0,j}$ are verified to satisfy the following modular equations:

$$\begin{aligned} v_{6,0,0} &\equiv F_{6,0}(X_6) \pmod{[X_6, z^{12} + z^3 + 1, 2]} \\ v_{6,0,1} &\equiv F_{6,0}(X_6) \pmod{[X_6 + 1, z^{12} + z^3 + 1, 2]} \\ v_{6,0,2} &\equiv F_{6,0}(X_6) \pmod{[X_6 + z, z^{12} + z^3 + 1, 2]} \\ v_{6,0,3} &\equiv F_{6,0}(X_6) \pmod{[X_6 + 1 + z, z^{12} + z^3 + 1, 2]} \\ &\vdots \\ v_{6,0,4095} &\equiv F_{6,0}(X_6) \pmod{[X_6 + 1 + z + z^2 + \dots + z^{11}, z^{12} + z^3 + 1, 2]} \end{aligned}$$

Overall, vCRYPTOLINE verified 6×4096 modular equations in butterflies. This concludes the verification of the AVX2 implementation for Gao-Mateer aFFTs in Classic McEliece.

5.2 Additive FFTs in HQC-128

In [CCPY26], Frobenius aFFTs are employed to perform polynomial multiplication in $\mathbb{F}_2^{2^{14}}[X]$. Consider an input polynomial $F(X) = \sum_{i=0}^{2^{15}-1} b_i X^i \in \mathbb{F}_2^{2^{15}}[X]$ with coefficients $b_i = 0$ for $2^{14} \leq i < 2^{15}$. The input polynomial $F(X)$ is stored in 64 256-bit memory cells. In the vCRYPTOLINE model, the 2^{14} bit coefficients are modeled by input bit variables `b0000`, `b0000`, ..., `b3fff`. The model first initializes memory cells with the `mov` instruction:

```
mov %MEM00 [b0000, b0001, b0002, b0003, b0004, b0005, b0006, b0007, b0008, b0009,
             b000a, b000b, b000c, b000d,      . . . . . , b00fc, b00fd, b00fe, b00ff];
. . .
mov %MEM3f [b3f00, b3f01, b3f02, b3f03, b3f04, b3f05, b3f06, b3f07, b3f08, b3f09,
             b3f0a, b3f0b, b3f0c, b3f0d,      . . . . . , b3ffc, b3ffd, b3ffe, b3fff];
```

Hence we have the following input polynomial in the standard monomial radix:

$$F(X) = \sum_{i=0}^{255} \%MEM00[i] X^i + \sum_{i=0}^{255} \%MEM01[i] X^{256+i} + \dots + \sum_{i=0}^{255} \%MEM3f[i] X^{63 \cdot 256 + i}$$

5.2.1 Radix Conversion

Frobenius aFFTs take a polynomial $F(X) = \sum_{i=0}^{2^{15}-1} b_i X^i \in \mathbb{F}_2^{2^{15}}[X]$ in the monomial radix and convert it to Cantor radix. Radix conversion is computed by dividing radix repeatedly. Define $F_{15,0}(X) = F(X)$ and $F_{i+1,j}(X) = F_{i,2j}(X) + S_i F_{i,2j+1}(X)$ where $F_{i,j} \in \mathbb{F}_2^{2^i}[X]$ for $0 \leq i < 15$ and $0 \leq j < 2^{15-i}$. Let $[S]_0 = 1$ and $[S]_k = S_{i_0} S_{i_1} \dots S_{i_j}$ where $k = 2^{i_0} + 2^{i_1} + \dots + 2^{i_j}$ with $i_0 < i_1 < \dots < i_j$. Recall the degree of S_i is hence 2^i .

The degree of $[S]_k$ is $2^{i_0} + 2^{i_1} + \dots + 2^{i_j} = k$. For radix conversion, we wish to convert $F_{15,0}$ in monomial radix X to Cantor radix $S_{14}, S_{13}, \dots, S_0$ through a sequence of divisions:

$$\begin{aligned} F_{15,0}(X) &= F_{14,0}(X) + S_{14}F_{14,1} \\ &= F_{13,0}(X) + S_{13}F_{13,1}(X) + S_{14}(F_{13,2}(X) + S_{13}F_{13,3}(X)) = \dots = \sum_{j=0}^{2^{15}-1} [S]_j F_{0,j} \end{aligned}$$

where $F_{0,j} \in \mathbb{F}_2$ for $0 \leq j < 2^{15}$.

Dividing S_i however is not equally easy. For instance, dividing $S_8 = X^{256} + X$ is simpler than dividing $S_7 = X^{128} + X^{64} + X^{32} + X^{16} + X^8 + X^4 + X^2 + X$. To improve efficiency, optimized implementations carry out radix conversion in three phases. In the first phase, the polynomial $F(X)$ in the monomial radix is converted to $G(X)$ in radix S_8 such that

$$F(X) = G(X) = \sum_{i=0}^{127} S_8^i G_{8,i}(X) = G_{8,0}(X) + S_8 G_{8,1}(X) + S_8^2 G_{8,2}(X) + \dots + S_8^{127} G_{8,127}(X)$$

where $G_{8,i}(X) \in \mathbb{F}_2^{256}[X]$ is in the monomial radix for $0 \leq i < 128$.

The second phase converts $G(X)$ in Cantor radix S_8 to $H(X)$ in radix $S_{14}, S_{13}, \dots, S_8$. Define $[S_{+h}]_0 = 1$ and $[S_{+h}]_k = S_{i_0+h} S_{i_1+h} \dots S_{i_j+h}$ when $k = 2^{i_0} + 2^{i_1} + \dots + 2^{i_j}$ and $i_0 < i_1 < \dots < i_j$. The second phase computes $H(X)$ such that

$$\begin{aligned} G(X) = H(X) &= \sum_{j=0}^{127} [S_{+8}]_j H_{8,j}(X) \\ &= H_{8,0}(X) + S_8 H_{8,1}(X) + S_9 H_{8,2}(X) + S_8 S_9 H_{8,3}(X) + \dots + S_8 S_9 \dots S_{14} H_{8,127}(X) \end{aligned}$$

where $H_{8,j}(X) \in \mathbb{F}_2^{256}[X]$ are in the standard monomial radix for $0 \leq j < 128$.

The last phase converts each $H_{8,j}(X)$ to Cantor radix $S_7, S_6, \dots, S_0$. That is,

$$\begin{aligned} H_{8,j}(X) &= \sum_{k=0}^{255} [S]_k H_{0,j,k} \\ &= H_{0,j,0} + S_0 H_{0,j,1} + S_1 H_{0,j,2} + S_0 S_1 H_{0,j,3} + \dots + S_0 S_1 \dots S_7 H_{0,j,255} \end{aligned}$$

where $H_{0,j,k} \in \mathbb{F}_2$ for $0 \leq j < 128$ and $0 \leq k < 256$. Observe that the second and third phases are orthogonal and can be swapped. Indeed, the NEON implementation computes the third phase before the second. Also, the first phase picks the radix $S_8 = X^{256} + X$ because S_8 is easy to divide on architectures with 256-bit vector registers. The NEON implementation divides $F(X)$ by S_8 with 128-bit vector registers in the first phase as well. This is because division by $S_7 = X^{128} + X^{64} + X^{32} + X^{16} + X^8 + X^4 + X^2 + X$ is still much more complicated than of S_8 with 128-bit vector instructions on NEON.

Further optimizations are implemented in each phase. In the first phase, the input polynomial in the standard monomial radix is converted to radix $S_8 = X^{256} + X$ by division. The AVX2 and NEON implementations thus utilize permutation and logic shift vector instructions. Corresponding vCRYPTOLINE models are obtained by almost direct translation. Consider the division of $S_8^2 = (X^{256} + X)^2 = X^{8192} + X^{32}$. The following fragment shows the vCRYPTOLINE model with AVX2 assembly instructions in comments:

```
mov %ymm1 %MEM3f; (* vmovdqu MEM3f,%ymm1 *)
mov %ymm4 %MEM3e; (* vmovdqu MEM3e,%ymm4 *)
(* vperm2i128 %$0x21,%ymm1,%ymm4,%ymm7 *)
mov %ymm7 (%ymm4[128:256]++%ymm1[0:128]);
```

```

(* vpsllq $0xc,%ymm7,%ymm1,%ymm1 *)
mov %ymm1 (%ymm7[ 96:128]++%ymm1[ 0: 96]++%ymm7[224:256]++%ymm1[128:224]);
adds %dc %ymm1 %ymm1 %MEM1f;
mov %MEM1f %ymm1;
(* vpxor MEM1f,%ymm1,%ymm1 *)
(* vmovdqu %ymm1, MEM1f *)

```

VCRYPTOLINE vector variables `%ymm1`, `%ymm4`, and `%ymm7` of type `bit[256]` model corresponding AVX2 256-bit vector registers. After loading input coefficients in `MEM3f` and `MEM3e`, the AVX2 implementation puts the high and low 128 bits of `ymm4` and `ymm1` AVX2 registers respectively to `ymm7` by the permutation instruction `vperm2i128`. The VCRYPTOLINE `mov` instruction thus sets the vector variable `%ymm7` to the vector atom `%ymm4[128:256]++%ymm1[0:128]` by extracting corresponding bits from `%ymm4` and `%ymm1`. The AVX2 packed align right instruction `vpsllq` is modeled similarly.

The AVX2 `vpxor` instruction performs the algebraic addition operation in $\mathbb{F}_2$, the VCRYPTOLINE model therefore uses the add-with-carry vector instruction `adds`. The last `mov` instruction stores the sum in memory. We use VCRYPTOLINE to verify the first phase by checking the following modular equation:

$$F(X) \equiv G(X) = \sum_{i=0}^{255} \%MEM00[i]X^i + S_8 \sum_{i=0}^{255} \%MEM01[i]X^i + \cdots + S_8^{63} \sum_{i=0}^{255} \%MEM3f[i]X^i \pmod{2}$$

The second phase converts polynomials in radix S_8 to radix $S_{14}, S_{13}, \dots, S_8$. Note that radix $S_9 = S_8^2 + S_8$, $S_{10} = S_9^2 + S_9 = S_8^4 + S_8$, and $S_{12} = S_8^{16} + S_8$ are easy to divide with polynomials in radix S_8 . Division by $S_{11} = S_8^8 + S_8^4 + S_8^2 + S_8$ on the other hand needs more operations. Nevertheless, it is easier to divide radix $S_{11} = S_{10}^2 + S_{10}$ with polynomials in radix S_{10} . Both AVX2 and NEON implementations therefore convert to radix S_{10}, S_9, S_8 from radix S_8 , and then convert to S_{11} with radix S_{10} . Similarly, division of $S_{13} = S_8^{32} + S_8^{16} + S_8^8 + S_8 = S_{12}^2 + S_{12}$ is hard for polynomials in radix S_8 but easy for radix S_{12} . Both implementations convert to radix S_{13} after polynomials are converted to radix $S_{12}, S_{11}, \dots, S_8$. VCRYPTOLINE verifies the following equation for this phase:

$$\begin{aligned}
G(X) \equiv H(X) &= \sum_{i=0}^{255} \%MEM00[i]X^i + S_8 \sum_{i=0}^{255} \%MEM01[i]X^i + S_9 \sum_{i=0}^{255} \%MEM02[i]X^i + \\
&\quad S_8 S_9 \sum_{i=0}^{255} \%MEM03[i]X^i + \cdots + S_8 S_9 \cdots S_{13} \sum_{i=0}^{255} \%MEM3f[i]X^i \pmod{2}
\end{aligned}$$

The last phase converts polynomials $H_{8,j}(X) = \sum_{i=0}^{255} \%MEM_j[i]X^i$ in the standard monomial radix to radix $S_7, S_6, \dots, S_0$ where $0 \leq j < 64$. The radix $S_4 = X^{16} + X$ and $S_2 = X^4 + X$ are easy to divide, but others need more work. Similar to the second phase, the third phase first converts $H_{8,j}(X)$ to radix S_4 and then to radix S_7, S_6, S_5, S_4 by exploiting the equations $S_7 = S_6^2 + S_6$, $S_6 = S_4^4 + S_4$, and $S_5 = S_4^2 + S_4$. Using various permutation and logical shift vector instructions, we obtain

$$\begin{aligned}
H_{8,j}(X) &\equiv \sum_{k=0}^{15} [S_{+4}]_k H_{4,j,k}(X) \\
&= H_{4,j,0}(X) + S_4 H_{4,j,1}(X) + S_5 H_{4,j,2}(X) + \cdots + S_4 S_5 S_6 S_7 H_{4,j,15}(X) \pmod{2}
\end{aligned}$$

where $H_{4,j,k}(X) \in \mathbb{F}_2^{16}[X]$ is in the monomial radix for $0 \leq j < 64$ and $0 \leq k < 16$.

It remains to convert $H_{4,j,k}(X)$ to radix S_3, S_2, S_1, S_0 . Observe the computation now involves 16-bit batches or less. Vector instructions are not available at such granularity. Efficient implementations naturally employ different optimizations.

AVX2 Implementation. To represent a polynomial of degree 15 in radix $S_3 = X^8 + X^4 + X^2 + X$, note $X^8 \equiv X^4 + X^2 + X + S_3 \pmod{2}$ and

$$\begin{aligned} H_{4,j,k}(X) &= \sum_{i=0}^{15} c_{j,k,i} X^i = \sum_{i=0}^7 c_{j,k,i} X^i + X^8 \sum_{i=0}^7 c_{j,k,i+8} X^i = \underline{H}_{3,j,k}(X) + X^8 \overline{H}_{3,j,k}(X) \\ &\equiv \underline{H}_{3,j,k}(X) + (X^4 + X^2 + X) \overline{H}_{3,j,k}(X) + S_3 \overline{H}_{3,j,k}(X) \pmod{2}. \end{aligned}$$

If one can compute $1024 = 64 \cdot 16$ product polynomials $(X^4 + X^2 + X) \overline{H}_{3,j,k}(X) \in \mathbb{F}_2^{12}[X]$ in parallel, the conversion to radix S_3 will be very efficient. AVX2 does offer bit polynomial multiplication instructions for polynomials in $\mathbb{F}_2^{64}[X]$. They are however not very useful to lots of low-degree bit polynomial multiplications.

To parallelize low-degree polynomial multiplications, the AVX2 implementation employs the shuffle instruction. Observe that there are only 16 bit polynomials in $\mathbb{F}_2^4[X]$ and products of $X^4 + X^2 + X$ with a polynomial in $\mathbb{F}_2^4[X]$ are in $\mathbb{F}_2^8[X]$. If one pre-computes all 16 possible products, polynomial multiplication can be implemented by table lookups. In the AVX2 implementation, the byte shuffle instruction `vpshufb` is used to look up products in $\mathbb{F}_2^8[X]$. 32 polynomial multiplications with $X^4 + X^2 + X$ are hence computed in one vector instruction. This trick greatly improves the speed of radix conversion for low-degree polynomials.

For formal verification, polynomial multiplication with $X^4 + X^2 + X$ has two VCRYPTOLINE models. The algebraic model specifies polynomial multiplication by arithmetic instructions; it is used to verify radix conversion algebraically. The bitvector model specifies polynomial multiplication by bit shuffles; it is used to model the computation of the AVX2 shuffle instruction. We first verify the algebraic model converts low-degree polynomials to Cantor radix correctly. Logic equivalence checking is then utilized to verify the outputs of bitvector and algebraic models are identical on every input bit pattern. Since the bitvector model computes the same function as the algebraic model and the algebraic model performs radix conversion correctly, we conclude the low-degree polynomial multiplication with $X^4 + X^2 + X$ by the AVX2 byte shuffle instruction is correct.

NEON Implementation. The NEON polynomial multiply instruction `pmul` multiplies 16 polynomials in $\mathbb{F}_2^8[X]$ stored in 128-bit vector registers in parallel. It is used to convert polynomials in $\mathbb{F}_2^{16}[X]$ to radix S_3 without table lookups. After converting to radix S_3 , we have 2048 polynomials from $\mathbb{F}_2^8[X]$ in the standard monomial basis

$$H_{3,j,k,l}(X) = \sum_{i=0}^7 d_{j,k,l,i} X^i$$

where $d_{j,k,l,i} \in \mathbb{F}_2$ with $0 \leq j < 64, 0 \leq k < 16, 0 \leq l < 2$. It remains to divide polynomials in $\mathbb{F}_2^8[X]$ by $S_2 = X^4 + X$ and $S_1 = X^2 + X$. NEON does not offer vector instructions at such granularity. The implementation again performs division by table lookups. Consider

$$\begin{aligned} H_{3,j,k,l}(X) &= \sum_{i=0}^7 d_{j,k,l,i} X^i = \sum_{i=0}^3 d_{j,k,l,i} X^i + X^4 \sum_{i=0}^3 d_{j,k,l,i+4} X^i \\ &= \underline{H}_{2,j,k,l}(X) + X^4 \overline{H}_{2,j,k,l}(X). \end{aligned}$$

To convert $H_{3,j,k,l}(X)$ to radix S_2 and S_1 , $X^4 \overline{H}_{2,j,k,l}(X)$ has to be divided by S_2 and then by S_1 . But $\overline{H}_{2,j,k,l}(X) \in \mathbb{F}_2^4[X]$, one can use its bit pattern as an index to find the result of two divisions with one NEON table lookup instruction `tbl`. After $\overline{H}_{2,j,k,l}(X)$ is converted, the remaining polynomial in $\mathbb{F}_2^4[X]$ is converted to radix S_1 by another table lookup instruction. The AVX2 implementation has the same optimization.

For verification, we give two models for each table lookup instruction. The algebraic model specifies functional behaviors; the bitvector model simulates the table lookup instruction. We verify the algebraic model converts 16 polynomials in $\mathbb{F}_2^8[X]$ to radix S_2, S_1 correctly. Logic equivalence checking then ensures the bitvector and algebraic models compute the same function. The AVX2 implementation is verified similarly.

Summary of Radix Conversion. At the end of radix conversion, vCRYPTOLINE verifies the following modular equation for any input polynomial $F(X) \in \mathbb{F}_2^{2^{15}}[X]$

$$F(X) \equiv \sum_{j=0}^{63} [S_{+8}]_j \sum_{i=0}^{255} [S]_i \% \text{MEM}_j[i] \pmod{2}. \quad (3)$$

For forward Frobenius aFFTs in HQC-128, input polynomials in $\mathbb{F}_2^{2^{14}}[X]$ are converted to Cantor radix $S_{13}, S_{12}, \dots, S_0$. For backward aFFTs, input polynomials are in $\mathbb{F}_2^{2^{15}}[X]$ and converted to the standard monomial radix. Radix conversions in forward and backward aFFTs are verified by property checking in vCRYPTOLINE.

5.2.2 Embedding

After radix conversion, Frobenius aFFTs evaluate polynomials on the subspace spanned by Cantor basis $\beta_{14} = \{v_1, v_2, \dots, v_{14}\}$ (Section 3). Yet the order of $\mathbb{F}_2$ is 2. It is impossible to find Cantor bases in such a small field. To address the problem, $\mathbb{F}_2^{2^{15}}[X]$ is embedded in $\mathbb{F}_{2^{64}}^{512}[X]$ so that Cantor bases in $\mathbb{F}_{2^{64}}$ are found. Note that polynomials in $\mathbb{F}_2^{2^{15}}[X]$ and $\mathbb{F}_{2^{64}}^{512}[X]$ are all represented by 2^{15} bits. The embedding is in fact an isomorphism.

To see how the embedding works, recall polynomials in $\mathbb{F}_2^{2^{15}}[X]$ are in radix $S_{14}, S_{13}, \dots, S_0$ now. Consider the 64 terms in products of $S_{14}, S_{13}, \dots$, or S_9

$$e_0 + e_1 S_9 + e_2 S_{10} + e_3 S_9 S_{10} + e_4 S_{11} + \dots + e_{63} S_9 S_{10} \dots S_{14} \quad (4)$$

where $e_i \in \mathbb{F}_2$ for $0 \leq i < 64$. If $S_{14}, S_{13}, \dots, S_9$ are replaced by elements from $\mathbb{F}_{2^{64}}$, we obtain a mapping from any bit polynomial in radix $S_{14}, S_{13}, \dots, S_9$ to $\mathbb{F}_{2^{64}}$. It remains to find elements in $\mathbb{F}_{2^{64}}$ to replace $S_{14}, S_{13}, \dots, S_9$. Since the polynomial is in radix $S_{14}, S_{13}, \dots, S_9$, it is natural to choose elements from Cantor basis in $\mathbb{F}_{2^{64}}$. However, arbitrary elements in Cantor basis would not work since they might not represent all possible 64-coefficient polynomials. In both implementations, $v_{28}, \dots, v_{32}, v_{33}^1$ are chosen [LCK⁺18]. We therefore obtain the following modular equation after embedding

$$F(X) \equiv \sum_{i=0}^{511} [S]_i f_i \pmod{[2, S_{14} + v_{28}, S_{13} + v_{29}, \dots, S_9 + v_{33}]} \quad (5)$$

where $f_i \in \mathbb{F}_{2^{64}}$ for $0 \leq i < 512$. The embedding can also be seen as evaluating $F(X)$ at $S_9 = v_{33}$. Since $S_{i+1} = S_i^2 + S_i$ and $v_i = v_{i+1}^2 + v_{i+1}$, we obtain $S_{10}, S_{11}, \dots, S_{14}$ are respectively equal to $v_{32}, v_{31}, \dots, v_{28}$. This gives the embedding of $\mathbb{F}_2^{2^{15}}[X]$ in $\mathbb{F}_{2^{64}}^{512}[X]$.

To compute the embedding efficiently, recall the bit polynomial with 64 coefficients in (4). After the substitution of $S_{14}, S_{13}, \dots, S_9$ by $v_{28}, v_{29}, \dots, v_{33}$, the result is simply a linear combination of 64 $\mathbb{F}_{2^{64}}$ elements $1, v_{33}, v_{32}, v_{33}v_{32}, v_{31}, v_{33}v_{31}, \dots, v_{33}v_{32} \dots v_{28}$. If the 64 elements are pre-computed, the embedding is but a bit matrix-vector multiplication and optimized by table lookups. The AVX2 and NEON implementations are further

$$^1 v_{33} = z^{63} + z^{62} + z^{61} + z^{59} + z^{58} + z^{56} + z^{54} + z^{52} + z^{50} + z^{49} + z^{48} + z^{47} + z^{46} + z^{43} + z^{42} + z^{41} + z^{38} + z^{37} + z^{36} + z^{34} + z^{33} + z^{32} + z^{31} + z^{27} + z^{26} + z^{25} + z^{24} + z^{19} + z^{18} + z^{16} + z^{14} + z^{13} + z^9 + z^6 + z^2.$$

optimized to exploit vector instructions. Instead of computing one $\mathbb{F}_{2^{64}}$ element for a bit polynomial with 64 coefficients at a time, 8 field elements for 8 bit polynomials are computed simultaneously to simplify memory access patterns.

To verify both implementations, we construct an algebraic model which computes one embedding by straightforward matrix-vector multiplication. vCRYPTOLINE first verifies that the algebraic model indeed computes the mapping from bit polynomials in radix $S_{14}, S_{13}, \dots, S_9$ to elements in $\mathbb{F}_{2^{64}}$. Logic equivalence checking then verifies bitvector models derived from AVX2 and NEON implementations compute the same function as the algebraic model. Finally, 512 copies of bitvector models are gathered and verified to be equal to optimized implementations. 512 coefficients in $\mathbb{F}_{2^{64}}$ are also collected to verify (5). Algebraic properties of embeddings are verified by property checking in vCRYPTOLINE.

5.2.3 Butterflies

The input polynomial $F(X) \in \mathbb{F}_2^{2^{15}}[X]$ is transformed to the polynomial $F_{9,0}(X) = \sum_{i=0}^{511} [S]_i f_i \in \mathbb{F}_{2^{64}}^{512}[X]$ after embedding. During butterflies, $F_{9,0}(X)$ is evaluated at 512 elements in $\mathbb{F}_{2^{64}} = \mathbb{F}_2[z]/\langle p(z) \rangle$ with $p(z) = z^{64} + z^4 + z^3 + z + 1$. Observe that if $F(X)$ is evaluated at $S_{k+1} = v_l$, then it is evaluated on $S_k = v_{l+1} + W_1$ at the next level of butterflies (Section 3). Since $F_{9,0}(X)$ is obtained by evaluating $F(X)$ at $S_9 = v_{33}$ in embedding, butterflies start by evaluating $F_{9,0}(X)$ on $S_8 = v_{34} + W_1$. After 9 levels of butterflies, Frobenius aFFTs evaluate $F_{9,0}(X)$ at $v_{42} + W_9$ with W_9 spanned by $\beta_9 = \{v_1, v_2, \dots, v_9\}$.²

Both AVX2 and NEON implementations compute butterflies by stages. All but the first stages perform two levels of butterflies for $F_{9,0}(X) \in \mathbb{F}_{2^{64}}^{512}[X]$. To explain verified properties, recall that $F_{9,0}(X)$ is in radix $S_8, S_7, \dots, S_0$ and $S_9 = v_{33}$. Hence the first level of butterflies evaluate on the space $S_8 + W_1 = v_{34} + W_1$. After substituting v_{34} and $v_{34} + 1$ for S_8 , we obtain two polynomials $F_{8,0}(X), F_{8,1}(X) \in \mathbb{F}_{2^{64}}^{256}[X]$ such that

$$\begin{aligned} F_{9,0}(X) &\equiv F_{8,0} \pmod{[S_8 + v_{34}, p(z), 2]} \\ &\equiv F_{8,1} \pmod{[S_8 + v_{34} + v_1, p(z), 2]}. \end{aligned}$$

The second stage consists of next two levels of butterflies. It evaluates $F_{9,0}(X)$ on the spaces $S_7 + W_2 = v_{35} + W_2$ and $S_6 + W_3 = v_{36} + W_3$. At the end of this stage, we obtain the following modular equations for $F_{6,i}(X) \in \mathbb{F}_{2^{64}}^{64}[X]$ where $0 \leq i < 8$:

$$\begin{aligned} F_{9,0}(X) &\equiv F_{6,0}(X) \pmod{[S_6 + v_{36}, p(z), 2]} && \equiv F_{6,1}(X) \pmod{[S_6 + v_{36} + v_1, p(z), 2]} \\ &\equiv F_{6,2}(X) \pmod{[S_6 + v_{36} + v_2, p(z), 2]} && \equiv F_{6,3}(X) \pmod{[S_6 + v_{36} + v_2 + v_1, p(z), 2]} \\ &\equiv F_{6,4}(X) \pmod{[S_6 + v_{36} + v_3, p(z), 2]} && \equiv F_{6,5}(X) \pmod{[S_6 + v_{36} + v_3 + v_1, p(z), 2]} \\ &\equiv F_{6,6}(X) \pmod{[S_6 + v_{36} + v_3 + v_2, p(z), 2]} \\ &\equiv F_{6,7}(X) \pmod{[S_6 + v_{36} + v_3 + v_2 + v_1, p(z), 2]} \end{aligned}$$

At the end of 5 stages of butterflies, we obtain $F_{0,0}, F_{0,1}, \dots, F_{0,511} \in \mathbb{F}_{2^{64}}$ that

$$\begin{aligned} F_{9,0}(X) &\equiv F_{0,0} \pmod{[X + v_{42}, p(z), 2]} \\ &\equiv F_{0,1} \pmod{[X + v_{42} + v_1, p(z), 2]} \\ &\equiv F_{0,2} \pmod{[X + v_{42} + v_2, p(z), 2]} \\ &\dots \\ &\equiv F_{0,511} \pmod{[X + v_{42} + v_9 + v_8 + \dots + v_1, p(z), 2]} \end{aligned}$$

² $v_9 = z^2 + z^3 + z^5 + z^6 + z^7 + z^9 + z^{12} + z^{13} + z^{17} + z^{19} + z^{20} + z^{21} + z^{22} + z^{24} + z^{25} + z^{28} + z^{29} + z^{31} + z^{32} + z^{33} + z^{35} + z^{38} + z^{39} + z^{43} + z^{44} + z^{45} + z^{47} + z^{48} + z^{49} + z^{50} + z^{55} + z^{62}$.

Our verification also proceeds by stages. At the end of each stage, vCRYPTOLINE verifies the modular equations corresponding to the stage. Particularly, all 512 modular equations for the last stage are verified by vCRYPTOLINE for butterflies.

Backward butterflies have a similar structure but in the reverse order. We are given 512 elements $G_{0,0}, G_{0,1}, \dots, G_{0,511} \in \mathbb{F}_{2^{64}}$ and a polynomial $G_{9,0}(X) \in \mathbb{F}_{2^{64}}^{512}[X]$ such that

$$\begin{aligned} G_{9,0}(X) &\equiv G_{0,0} \pmod{[X + v_{42}, p(z), 2]} \\ &\equiv G_{0,1} \pmod{[X + v_{42} + v_1, p(z), 2]} \\ &\equiv G_{0,2} \pmod{[X + v_{42} + v_2, p(z), 2]} \\ &\dots \\ &\equiv G_{0,511} \pmod{[X + v_{42} + v_9 + v_8 + \dots + v_1, p(z), 2]}. \end{aligned}$$

After the first backward stage consisting of two levels of backward butterflies, we have

$$\begin{aligned} G_{9,0}(X) &\equiv G_{2,0}(X) \pmod{[S_2 + v_{40}, p(z), 2]} \\ &\equiv G_{2,1}(X) \pmod{[S_2 + v_{40} + v_1, p(z), 2]} \\ &\equiv G_{2,2}(X) \pmod{[S_2 + v_{40} + v_2, p(z), 2]} \\ &\dots \\ &\equiv G_{2,127}(X) \pmod{[S_2 + v_{40} + v_7 + v_6 + \dots + v_1, p(z), 2]} \end{aligned}$$

where $G_{2,i}(X) \in \mathbb{F}_{2^{64}}^4[X]$ for $0 \leq i < 128$. Similarly, we obtain the following modular equations after 4 stages of backward butterflies:

$$\begin{aligned} G_{9,0}(X) &\equiv G_{8,0}(X) \pmod{[S_8 + v_{34}, p(z), 2]} \\ &\equiv G_{8,1}(X) \pmod{[S_8 + v_{34} + v_1, p(z), 2]} \end{aligned}$$

with $G_{8,0}(X), G_{8,1}(X) \in \mathbb{F}_{2^{64}}^{256}[X]$. The last level is again specialized and yields

$$G_{9,0}(X) \equiv G_{9,0}(X) \pmod{[S_9 + v_{33}, p(z), 2]} \quad (6)$$

where $G_{9,0} \in \mathbb{F}_{2^{64}}^{512}[X]$ is in radix $S_8, S_7, \dots, S_0$. Our verification for backward butterflies again proceeds by stages. For each stage, vCRYPTOLINE verifies the modular equations corresponding to the stage. Specifically, (6) is verified at the end of backward butterflies.

Evaluation of a polynomial from $\mathbb{F}_{2^{64}}[X]$ necessarily requires multiplication in $\mathbb{F}_{2^{64}}$. Recall that an element in $\mathbb{F}_{2^{64}}$ is a polynomial in $\mathbb{F}_2[z]/\langle p(z) \rangle$ with $p(z) = z^{64} + z^4 + z^3 + z + 1$. Bit polynomial instructions (`vpclmulhghqdq` in AVX2 and `pmull` in NEON) are employed by both implementations. Recall that products of two polynomials in $\mathbb{F}_2^{64}[z]$ are of the form $\sum_{i=0}^{126} b_i z^i$. They need to be reduced by $p(z)$ to obtain field elements. Observe

$$\begin{aligned} \sum_{i=0}^{126} b_i z^i &= \sum_{i=0}^{63} b_i z^i + z^{64} \sum_{i=0}^{62} b_{64+i} z^i \equiv \sum_{i=0}^{63} b_i z^i + (z^4 + z^3 + z + 1) \sum_{i=0}^{62} b_{64+i} z^i \\ &= \sum_{i=0}^{63} b_i z^i + \sum_{j=0}^{66} c_j z^j = \sum_{i=0}^{63} b_i z^i + \sum_{j=0}^{63} c_j z^j + z^{64} \sum_{j=0}^2 c_{64+j} z^j \\ &\equiv \sum_{i=0}^{63} b_i z^i + \sum_{j=0}^{63} c_j z^j + (z^4 + z^3 + z + 1) \sum_{j=0}^2 c_{64+j} z^j \pmod{p(z)}. \end{aligned}$$

Two polynomial multiplications with $z^4 + z^3 + z + 1$ can be used to compute a field multiplication. Note that the second multiplication $(z^4 + z^3 + z + 1) \sum_{j=0}^2 c_{64+j} z^j$ has only 8 possible outcomes since its multiplicand is $\sum_{j=0}^2 c_{64+j} z^j$. The AVX2 implementation parallelizes four of such multiplications by table lookups. We again construct two models and apply property and logic equivalence checking to verify field multiplications with vCRYPTOLINE. The NEON implementation does not have this optimization.

Table 1: Experimental Results

Name		Function	#Asm	#CL	#Prop	Time
mceliece [ABC ⁺ 22]	348864	radix_conversions	3.2	16.6	99	12
		vec256_mul_asm	0.3	2.2	256	60
		butterflies	23.9	268.7	37966	1508
	8192128	radix_conversions	4.8	8.5	167	312
		vec128_mul_asm	0.3	1.3	128	21
		vec256_maa_asm	0.4	5.7	256	91
		transpose_64x256	1.6	22.3	0	29
		butterflies	68.0	414.9	74432	4763
HQC-128 [CCPY26]	AVX2	varsub_x256	1.5	61.1	4	98
		bc_256	2.1	52.5	8	64
		bc_1_256	4.3	58.0	39	76
		encoder_64	6.5	15.9	6	6528+1030
		btfy_64	18.5	194.5	5560	2117
		ibtbfy_64	17.7	178.0	2745	5052
		decoder_64	6.5	32.3	8	2403+1811
		ibc_1_256	6.0	132.5	194	209
		ibc_256	4.6	218.8	20	158
		i_varsub_x256	3.4	150.1	73	264
	NEON	repr_s8_1	3.0	49.3	7	147
		bc_1_256	4.8	51.8	9	289
		bc_256	2.2	168.5	65	57
		encoder_64	9.8	188.1	6	6444+779
		btfy_64	19.1	369.0	3770	1642
		ibtbfy_64	18.8	461.1	2753	1555
		decoder_64	19.1	37.5	8	2447+1803
		ibc_256	6.2	608.6	20	126
		ibc_1_256	9.9	137.3	194	96
		irepr_s8_1	7.2	171.8	151	264

6 Evaluation

We use vCRYPTOLINE to verify Gao-Mateer aFFTs in the optimized AVX2 implementation for Classic McEliece provided by the design team [ABC⁺22], and the AVX2 and NEON implementations of Frobenius aFFTs and their inverse for HQC-128 from [CCPY26]. The computer algebra system SINGULAR 4.4.1 is invoked by vCRYPTOLINE to verify algebraic properties [DP13, GP07]. The ABC tool (commit 3ce53c361f601715) is also used by vCRYPTOLINE for logic equivalence checking [Mis]. Experiments are running on an Ubuntu 24.04.4 LTS Linux server with 4 2.45GHz AMD EPYC 7763 64-Core Processors and 2TB RAM. Table 1 gives the experimental results. The columns *Name* and *Function* show the package and function names. *#Asm* and *#CL* give the number of assembly instructions and lines of vCRYPTOLINE models in thousands respectively. *#Prop* shows the number of modular equations verified by vCRYPTOLINE. *Time* indicates verification time in seconds.

In its optimized AVX2 implementation, Classic McEliece uses Gao-Mateer aFFTs to find roots of polynomials with coefficients in fields of characteristic 2. With the parameter 348864, polynomials in $\mathbb{F}_{2^{12}}[X]$ are evaluated at 4096 elements in the field $\mathbb{F}_{2^{12}} = \mathbb{F}_2[z]/\langle z^{12} + z^3 + 1 \rangle$. For the parameter 8192128, polynomials in $\mathbb{F}_{2^{13}}[X]$ are evaluated at 8192 elements in the field $\mathbb{F}_{2^{13}} = \mathbb{F}_2[z]/\langle z^{13} + z^4 + z^2 + z + 1 \rangle$. The functions **radix_conversions** and **butterflies** perform radix conversion and butterflies (Section 5.1). Assembly functions **vec256_mul_asm**, **vec128_mul_asm**, **vec256_maa_asm**

compute field operations in parallel. The **transpose_64x256** function transposes $4 \times 64 \times 64$ bit matrices in parallel and is only used in large parameters.

The efficient HQC-128 implementation adopts Frobenius aFFTs for multiplications in the polynomial ring $\mathbb{F}_2^{2^{14}}[X]$. Since product polynomials are of degree $2^{15} - 2$, forward and backward Frobenius aFFTs from $\mathbb{F}_2^{2^{15}}[X]$ to 512 elements in $\mathbb{F}_{2^{64}}$ are implemented. Both AVX2 and NEON implementations are verified. The functions **varsub_x256** and **repr_s8_1** perform the first phase of radix conversion for AVX2 and NEON respectively. The functions **bc_256** and **bc_1_256** are the second and third phases respectively (Section 5.2.1). The embedding of 64-coefficient bit polynomials to $\mathbb{F}_{2^{64}}$ and its inverse are implemented in **encoder_64** and **decoder_64** respectively (Section 5.2.2). The forward butterflies from $\mathbb{F}_{2^{64}}^{512}[X]$ to 512 elements in $\mathbb{F}_{2^{64}}$ are computed in the function **btfy_64**. Functions prefixed by **i** such as **ibtbfy_64** are counterparts in backward aFFTs.

From the table, one notices the sizes of vCRYPTOLINE models are significantly larger than assembly implementations. This is mainly due to algebraic predicates in assertions. For instance, 2^{12} field elements are obtained after Classic McEliece 348864 butterflies. Each field element is a polynomial in z . 4096 modular equations are needed to specify only 1 stage of butterflies. They contribute to ≈ 20000 lines in our vCRYPTOLINE model. Since specifications are necessary in verification, sizes of vCRYPTOLINE models are unavoidably larger than assembly codes. Fortunately, these specifications are easily generated by scripts. We did not write down thousands of modular equations by hand.

Number of modular equations in different functions vary significantly. As mentioned, aFFTs naturally have thousands of modular equations to be verified. Additional modular equations are needed to verify field multiplications since they are in fact polynomial multiplications. Thanks to the computer algebra system SINGULAR, vCRYPTOLINE verifies tens of thousands of modular equations within reasonable time automatically.

Radix conversion has much less modular equations to be verified. It consequently requires little time for verification. For all implementations from Classic McEliece and HQC-128 aFFTs, every function for radix conversion is verified within six minutes. Butterflies in both aFFTs have thousands of modular equations in assertions and naturally require much time. Classic McEliece has about 38 and 74 thousands modular equations for parameters **348864** and **8192128** respectively. Its verification takes about 0.4 and 1.3 hours respectively. For Frobenius aFFTs, forward and backward butterflies on AVX2 and NEON have thousands of modular equations and take from 0.4 to 1.4 hours to verify respectively.

AVX2 implementations of Classic McEliece provide several parallel field arithmetic functions. Functions **vec256_mul_asm** and **vec128_mul_asm** compute 256 and 128 $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{13}}$ products in parallel respectively. **vec256_maa_asm** performs 256 multiplication-and-sum in $\mathbb{F}_{2^{13}}$. The function **transpose_64x256** performs $4 \times 64 \times 64$ bit matrix transpositions simultaneously, and has no modular equation but $4 \times 64 \times 64$ equations to verify. Each of them is verified in 2 minutes. HQC-128 aFFTs additionally embed polynomials from $\mathbb{F}_2^{2^{15}}[X]$ into $\mathbb{F}_{2^{64}}^{512}[X]$ (Section 5.2.2). 512 embeddings and their inverses are verified by property checking from 0.7 to 1.8 hours respectively. Complete bitvector models for 512 embeddings and their inverses are also verified against their algebraic counterparts with logic equivalence checking from 0.2 and 0.5 hours respectively.

References

- [AAB⁺22] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert,

- and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions>.
- [ABB⁺25] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Lionel Blatter, Gustavo Xavier Delerue Marinho Alves, João Diogo Duarte, Benjamin Grégoire, Tiago Oliveira, Miguel Quaresma, Pierre-Yves Strub, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Jazzline: Composable CryptoLine functional correctness proofs for jasmin programs. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025*, pages 1409–1423, Taipei, Taiwan, October 13–17, 2025. ACM Press.
- [ABC⁺22] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions>.
- [AMBD⁺18] Carlos Aguilar-Melchor, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Efficient encryption from random quasi-cyclic codes. *IEEE Transactions on Information Theory*, 64(5):3927–3943, 2018.
- [AOB⁺24] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Lécenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying Kyber - episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part II*, volume 14921 of *LNCS*, pages 384–421, Santa Barbara, CA, USA, August 18–22, 2024. Springer, Cham, Switzerland.
- [BBB⁺21] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy*, pages 777–795, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.
- [BCH⁺26] Daniel J. Bernstein, Han-Ting Chen, John R. Harrison, Cesare Huang, Gregory Maxwell, Bow-Yaw Wang, Pieter Wuille, and Bo-Yin Yang. Accelerating and verifying constant-time modular inversion. In Joan Daemen and Picture Emmanuel Thomé, editors, *EUROCRYPT 2026, Part VII*, volume 16547 of *Lecture Notes in Computer Science*, pages 598–627. Springer, Cham, May 2026.
- [BCPR23] Martin Brain, Carlos Cid, Rachel Player, and Wrenna Robson. Verifying classic McEliece: examining the role of formal methods in post-quantum cryptography standardisation. Cryptology ePrint Archive, Report 2023/010, 2023.
- [BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 250–272, Santa Barbara, CA, USA, August 20–23, 2013. Springer Berlin Heidelberg, Germany.

- [BHK⁺25] Karthikeyan Bhargavan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. Formal security and functional verification of cryptographic protocol implementations in rust. In Chun-Ying Huang, Jyh-Cheng Chen, Shih-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025*, pages 2729–2743, Taipei, Taiwan, October 13–17, 2025. ACM Press.
- [CCPY26] Ming-Shing Chen, Chun-Ming Chiu, Chun-Tao Peng, and Bo-Ying Yang. Accelerating HQC with additive FFT. *IACR TCHES*, 2026(2):520–544, 2026.
- [Cho18] Tung Chou. McBits revisited: toward a fast constant-time code-based KEM. *Journal of Cryptographic Engineering*, 8(2):95–107, June 2018.
- [CLT⁺25] Chun-Ming Chiu, Jiaxiang Liu, Ming-Hsien Tsai, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang. Algebraic linear analysis for number theoretic transform in lattice-based cryptography. *IACR TCHES*, 2025(3):668–692, 2025.
- [CYW25] Chun-Ming Chiu, Bo-Yin Yang, and Bow-Yaw Wang. A new trick for polynomial multiplication A verified CRT polymul utilizing a monomial factor. *IACR TCHES*, 2025(4):795–816, 2025.
- [DP13] Wolfram Decker and Gerhard Pfister. *A First Course in Computational Algebraic Geometry*. Cambridge University Press, 2013.
- [EPG⁺19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy*, pages 1202–1219, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.
- [FLS⁺19] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed CryptoLine. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1591–1606, London, UK, November 11–15, 2019. ACM Press.
- [fS26] The International Organization for Standardization. Information technology — security techniques — encryption algorithms — part 2: Asymmetric ciphers — amendment 2, June 2026.
- [GM10] Shuhong Gao and Todd Mateer. Additive fast fourier transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, 2010.
- [GP07] Gert-Martin Greuel and Gerhard Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, 2nd edition, 2007.
- [HLS⁺22] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT multiplications for NIST-PQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR TCHES*, 2022(4):718–750, 2022.
- [Hun12] Thomas W. Hungerford. *Algebra*, volume 73 of *Graduate Texts in Mathematics*. Springer New York, NY, 2012.
- [KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops*,

- Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.
- [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.
 - [LLS⁺24] Li-Chang Lai, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Automatic verification of cryptographic block function implementations with logical equivalence checking. In Joaquin Garcia-Alfaro, Rafał Kozik, Michał Choraś, and Sokratis Katsikas, editors, *ESORICS 2024, Part IV*, volume 14985 of *LNCS*, pages 377–395, Bydgoszcz, Poland, September 16–20, 2024. Springer, Cham, Switzerland.
 - [McE78] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. Technical report, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA, 1978. DSN Progress Report.
 - [Mis] Alan Mishchenko. ABC: System for sequential logic synthesis and formal verification.
 - [Mul54] David E. Muller. Application of Boolean algebra to switching circuit design and to error detection. *Transactions of the I.R.E. Professional Group on Electronic Computers*, EC-3(3):6–12, 1954.
 - [Nie86] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):157—166, 1986.
 - [oST25] The United States National Institute of Standards and Technology. NIST selects HQC as fifth algorithm for post-quantum encryption, March 2025.
 - [Ree54] Irving S. Reed. A class of multiple-error-correcting codes and the decoding scheme. *Transactions of the IRE Professional Group on Information Theory*, 4(4):38–49, 1954.
 - [RS60] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
 - [TFL⁺23] Ming-Hsien Tsai, Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification for algebraic abstraction. In C. Enea and A. Lal, editors, *Computer Aided Verification (CAV)*, volume 13966 of *Lecture Notes in Computer Science*, pages 329–349. Springer, Cham, 2023.
 - [vdHL17] Joris van der Hoeven and Robin Larrieu. The Frobenius FFT. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation, ISSAC ’17*, pages 437—444, New York, NY, USA, 2017. Association for Computing Machinery.
 - [ZHZ⁺24] Jipeng Zhang, Junhao Huang, Lirui Zhao, Donglong Chen, and Çetin Kaya Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*, Philadelphia, PA, USA, August 14–16, 2024. USENIX Association.

Temporary page!

L^AT_EX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because L^AT_EX now knows how many pages to expect for this document.