Type-II Optimal Polynomial Bases

Daniel J. Bernstein[1] and Tanja Lange[2]

[1] Department of Computer Science (MC 152)
University of Illinois at Chicago, Chicago, IL 60607–7053, USA
djb@cr.yp.to

[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
tanja@hyperelliptic.org

Abstract. In the 1990s and early 2000s several papers investigated the relative merits of polynomial-basis and normal-basis computations for $\mathbf{F}_{2^n}$. Even for particularly squaring-friendly applications, such as implementations of Koblitz curves, normal bases fell behind in performance unless a type-I normal basis existed for $\mathbf{F}_{2^n}$.

In 2007 Shokrollahi proposed a new method of multiplying in a type-II normal basis. Shokrollahi's method efficiently transforms the normal-basis multiplication into a single multiplication of two size-$(n+1)$ polynomials.

This paper speeds up Shokrollahi's method in several ways. It first presents a simpler algorithm that uses only size-n polynomials. It then explains how to reduce the transformation cost by dynamically switching to a 'type-II optimal polynomial basis' and by using a new reduction strategy for multiplications that produce output in type-II polynomial basis.

As an illustration of its improvements, this paper explains in detail how the multiplication overhead in Shokrollahi's original method has been reduced by a factor of 1.4 in a major cryptanalytic computation, the ongoing attack on the ECC2K-130 Certicom challenge. The resulting overhead is also considerably smaller than the overhead in a traditional low-weight-polynomial-basis approach. This is the first state-of-the-art binary-elliptic-curve computation in which type-II bases have been shown to outperform traditional low-weight polynomial bases.

Keywords: Optimal normal basis, ONB, polynomial basis, transformation, elliptic-curve cryptography.

1 Introduction

If $n+1$ is prime and 2 has order n modulo $n+1$ then the field $\mathbf{F}_{2^n} = \mathbf{F}_2[\zeta]/(\zeta^n + \cdots + \zeta + 1)$ has a "type-I optimal normal basis" $\zeta, \zeta^2, \zeta^4, \ldots$. It has been known for many years that this basis allows not only fast repeated squarings but also surprisingly fast multiplications, costing only $M(n) + 2n - 2$ bit operations where $M(n)$ is the minimum cost of multiplying n-coefficient polynomials. The idea is to permute the basis into $\zeta, \zeta^2, \zeta^3, \ldots, \zeta^n$, and to decompose multiplication into the following operations:

- $M(n)$ bit operations: multiply the polynomials $f_1\zeta + \cdots + f_n\zeta^n$ and $g_1\zeta + \cdots + g_n\zeta^n$ in $\mathbf{F}_2[\zeta]$.
- $n-2$ bit operations: eliminate the coefficients of $\zeta^{n+2}, \ldots, \zeta^{2n}$ using the identities $\zeta^{n+2} = \zeta$, $\ldots$, $\zeta^{2n} = \zeta^{n-1}$; this requires additions to the existing coefficients of $\zeta^2, \ldots, \zeta^{n-1}$.
- n bit operations: eliminate the coefficient of ζ^{n+1} using the identity $\zeta^{n+1} = \zeta + \zeta^2 + \cdots + \zeta^n$.

[*] Permanent ID of this document: 90995f3542ee40458366015df5f2b9de. Date of this document: 2010.02.09. This work has been supported in part by the European Commission through the ICT Programme under Contract ICT–2007–216676 ECRYPT-II and in part by the National Science Foundation under grant ITR–0716498.

An alternative introduced in [IT89] is to use a redundant representation, specifically coefficients of $1, \zeta, \ldots, \zeta^n$, with arithmetic modulo $\zeta^{n+1}+1$. Multiplication then costs $M(n+1)+n$ bit operations; this is worse than $M(n)+2n-2$ for small n, but since $M(n)$ is subquadratic it becomes better for large n.

However, most integers n do not have type-I optimal normal bases. In particular, an odd prime n cannot have a type-I optimal normal basis. This poses severe problems for cryptographic applications that, for security reasons, prohibit composite values of n.

The conventional wisdom for many years was that type-I normal bases were a unique exception. For all other normal bases the best multiplication methods in the literature were quite slow, asymptotically at least twice the cost of multiplication in traditional low-weight polynomial bases (trinomial bases and pentanomial bases). Normal bases were competitive only in extreme situations: applications where n was very small; applications having many repeated squarings and very few multiplications; and applications that imposed extremely small hardware-area requirements, effectively punishing polynomial bases by prohibiting fast-multiplication techniques.

The picture changed a few years ago when Shokrollahi introduced a new multiplier using only $M(n)+O(n\log_2 n)$ operations for a "type-II optimal normal basis" of $\mathbf{F}_{2^n}$. See Shokrollahi's thesis [Sho07, Chapter 4] and the subsequent WAIFI 2007 publication [vzGSS07] by von zur Gathen, Shokrollahi, and Shokrollahi. This new multiplier makes type-II normal bases competitive with traditional low-weight polynomial bases for a much wider variety of applications. The overhead term $O(n\log_2 n)$ is not quite as small as the $O(n)$ cost of low-weight polynomial reduction, but this difference is quite often outweighed by the benefit of fast repeated squarings.

In this paper we reduce the overhead in Shokrollahi's method in several ways. The overall reduction depends on the pattern of desired squarings and multiplications but can be as much as a factor of 2. We give a real-world example in which the overhead is reduced by more than a factor of 1.4.

1.1. Model of computation. All of the algorithms in this paper are straight-line (branchless) sequences of bit operations. The "bit operations" allowed are two-input XORs (addition in $\mathbf{F}_2$) and two-input ANDs (multiplication in $\mathbf{F}_2$). We measure algorithm cost by counting the number of bit operations, as in [Sho07], [Ber09a], etc.

Optimizing bit operations is not the same as optimizing hardware area: a very small circuit can carry out many bit operations if the operations to be performed are sufficiently regular. Optimizing bit operations is also not the same as optimizing hardware latency. However, optimizing bit operations is very close to optimizing the *throughput* of pipelined hardware. We predict that the speedups discussed in this paper will turn out to be useful in applications that need to maximize the number of $\mathbf{F}_{2^n}$ operations that can be carried out per second for a given chip area.

This work began as part of a larger project described in [BBB$^+$09] to solve a cryptanalytic challenge, the Certicom "ECC2K-130" challenge [Cer97]. One of the surprises in [BBB$^+$09] is that type-II bases save time for the ECC2K-130 computation on various *software* platforms, solidly outperforming low-weight (in this case pentanomial) polynomial bases. At the time of this writing, the latest version of [BBB$^+$09] reports the speed of software that uses Shokrollahi's approach together with the improvements described in Section 3 of this paper; the latest software incorporates additional improvements described in subsequent sections of this paper.

Optimizing bit operations is often believed to be even farther from optimizing software than it is from optimizing hardware. However, [Ber09a] recently set new software speed records for public-key cryptography by exploiting a synergy between "bitsliced" data structures, bit-operation-optimized polynomial-multiplication techniques, and the 128-bit vector operations available on common CPUs; [BBB$^+$09] extended this synergy to include type-II bases. We are now investigating the constructive use of the same techniques for fast Koblitz-curve cryptography — of course, at much larger sizes than ECC2K-130!

1.2. Outline of the paper. Section 2 reviews Shokrollahi's algorithm for type-II normal-basis multiplication. Section 3 presents a streamlined algorithm for type-II normal-basis multiplication. The streamlined algorithm uses approximately $M(n)+2n\log_2(n/2)$ bit operations. More precisely, if $n = 2^{n_0} + 2^{n_1} + \cdots$ with $n_0 > n_1 > \cdots$, then the streamlined algorithm uses $M(n) + \sum_i (2^{n_i}(2n_i - 2 + 4i) + 3)$ bit operations.

Section 4 presents an algorithm to multiply in a "type-II polynomial basis" using approximately $M(n) + n\log_2 n$ bit operations. The overhead $n\log_2 n$ saves almost half of the previous overhead $2n\log_2(n/2)$; about $0.5n\log_2(n/4)$ is saved by a new reduction method, and about $0.5n\log_2(n/4)$ is saved by a simple exercise in caching. Repeated squaring in a "type-II polynomial basis" is not as fast as repeated squaring in a normal basis but is still much faster than repeated squaring in a traditional polynomial basis.

We further reduce the total overhead of multiplications and repeated squarings by dynamically mixing a type-II polynomial basis P with a type-II normal basis N. In applications that contain only occasional multiplications, this mixture is tantamount to working purely in N, as in Section 3. In applications that contain only occasional repeated squarings, this mixture is tantamount to working purely in P. However, in many applications the mixture is better than any pure method. Section 5 uses ECC2K-130 as an illustrative example of this paper's overall improvements.

2 Review of the Original Shokrollahi Approach

In this section we review the normal-basis multiplier from [Sho07, Section 4] for the special case of binary fields $\mathbf{F}_{2^n}$.

In a nutshell the main achievement of [Sho07] is a map S between a normal-basis representation and a special polynomial-basis representation, taking $\Theta(n\log_2 n)$ bit operations instead of the usual $\Theta(n^2)$. The multiplication in normal basis can be performed as $S^{-1}(S(a_1)\cdot S(a_2))$: first use S on a_1 and a_2 to map to polynomial-basis representation, where efficient algorithms for polynomial multiplication can be used, and finally map the result back to normal basis representation.

The irreducible polynomial for this special polynomial basis is usually rather dense. To avoid reduction modulo this polynomial, Shokrollahi defines a double-length map that takes an unreduced polynomial product back to normal basis. The reduction is done on the normal-basis side, where it is simple addition of length-n vectors.

Before giving the details we review the construction of a type-II normal basis for $\mathbf{F}_{2^n}$ and establish notations P, N used throughout the paper. For more information on type-II normal bases, see Mullin et al. [MOVW89].

2.1. Type-II normal bases. For the rest of the paper we assume that n is a positive integer, that $2n+1$ is prime, and that either (1) the order of 2 modulo $2n+1$ is $2n$ or (2) the order of 2 modulo $2n+1$ is n and $2n+1 \equiv 3 \pmod 4$. In the latter case 2 generates the subgroup of squares modulo $2n+1$. The congruence $2n+1 \equiv 3 \pmod 4$ implies that -1 is not a square, so -1 is not a power of 2.

The conditions on n imply that $2^{2n} \equiv 1 \pmod{2n+1}$ and so there exists an element $\zeta \in \mathbf{F}_{2^{2n}}$ that is a primitive $(2n+1)$st root of unity. Now $2^n \equiv \pm 1 \pmod{2n+1}$, since 2 has order $2n$ or n, so $\zeta^{2^n} \in \{\zeta, \zeta^{-1}\}$. Define $c = \zeta + \zeta^{-1}$. Then $c^{2^n} = \zeta^{2^n} + \zeta^{-2^n} = \zeta + \zeta^{-1} = c$, so $c \in \mathbf{F}_{2^n}$.

The elements $c, c^2, c^{2^2}, c^{2^3}, \ldots, c^{2^{n-1}}$ are distinct. Indeed, any repetition would (after some square roots) imply an equation of the form $c^{2^i} = c$ with $i \in \{1, 2, \ldots, n-1\}$, i.e., $(\zeta+\zeta^{-1})^{2^i} = \zeta + \zeta^{-1}$, i.e., $\zeta^{2^i} + \zeta^{-2^i} + \zeta + \zeta^{-1} = 0$. This equation factors as $(\zeta^{2^i} + \zeta)(1 + \zeta^{-2^i-1}) = 0$, so $\zeta^{2^i} = \zeta$ or $\zeta^{2^i+1} = 1$, so $2^i \equiv \pm 1 \pmod{2n+1}$. This implies $2^{2i} \equiv 1 \pmod{2n+1}$ which contradicts that the order of 2 is $2n$ modulo $(2n+1)$. If the order of 2 is n and $2n+1 \equiv 3 \pmod 4$ then -1 is not a power of 2 contradicting $2^i \equiv -1 \pmod{2n+1}$ while $2^i \equiv 1 \pmod{2n+1}$ contradicts that the order is n.

Each power c^{2^i} can be written as $\zeta^j + \zeta^{-j}$ for the unique $j \in \{1, 2, \ldots, n\}$ with $2^i \equiv \pm j \pmod{2n+1}$, since $c^{2^i} = (\zeta + \zeta^{-1})^{2^i} = \zeta^{2^i} + \zeta^{-2^i} = \zeta^j + \zeta^{-j}$. Therefore $(c, c^2, c^{2^2}, \ldots, c^{2^{n-1}})$ is a permutation of $(\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^n + \zeta^{-n})$.

If a vector $(e_1, e_2, \ldots, e_n) \in \mathbf{F}_2^n$ satisfies $e_1(\zeta + \zeta^{-1}) + e_2(\zeta^2 + \zeta^{-2}) + \cdots + e_n(\zeta^n + \zeta^{-n}) = 0$ then $e_1(\zeta + \zeta^{2n}) + e_2(\zeta^2 + \zeta^{2n-1}) + \cdots + e_n(\zeta^n + \zeta^{n+1}) = 0$, so ζ is a root of the polynomial $p = e_1(1 + z^{2n-1}) + \cdots + e_n(z^{n-1} + z^n) \in \mathbf{F}_2[z]$ of degree at most $2n-1$. Exchanging ζ and ζ^{-1} shows that ζ^{-1} is also a root of p. If 2 has order $2n$ then the cyclotomic polynomial Φ_{2n+1} is irreducible in $\mathbf{F}_2[z]$, so Φ_{2n+1} divides p. If 2 has order n and $2n+1 \equiv 3 \pmod 4$ then Φ_{2n+1} factors into the coprime irreducible polynomials $(z-\zeta)(z-\zeta^2)\cdots(z-\zeta^{2^{n-1}})$ and $(z-\zeta^{-1})(z-\zeta^{-2})\cdots(z-\zeta^{-2^{n-1}})$; each of these polynomials divides p, so again Φ_{2n+1} divides p. Since $\deg(\Phi_{2n+1}) = 2n$ this implies $p = 0$ so $(e_1, e_2, \ldots, e_n) = 0$.

Summary: $(\zeta+\zeta^{-1}, \zeta^2+\zeta^{-2}, \zeta^3+\zeta^{-3}, \ldots, \zeta^n+\zeta^{-n})$ is a basis of $\mathbf{F}_{2^n}$, and $(c, c^2, c^{2^2}, \ldots, c^{2^{n-1}})$ is a normal basis of $\mathbf{F}_{2^n}$. This normal basis is called a "type-II optimal normal basis", and the permutation $(\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^n + \zeta^{-n})$ is called a "permuted type-II optimal normal basis".

2.2. The functions N and P. We denote by $N(x)$ the representation of $x \in \mathbf{F}_{2^n}$ with respect to the permuted normal basis $(\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^n + \zeta^{-n})$. In other words, the vector $N(x) = (N(x)_1, \ldots, N(x)_n) \in \mathbf{F}_2^n$ satisfies $\sum_i N(x)_i(\zeta^i + \zeta^{-i}) = x$.

We denote by $P(x)$ the representation of $x \in \mathbf{F}_{2^n}$ with respect to the polynomial basis $(c, c^2, c^3, \ldots, c^n)$. In other words, the vector $P(x) = (P(x)_1, \ldots, P(x)_n) \in \mathbf{F}_2^n$ satisfies $\sum_i P(x)_i(\zeta + \zeta^{-1})^i = x$. Note that this is not exactly a conventional polynomial basis: the corresponding polynomials have degree $\le n$ and constant term zero.

2.3. Shokrollahi's transformation. Shokrollahi extends the normal basis to the redundant generating set $\mathcal{N} = (1, \zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^n + \zeta^{-n})$ and extends the polynomial basis to the redundant generating set $\mathcal{P} = (1, (\zeta + \zeta^{-1}), (\zeta + \zeta^{-1})^2, (\zeta + \zeta^{-1})^3, \ldots, (\zeta + \zeta^{-1})^{n-1}, (\zeta+\zeta^{-1})^n)$. Shokrollahi recursively defines a transformation S_k from $(1, \zeta+\zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^{k-1} + \zeta^{1-k})$ to $(1, (\zeta + \zeta^{-1}), (\zeta + \zeta^{-1})^2, (\zeta + \zeta^{-1})^3, \ldots, (\zeta + \zeta^{-1})^{k-1}$, and in particular defines a transformation S_{n+1} from $\mathcal{N}$ to $\mathcal{P}$.

We first describe S_8^{-1} and then generalize. The central observation is that if

$$f_0+f_1\left(\zeta+\zeta^{-1}\right)+f_2\left(\zeta+\zeta^{-1}\right)^2+f_3\left(\zeta+\zeta^{-1}\right)^3 = g_0+g_1\left(\zeta+\zeta^{-1}\right)+g_2\left(\zeta^2+\zeta^{-2}\right)+g_3\left(\zeta^3+\zeta^{-3}\right)$$

and

$$f_4+f_5\left(\zeta+\zeta^{-1}\right)+f_6\left(\zeta+\zeta^{-1}\right)^2+f_7\left(\zeta+\zeta^{-1}\right)^3 = g_4+g_5\left(\zeta+\zeta^{-1}\right)+g_6\left(\zeta^2+\zeta^{-2}\right)+g_7\left(\zeta^3+\zeta^{-3}\right)$$

then

$$\begin{aligned} &f_0 + f_1\left(\zeta+\zeta^{-1}\right) + f_2\left(\zeta+\zeta^{-1}\right)^2 + f_3\left(\zeta+\zeta^{-1}\right)^3 \\ &\qquad + f_4\left(\zeta+\zeta^{-1}\right)^4 + f_5\left(\zeta+\zeta^{-1}\right)^5 + f_6\left(\zeta+\zeta^{-1}\right)^6 + f_7\left(\zeta+\zeta^{-1}\right)^7 \\ &\quad = g_0 + (g_1+g_7)\left(\zeta+\zeta^{-1}\right) + (g_2+g_6)\left(\zeta^2+\zeta^{-2}\right) + (g_3+g_5)\left(\zeta^3+\zeta^{-3}\right) \\ &\qquad\quad + g_4\left(\zeta^4+\zeta^{-4}\right) + g_5\left(\zeta^5+\zeta^{-5}\right) + g_6\left(\zeta^6+\zeta^{-6}\right) + g_7\left(\zeta^7+\zeta^{-7}\right). \end{aligned}$$

Converting from coefficients of $1, \zeta+\zeta^{-1}, (\zeta+\zeta^{-1})^2, \dots, (\zeta+\zeta^{-1})^7$ to coefficients of $1, \zeta+\zeta^{-1}, \zeta^2+\zeta^{-2}, \zeta^3+\zeta^{-3}, \dots, \zeta^7+\zeta^{-7}$ can thus be done with two half-size conversions and three additions of bits: first convert f_0, f_1, f_2, f_3 to g_0, g_1, g_2, g_3; separately convert f_4, f_5, f_6, f_7 to g_4, g_5, g_6, g_7; and then add g_7 to g_1, g_6 to g_2, and g_5 to g_3. The inverse, converting from coefficients of $1, \zeta+\zeta^{-1}, \zeta^2+\zeta^{-2}, \zeta^3+\zeta^{-3}, \dots, \zeta^7+\zeta^{-7}$ to coefficients of $1, \zeta+\zeta^{-1}, (\zeta+\zeta^{-1})^2, \dots, (\zeta+\zeta^{-1})^7$, has exactly the same cost. These conversions are extremely efficient.

More generally, instead of splitting 8 into $(4,4)$, one can (and should) split k into $(j, k-j)$, where j is the unique power of 2 satisfying $j+1 \le k \le 2j$. This is exactly what Shokrollahi's transformations S_k and S_k^{-1} do.

2.4. Shokrollahi's multiplication algorithm. Shokrollahi expands $N(a)$ and $N(b)$ by inserting a leading 0, obtaining linear combinations of $\mathcal{N}$, and then uses the transformation S_{n+1} to obtain linear combinations of $\mathcal{P}$, which are then interpreted as polynomials of degree at most n. Multiplying these two size-$(n+1)$ polynomials takes $M(n+1)$ bit operations and produces a polynomial of degree at most $2n$. Shokrollahi uses the transformation S_{2n+1}^{-1} to obtain a linear combination of $(1, \zeta+\zeta^{-1}, \zeta^2+\zeta^{-2}, \zeta^3+\zeta^{-3}, \dots, \zeta^{2n}+\zeta^{-2n})$, uses $\zeta^{n+i}+\zeta^{-(n+i)} = \zeta^{n+i-2n-1}+\zeta^{2n+1-(n+i)} = \zeta^{-(n+1-i)}+\zeta^{n+1-i}$ for $1 \le i \le n$ to reduce the intermediate result back to $\mathcal{N}$, and finally discards the coefficient of 1 (which is always 0 by [Sho07, Theorem 31]), obtaining $N(ab)$.

2.5. Shokrollahi's analysis. Shokrollahi shows in [Sho07, Lemma 21] that the cost for a size-2^r transformation is $\eta(r) = 2^{r-1}(r-2)+1$. He then computes the following upper bound on the cost of his multiplication algorithm:

- two conversions from $\mathcal{N}$ to $\mathcal{P}$, costing $\eta(\lceil\log_2(n+1)\rceil)$ each; plus
- a multiplication of polynomials of degree $\le n$, costing $M(n+1)$; plus
- one double-length conversion of a polynomial of degree $\le 2n$, costing $\eta(\lceil\log_2(2n+1)\rceil)$; plus
- n final additions.

See [Sho07] and [vzGSS07, Theorem 8, first display].

We point out that Shokrollahi's bounds are much higher than the actual costs of his algorithm, often losing a factor of 2 or more outside the $M(n+1)$ term. See Section 5 of this paper for an example. The "$+16n\log_2 n$" appearing in [vzGSS07, Section 1], and in more generality in [vzGSS07, Theorem 8, second display], is even more misleading. Those bounds should be disregarded by readers evaluating the performance of normal-basis arithmetic. We present quickly computable formulas for exact operation counts of our algorithms, along with reasonably precise approximations such as $(n/2)\log_2(n/4)$.

3 Streamlined Multiplication in Type-II Normal Basis

This section presents a simpler, smaller, slightly faster algorithm to compute $N(a), N(b) \mapsto N(ab)$. This algorithm is a convenient starting point for the larger speedups discussed in subsequent sections, so we present the algorithm from scratch, but we begin by summarizing the most important differences between the algorithm and Shokrollahi's algorithm.

3.1. Summary of the simplification. Recall that Shokrollahi's original algorithm extends the basis $\zeta + \zeta^{-1}, \dots, \zeta^n + \zeta^{-n}$ to $1, \zeta + \zeta^{-1}, \dots, \zeta^n + \zeta^{-n}$. Note that $1 \neq \zeta^0 + \zeta^{-0}$; evidently 1 plays a special role here.

The algorithm in this section shifts the underlying transformation by one position, avoiding the need to extend the original basis. The new transformation works with only n elements rather than $n+1$, and feeds the multiplier polynomials of size n rather than $n+1$.

3.2. Summary of the speedup. This multiplication algorithm has overhead approximately $2n \log_2(n/2)$: i.e., it uses approximately $M(n) + 2n \log_2(n/2)$ bit operations. It saves $M(n+1) - M(n)$ bit operations compared to Shokrollahi's original algorithm (according to our analysis of Shokrollahi's algorithm; Shokrollahi's analysis produces a much larger upper bound, as discussed in Section 2.5).

The differences $M(1)-M(0), M(2)-M(1), \dots, M(n)-M(n-1)$ have sum $M(n)-M(0) = M(n) \in O(n \log_2 n \log_2 \log_2 n)$, so the average difference is bounded by $O(\log_2 n \log_2 \log_2 n)$, which is *asymptotically* not nearly as large as $2n \log_2 n$. However, for *typical* values of n there is a quite significant difference between the best known upper bound on $M(n+1)$ and the best known upper bound on $M(n)$. For example, these differences for $n = 53$, $n = 83$, $n = 89$, $n = 113$, $n = 131$, and $n = 173$ are 67, 121, 73, 81, 154, and 108 respectively.

This section's algorithm makes structurally clear that the polynomials to be multiplied have size only n. An alternate, more complicated, way to save $M(n+1) - M(n)$ is as follows: observe that the coefficient of 1 inside Shokrollahi's algorithm is initialized to 0 and is never modified; conclude that the size-$(n+1)$ polynomials in the algorithm always have constant coefficient 0; speed up the algorithm accordingly. The intermediate conclusion appeared (with a different proof) in [Sho07, Theorem 31, proof, third sentence], but was not exploited in the algorithm.

3.3. The transformation. For each $k \geq 1$, each vector $e \in \mathbf{F}_2^k$, and each $i \in \{1, 2, \dots, k\}$, define e_i as the ith component of e. Then $e = (e_1, e_2, \dots, e_k)$. To support infinite sums over i, as in [Knu97a], we also allow "out-of-range" indices: define $e_i = 0$ for $i \in \mathbf{Z} \setminus \{1, 2, \dots, k\}$. We also use the notation $[i \neq 0]$ to mean 0 if $i = 0$ and 1 if $i \neq 0$.

For each $k \geq 1$ we define an invertible function $T_k : \mathbf{F}_2^k \to \mathbf{F}_2^k$ by the following recursion:

- Define $T_1(e) = e$.
- For $k \geq 2$: Define j as the largest power of 2 in $\{1, 2, \dots, k-1\}$. For each $f \in \mathbf{F}_2^j$ and each $g \in \mathbf{F}_2^{k-j}$ define $T_k(f, g) = (T_j(h), T_{k-j}(g))$ where $h_i = f_i + [i \neq 0] g_{j-i}$.

To recover f, g from $T_k(f, g) = (T_j(h), T_{k-j}(g))$, first invert T_j and T_{k-j} to obtain h and g, and then compute f from $f_i = h_i + [i \neq 0] g_{j-i}$.

For example:

- $T_2(e_1, e_2) = (e_1, e_2)$. Here $j = 1$, $f = (e_1)$, $g = (e_2)$, and $h = (e_1)$.
- $T_3(e_1, e_2, e_3) = (e_1 + e_3, e_2, e_3)$. Here $j = 2$, $f = (e_1, e_2)$, $g = (e_3)$, and $h = (e_1 + e_3, e_2)$.
- $T_4(e_1, e_2, e_3, e_4) = (e_1 + e_3, e_2, e_3, e_4)$. Here $j = 2$, $f = (e_1, e_2)$, $g = (e_3, e_4)$, and $h = (e_1 + e_3, e_2)$.
- $T_5(e_1, e_2, e_3, e_4, e_5) = (e_1 + e_3 + e_5, e_2, e_3 + e_5, e_4, e_5)$. Here $j = 4$, $f = (e_1, e_2, e_3, e_4)$, $g = (e_5)$, and $h = (e_1, e_2, e_3 + e_5, e_4)$.

- $T_6(e_1, e_2, e_3, e_4, e_5, e_6) = (e_1{+}e_3{+}e_5, e_2{+}e_6, e_3{+}e_5, e_4, e_5, e_6)$. Here $j = 4$, $f = (e_1, e_2, e_3, e_4)$, $g = (e_5, e_6)$, and $h = (e_1, e_2 + e_6, e_3 + e_5, e_4)$.

One can visualize the computation of h as folding g onto f in reverse order, but skipping the highest coefficient of f, and skipping the highest coefficient of g if g is as long as f.

Theorem 3.4. *Let k be a positive integer. Let ζ be an element of a field of characteristic 2. Then $\sum_i (T_k(e))_i(\zeta + \zeta^{-1})^i = \sum_i e_i(\zeta^i + \zeta^{-i})$ for each $e \in \mathbf{F}_2^k$.*

Proof. For $k = 1$: $T_1(f) = f$ so $(T_1(f))_1(\zeta + \zeta^{-1})^1 = f_1(\zeta + \zeta^{-1})$.

For $k \ge 2$: Define j as the largest power of 2 in $\{1, 2, \ldots, k-1\}$. Write e as (f, g) for some $f \in \mathbf{F}_2^j$ and $g \in \mathbf{F}_2^{k-j}$. Then $T_k(e) = (T_j(h), T_{k-j}(g))$ where $h_i = f_i + g_{j-i}$.

Both j and $k - j$ are smaller than k, so by induction

$$\begin{aligned}\sum_i (T_j(h))_i(\zeta + \zeta^{-1})^i &= \sum_i h_i(\zeta^i + \zeta^{-i}),\\ \sum_i (T_{k-j}(g))_i(\zeta + \zeta^{-1})^i &= \sum_i g_i(\zeta^i + \zeta^{-i}).\end{aligned}$$

Now use $(\zeta + \zeta^{-1})^j = \zeta^j + \zeta^{-j}$ to see that

$$\begin{aligned}\sum_i (T_k(e))_i(\zeta + \zeta^{-1})^i &= \sum_i (T_j(h))_i(\zeta + \zeta^{-1})^i + (\zeta + \zeta^{-1})^j \sum_i (T_{k-j}(g))_i(\zeta + \zeta^{-1})^i\\ &= \sum_i h_i(\zeta^i + \zeta^{-i}) + (\zeta^j + \zeta^{-j}) \sum_i g_i(\zeta^i + \zeta^{-i})\\ &= \sum_i (f_i + [i \ne 0]g_{j-i})(\zeta^i + \zeta^{-i}) + \sum_i g_i(\zeta^{i+j} + \zeta^{-i-j} + \zeta^{i-j} + \zeta^{j-i})\\ &= \sum_i (f_i + g_{j-i})(\zeta^i + \zeta^{-i}) + \sum_i g_i(\zeta^{i+j} + \zeta^{-i-j} + \zeta^{i-j} + \zeta^{j-i})\\ &= \sum_i f_i(\zeta^i + \zeta^{-i}) + \sum_i g_i(\zeta^{i+j} + \zeta^{-i-j})\\ &= \sum_i e_i(\zeta^i + \zeta^{-i})\end{aligned}$$

as claimed. The replacement of $[i \ne 0]$ by 1 on the fourth line follows from $[i \ne 0](\zeta^i + \zeta^{-i}) = \zeta^i + \zeta^{-i}$; note that $\zeta^0 + \zeta^{-0} = 0$. □

3.5. Speed of the transformation. The following in-place algorithm replaces $e \in \mathbf{F}_2^k$ by $T_k(e)$:

- Stop if $k = 1$.
- Define j as the largest power of 2 in $\{1, 2, \ldots, k-1\}$.
- Add e_{2j-i} into e_i for $\max\{1, 2j-k\} \le i \le j-1$. (Now $e = (h, g)$ in the notation of the definition of T_k.)
- Recursively apply T_j to the first j coefficients of e.
- Recursively apply T_{k-j} to the remaining coefficients of e.

Inverting this algorithm is a simple matter of carrying out the same additions in reverse order.

The cost of T_k is $\min\{j-1, k-j\}$ plus the costs of T_j and T_{k-j}. An easy induction shows that if $k = 2^{k_0} + 2^{k_1} + \cdots$, with $k_0 > k_1 > \ldots$, then the cost of T_k is exactly $\sum_i (2^{k_i-1}(k_i - 2 + 2i) + 1)$.

3.6. The $\boldsymbol{N \times N \to N}$ multiplication algorithm. The following algorithm computes $N(ab)$ given $N(a)$ and $N(b)$:

- Compute $A = T_n(N(a))$ and $B = T_n(N(b))$.
- Compute the product $P_2 z^2 + \cdots + P_{2n} z^{2n}$ of the polynomials $A_1 z + \cdots + A_n z^n$ and $B_1 z + \cdots + B_n z^n$ in the polynomial ring $\mathbf{F}_2[z]$.
- Compute $p = T_{2n}^{-1}(0, P_2, \ldots, P_{2n})$.
- Compute $N(ab) = (p_1 + p_{2n}, p_2 + p_{2n-1}, \ldots, p_n + p_{n+1})$.

Recall that $a = \sum_i N(a)_i (\zeta^i + \zeta^{-i})$ by definition of N, so $a = \sum_i A_i (\zeta + \zeta^{-1})^i$ by Theorem 3.4. Similarly $b = \sum_i B_i (\zeta + \zeta^{-1})^i$. Hence $ab = \sum_i P_i (\zeta + \zeta^{-1})^i$, so $ab = \sum_i p_i (\zeta^i + \zeta^{-i})$ by Theorem 3.4, so $ab = (p_1 + p_{2n})(\zeta + \zeta^{-1}) + \cdots + (p_n + p_{n+1})(\zeta^n + \zeta^{-n})$.

The two computations of T_n each cost $\sum_i (2^{n_i-1}(n_i - 2 + 2i) + 1)$ if $n = 2^{n_0} + 2^{n_1} + \cdots$ with $n_0 > n_1 > \ldots$. The polynomial multiplication costs $M(n)$. The computation of p costs $\sum_i (2^{n_i}(n_i - 1 + 2i) + 1)$. The final computation of $N(ab)$ costs $n = \sum_i 2^{n_i}$.

The total number of bit operations is $M(n) + \sum_i (2^{n_i}(2n_i - 2 + 4i) + 3)$. The overhead term $\sum_i (2^{n_i}(2n_i - 2 + 4i) + 3)$ is approximately $2n \log_2(n/2)$, and a trivial computer calculation shows that it is bounded by $2(n+2)\log_2(n/2)$ for $4 \le n \le 100000$.

We comment that the 0 component in the T_{2n}^{-1} input allows a subsequent addition of 0 to be eliminated. This speedup might seem too minor to be worth mentioning, and our operation counts in this section do not take it into account, but the underlying idea helps produce much larger savings in subsequent sections.

4 Type-II Polynomial Basis

Let us pause to review the attractive features of the (permuted) type-II normal basis $\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \ldots, \zeta^n + \zeta^{-n}$. The multiplication overhead, compared to size-n polynomial multiplication, is only about $2n \log_2(n/2)$. Repeated squaring is a very fast permutation, costing no bit operations.

This section presents a multiplication algorithm for the non-traditional polynomial basis $c, c^2, \ldots, c^n$, where $c = \zeta + \zeta^{-1}$. The overhead in the new algorithm is only about $n \log_2 n$. Repeated squaring in this basis is more complicated than a permutation but is still very fast, costing only $n \log_2(n/4)$ bit operations. We refer to this basis as a "type-II optimal polynomial basis" because of its close connection to the type-II optimal normal basis.

For comparison, in a traditional low-weight polynomial basis, the multiplication overhead is typically $2n$ (for trinomials) or $4n$ (for pentanomials), and *single* squarings are fast, but *repeated* squarings such as $a \mapsto a^{2^{\lfloor n/3 \rfloor}}$ are very slow.

We obtain our best results by combining type-II polynomial basis P with type-II normal basis N. This combined system keeps repeated-squaring inputs in N form, and keeps multiplication inputs in P form. Multiplications are $P \times P \to N$ when the outputs are used for repeated squarings, and $P \times P \to N \to P$ when the outputs are used for repeated squarings *and* for multiplications, but $P \times P \to P$ when the outputs are used solely for multiplications.

4.1. The $N \to P$ and $P \to N$ conversions. We begin by reinterpreting the transformation T_n in Section 3 as a fast conversion from type-II normal basis to type-II polynomial basis: Theorem 3.4 implies that $T_n(N(a)) = P(a)$. This also means that T_n^{-1} is a fast conversion from type-II polynomial basis to type-II normal basis: $T_n^{-1}(P(a)) = N(a)$. Recall that each of these conversions costs $\sum_i (2^{n_i-1}(n_i - 2 + 2i) + 1) \approx (n/2)\log_2(n/4)$.

For comparison: Shokrollahi in [Sho07, Theorem 28], emphasizing "the most important property" of his multiplier, showed that conversion between a type-II normal basis and the basis $1, c, c^2, \ldots, c^{n-1}$ takes time $O(n \log_2 n)$. We simplify and accelerate the conversion by shifting to the basis $c, c^2, \ldots, c^n$. The speedup is $\Theta(n)$ operations. The simplification is illustrated by the fact that our basis conversion naturally appears as a subroutine in our multiplication algorithm, whereas modifying the multiplication algorithm from [Sho07] to use the basis conversion from [Sho07, Theorem 28] would slow down the multiplication algorithm.

4.2. The $P \times P \to N$ multiplication algorithm. We next observe that the $N \times N \to N$ multiplication algorithm of Section 3 factors into two $N \to P$ conversions and a $P \times P \to N$ multiplication algorithm.

Specifically, the first step of the $N(a), N(b) \mapsto N(ab)$ algorithm of Section 3 computes $A = T_n(N(a))$ and $B = T_n(N(b))$; i.e., it computes $A = P(a)$ and $B = P(b)$. The remaining steps make no further use of $N(a)$ and $N(b)$: they start from $A = P(a)$ and $B = P(b)$ and compute $N(ab)$. In other words, the remaining steps are exactly a $P \times P \to N$ multiplication algorithm. This $P \times P \to N$ multiplication algorithm costs $M(n) + \sum_i (2^{n_i}(n_i + 2i) + 1) \approx M(n) + n \log_2 n$.

4.3. The $P \times P \to P$ multiplication algorithm. Composing $P \times P \to N$ with a final $N \to P$ conversion would produce a $P \times P \to P$ multiplication algorithm with overhead approximately $n(1.5 \log_2 n - 1)$. This algorithm would feed the size-$2n$ polynomial product through a size-$2n$ transform, then fold the result in half using $\zeta^{2n+1-i} + \zeta^{i-2n-1} = \zeta^i + \zeta^{-i}$, then transform the size-n result from N to P.

We do better by separately handling the two halves of the polynomial product. The point is that

$$\begin{aligned} &T_n(\mathrm{fold}(T_{2n}^{-1}(\mathrm{bottom}, \mathrm{top}))) \\ &\quad = T_n(\mathrm{fold}(T_{2n}^{-1}(\mathrm{bottom}, 0, \ldots, 0))) + T_n(\mathrm{fold}(T_{2n}^{-1}(0, \ldots, 0, \mathrm{top}))) \\ &\quad = \mathrm{bottom} + T_n(\mathrm{fold}(T_{2n}^{-1}(0, \ldots, 0, \mathrm{top}))). \end{aligned}$$

Instead of uselessly transforming the bottom half back and forth between P and N, we simply leave it in P and add it at the end. We use transforms as a fast mechanism to reduce the top half from coefficients of $c^{n+1}, \ldots, c^{2n}$ to coefficients of $c^1, \ldots, c^n$.

In the computation of $T_{2n}^{-1}(0, \ldots, 0, \mathrm{top})$, and in the subsequent folding, we systematically eliminate all additions of 0. For any particular n one can do this elimination by hand, keeping track of which intermediate values are 0; or one can generate straight-line code for the entire computation and use standard optimizing-compiler tools.

We have done this optimization for all $n \in \{1, 2, \ldots, 100000\}$ and found that the cost of $\mathrm{fold}(T_{2n}^{-1}(0, \ldots, 0, \mathrm{top}))$ can in every case be computed as follows. Write $n + 1$ (not n) as $2^{n_0} + 2^{n_1} + \cdots + 2^{n_r}$ with $n_0 > n_1 > \cdots > n_r$. The cost is then $\sum_i 2^{n_i-1}(n_i + 4i)$ *minus* a nonnegative rebate. The rebate is $2r$, plus n_r, plus 1 for each 11 in the binary expansion of $n + 1 - 2^{n_r}$, plus 2 for each 111 in the same binary expansion, plus 4 for each 1111, plus 8 for each 11111, etc.

Examples: If $n = 131$ then $n+1$ has binary expansion 10000100, so the rebate is $2 \cdot 1 + 2 = 4$, and the cost is $2^{7-1}(7) + 2^{2-1}(2+4) - 4 = 456$. If $n = 491$ then $n+1$ has binary expansion 111101100, with 3 occurrences of 11 before the last bit, 2 occurrences of 111, and 1 occurrence of 1111, so the rebate is $2 \cdot 5 + 2 + 3 \cdot 1 + 2 \cdot 2 + 1 \cdot 4 = 23$, and the cost is $2^{8-1}(8) + 2^{7-1}(7+4) + 2^{6-1}(6+8) + 2^{5-1}(5+12) + 2^{3-1}(3+16) + 2^{2-1}(2+20) - 23 = 2545$.

To summarize, $P \times P \to P$ multiplication involves

- cost $M(n)$ for the polynomial product;
- the cost discussed above, approximately $(n/2)\log_2 n$;
- the cost of T_n, approximately $(n/2)\log_2(n/4)$; and
- cost n for the final addition of the bottom half.

The total cost is approximately $M(n) + n\log_2 n$, similar to the cost of $P \times P \to N$ multiplication. These approximations should not be viewed as equalities: a closer look shows that $P \times P \to P$ multiplication costs about $\sum_i 2^{n_i} i$ more than $P \times P \to N$ multiplication.

4.4. Dynamically mixing $\boldsymbol{N}$ and $\boldsymbol{P}$. At this point our basic tools are as follows:

- $N \to P$ conversion: 1 transform, cost $\sum_i (2^{n_i-1}(n_i - 2 + 2i) + 1)$.
- $P \to N$ conversion: 1 transform, cost $\sum_i (2^{n_i-1}(n_i - 2 + 2i) + 1)$.
- $N \to N$ repeated squarings: 0 transforms, cost 0.
- $P \times P \to N$ multiplication: 2 transforms (actually one double-size transform), cost $M(n) + \sum_i (2^{n_i}(n_i + 2i) + 1)$.
- $P \times P \to P$ multiplication: 2 transforms, slightly larger cost as discussed above.

There are several reasonable ways to combine these tools. One extreme is to work everywhere in N, using $N \to N$ repeated squarings (0 transforms) and $N \times N \to P \times P \to N$ multiplications (4 transforms). Another extreme is to work everywhere in P, using $P \to N \to N \to P$ repeated squarings (2 transforms) and $P \times P \to P$ multiplications (2 transforms).

We take a more fluid approach, mixing the advantages of both approaches. We compute $N(a)$ for variables a that will be used in repeated squarings; we compute $P(a)$ for variables a that will be used in multiplications; we compute both $N(a)$ and $P(a)$ for variables a that will be used in both repeated squarings and multiplications. The overall number of transforms in this approach is 0 for each squaring and between 2 and 4 for each multiplication, depending on the exact pattern of multiplications and repeated squarings. See Section 5 for an illustrative example.

We briefly comment that $P \to P$ *single* squaring can be sped up by the same idea used in $P \times P \to P$ multiplication. However, in every application so far where we have tried this approach, we have found a faster solution that uses $N \to N$ squaring and rearranges the earlier computations.

5 Case Study: ECC2K-130

This section illustrates the use of optimal polynomial bases and optimal normal bases in the ECC2K-130 computation mentioned in Section 1. Specifically, this section shows that the $5B+5$ multiplications in B iterations of the ECC2K-130 iteration function from [BBB$^+$09] can be carried out with an overhead of only $5854B + 9408$ bit operations. The original Shokrollahi approach, with our improved analysis, would have used $8565B + 8565$ bit operations.

5.1. Review of the iteration function. We take the perspective of an implementor faced with the job of implementing the ECC2K-130 iteration function from [BBB$^+$09], the bottleneck in the ECC2K-130 computation. To keep this paper self-contained we now repeat the definition of the iteration function.

The input to an iteration is a pair $(x, y) \in \mathbf{F}_{2^{131}} \times \mathbf{F}_{2^{131}}$ satisfying two conditions: first, $y^2 + xy = x^3 + 1$; second, x has trace 0, i.e., $N(x)$ has even Hamming weight. The output of the iteration is the pair (x', y') defined by the equations

$$j = 3 + \left(\frac{\text{weight}(N(x))}{2} \bmod 8\right), \qquad \lambda = \frac{y + y^{2^j}}{x + x^{2^j}},$$
$$x' = \lambda^2 + \lambda + x + x^{2^j}, \qquad y' = \lambda(x + x') + x' + y.$$

One can check that $(y')^2 + x'y' = (x')^3 + 1$ and that x' has trace 0.

This iteration function can be computed using $3 + (5/B)$ multiplications for a B-way-batched inversion of $x + x^{2^j}$; 1 multiplication of the inverse by $y + y^{2^j}$, producing λ; and 1 multiplication of λ by $x + x'$. All of these stages are discussed in more detail below.

See [BBB$^+$09] for further information on how these iterations are being used to solve the ECC2K-130 challenge. We comment that thousands of CPU cores have already been busy for months computing these iterations, and that many more cores are being added; obviously every speedup in the computation is valuable.

5.2. The main loop. It is natural to represent the input (x, y) as $(N(x), N(y))$: the first step is to compute the weight of $N(x)$, and both x and y are then fed through repeated squarings. On the other hand, dividing $y + y^{2^j}$ by $x + x^{2^j}$ requires $P(y + y^{2^j})$ and $P(1/(x + x^{2^j}))$. The quotient λ is then used for both a squaring λ^2 and a multiplication $\lambda(x + x')$, so we compute both $N(\lambda)$ and $P(\lambda)$.

Figure 5.1 shows the resulting data flow between representations of various field elements. There are 4 explicit size-131 transforms, and 2 multiplications $P \times P \to N$ each involving 2 size-131 transforms. Working solely with N, and with an $N \times N \to N$ multiplication subroutine, would require an extra transform for $N(1/d)$. Note that more transforms are saved inside the inversion, as discussed below.

Figure 5.1 shows computations from $N(y)$ through $N(y^{2^j})$ in parallel with computations from $N(x)$ through $N(x^{2^j})$. To reduce storage requirements, cache misses, etc., the ECC2K-130 software actually delays the $N(y^{2^j})$ computations until after the inversion.

5.3. Batching inversions. Montgomery in [Mon87, Section 10.3.1] suggested computing $1/d_1$ and $1/d_2$ as $d_2/(d_1d_2)$ and $d_1/(d_1d_2)$. This suggestion eliminates 1 inversion in favor of 3 multiplications. We are not aware of any inversion method for $\mathbf{F}_{2^{131}}$ that can compete with 3 multiplications if the multiplications are performed by state-of-the-art techniques.

A batch of B parallel iterations involves B inversions $1/d_1, 1/d_2, \ldots, 1/d_B$. Merging the first two inversions, then merging with the next, etc., leads to the following standard computation, replacing $B - 1$ inversions with $3(B - 1)$ multiplications: first compute $d_1d_2, d_1d_2d_3, \ldots,$ $d_1d_2 \cdots d_B$ using $B - 1$ multiplications; then compute $1/(d_1d_2 \cdots d_B)$ using a single inversion; then compute $1/d_B = (d_1d_2 \cdots d_{B-1})/(d_1d_2 \cdots d_B)$ and $1/(d_1d_2 \cdots d_{B-1}) = d_B/(d_1d_2 \cdots d_B)$ using 2 multiplications, etc.

The single central inversion begins with squarings, as discussed below, and therefore takes $N(d_1d_2 \cdots d_B)$ as input. However, all of the intermediate products here are used solely for further multiplications, so we represent them in P form. Figure 5.2 shows the resulting data

Compute weight$(N(x)) = 2b_1 + 4b_2 + 8b_3 + \cdots$.

Compute $r = x^{2^3}$, $s = r + b_1(r^2 + r)$, $t = s + b_2(s^4 + s)$, $x^{2^j} = t + b_3(t^{16} + t)$. Similarly y^{2^j}.

Compute $d = x + x^{2^j}$, $e = y + y^{2^j}$.

Compute $1/d$.

Compute $\lambda = e/d$.

Compute $x' = \lambda^2 + \lambda + d$.

Compute $\lambda(x + x')$.

Compute $y' = \lambda(x + x') + x' + y$.

$$\begin{tikzcd}
N(x) \arrow[dr] \arrow[drr] \arrow[drrr] \arrow[dd] \arrow[dddddd, bend right] \arrow[ddddddddddddrr, bend left] & & & & N(y) \arrow[dd] \arrow[dddddd, bend left] \arrow[ddddddddddddddd, bend left] \\
& b_1 \arrow[ddl] \arrow[ddrrr] & b_2 \arrow[dddll] \arrow[dddrr] & b_3 \arrow[ddddlll] \arrow[ddddr] & \\
N(x^{2^3}) \arrow[d] & & & & N(y^{2^3}) \arrow[d] \\
N(s) = N(x^{2^{3+b_1}}) \arrow[d] & & & & N(y^{2^{3+b_1}}) \arrow[d] \\
N(x^{2^{3+b_1+2b_2}}) \arrow[d] & & & & N(y^{2^{3+b_1+2b_2}}) \arrow[d] \\
N(x^{2^j}) \arrow[d] & & & & N(y^{2^j}) \arrow[d] \\
N(d) \arrow[d, "\text{convert}"] \arrow[ddddd, bend right] & & & & N(e) \arrow[d, "\text{convert}"] \\
P(d) \arrow[d, "\text{and then a miracle occurs}"] & & & & P(e) \arrow[ddll] \\
P(1/d) \arrow[drr] & & & & \\
& & N(\lambda) \arrow[dll] \arrow[ddll] \arrow[drr, "\text{convert}"] & & \\
N(\lambda^2) \arrow[d] & & & & P(\lambda) \arrow[ddddl] \\
N(x') \arrow[drr] \arrow[ddddrrrr, bend right] & & & & \\
& & N(x + x') \arrow[d, "\text{convert}"] & & \\
& & P(x + x') \arrow[dr] & & \\
& & & N(\lambda(x + x')) \arrow[dr] & \\
& & & & N(y')
\end{tikzcd}$$

Fig. 5.1. The ECC2K-130 iteration function.

flow for $B = 4$. Working solely with N, and with an $N \times N \to N$ multiplication subroutine, would *double* the number of transforms.

Other merging patterns, such as a balanced tree, reduce latency without changing the number of operations. The same comments regarding P and N apply to arbitrary merging patterns.

$$\begin{tikzcd}
 & P(d_2) \arrow[dr] \arrow[dddddddr, bend right=40] & P(d_1) \arrow[d] \arrow[dddddddr, bend left=40] & \\
\text{Compute } d_1d_2. & P(d_3) \arrow[dr] \arrow[dddddr, bend right=40] & P(d_1d_2) \arrow[d] \arrow[dddddr, bend left=40] & \\
\text{Compute } d_1d_2d_3. & P(d_4) \arrow[dr] \arrow[dddr, bend right=40] & P(d_1d_2d_3) \arrow[d] \arrow[dddr, bend left=40] & \\
\text{Compute } d_1d_2d_3d_4. & & N(d_1d_2d_3d_4) \arrow[d, "\text{and then a miracle occurs}"] & \\
\text{Compute } 1/(d_1d_2d_3d_4). & & P(1/(d_1d_2d_3d_4)) \arrow[d] \arrow[dr] & \\
\text{Compute } 1/d_4 \text{ and } 1/(d_1d_2d_3). & & P(1/(d_1d_2d_3)) \arrow[d] \arrow[dr] & P(1/d_4) \\
\text{Compute } 1/d_3 \text{ and } 1/(d_1d_2). & & P(1/(d_1d_2)) \arrow[d] \arrow[dr] & P(1/d_3) \\
\text{Compute } 1/d_2 \text{ and } 1/d_1. & & P(1/d_1) & P(1/d_2)
\end{tikzcd}$$

Fig. 5.2. Batching 4 independent inversions.

5.4. Core inversions. Eventually one has to actually invert something. Inversion time is amortized across a batch of B iterations, but B is often limited by communication costs, making inversion an important part of the ECC2K-130 computation.

The standard branchless inversion method for $\mathbf{F}_{2^n}$, certainly not the only method, is to compute a $(2^n - 2)$nd power. This inversion method is also important in many other computations, so we describe the details for general n before focusing on $n = 131$.

The standard method of computing a $(2^n - 2)$nd power uses $n - 1$ squarings and just r multiplications, where r is the length of an "ℓ_0 chain" for $n - 1$; an ℓ_0 chain is a particular type of addition chain. The idea is to convert a chain $1 = e_0, e_1, \ldots, e_r = n - 1$ into a chain containing $1 = 2^{e_0} - 1, 2^{e_1} - 1, \ldots, 2^{e_r} - 1 = 2^{n-1} - 1$ along with various doublings; i.e., to compute $x^1 = x^{2^{e_0}-1}, \ldots, x^{2^{e_r}-1} = x^{2^{n-1}-1}$ along with various squarings. This powering method was introduced by Brauer in 1939 for the special case of "star chains" and by Hansen in 1959 for all ℓ_0 chains. See [Bra39], [Han59], and [Knu97b, Section 4.6.3, Theorem G]. Note that the shortest ℓ_0 chains are as short as the shortest addition chains for all integers below 5784689; see [Cli05].

In particular, a simple binary addition chain achieves $r = \lfloor \log_2(n - 1) \rfloor + \text{weight}(n-1) - 1$, producing an inversion method that takes $n-1$ squarings and $\lfloor \log_2(n - 1) \rfloor + \text{weight}(n-1) - 1$

multiplications. This inversion method is often credited to the 1988 paper [IT88] by Itoh and Tsujii. For most values of n one can do noticeably better (often more than $1.5\times$ better!) by switching to a standard "windowing" addition chain for $n-1$, producing an inversion method that takes $n-1$ squarings and $(1+o(1))\log_2 n$ multiplications. For further discussion of this inversion method see [vzGN99] and [Nöc01].

In the case $n = 131$ we take the length-8 addition chain $1, 2, 4, 8, 16, 32, 64, 65, 130$ for $n-1$. We could compute

$$x, x^2, x^3, x^{12}, x^{15}, x^{240}, x^{255} = x^{2^8-1}, x^{2^{16}-2^8}, x^{2^{16}-1}, x^{2^{32}-2^{16}}, x^{2^{32}-1},$$
$$x^{2^{64}-2^{32}}, x^{2^{64}-1}, x^{2^{65}-2}, x^{2^{65}-1}, x^{2^{130}-2^{65}}, x^{2^{130}-1}, x^{2^{131}-2} = x^{-1}$$

but we eliminate a final transform by moving the final squaring to the beginning:

$$x, x^2, x^4, x^6, x^{24}, x^{30}, x^{480}, x^{510} = x^{2^9-2}, x^{2^{17}-2^9}, x^{2^{17}-2}, x^{2^{33}-2^{17}}, x^{2^{33}-2},$$
$$x^{2^{65}-2^{33}}, x^{2^{65}-2}, x^{2^{66}-4}, x^{2^{66}-2}, x^{2^{131}-2^{66}}, x^{2^{131}-2} = x^{-1}.$$

Figure 5.3 shows the resulting data flow. Working solely with N, and with an $N \times N \to N$ multiplication subroutine, would require an extra transform for $P(x^2)$, and an extra transform for $P(x^{-1})$.

5.5. Total overhead. A batch of $B \geq 2$ iterations involves the following multiplications and conversions:

- Inversion (see Figure 5.3): 8 multiplications $P \times P \to N$ and 15 conversions $N \to P$.
- Batching (see Figure 5.2 for $B = 4$): 1 multiplication $P \times P \to N$ and $3B-4$ multiplications $P \times P \to P$. Note that all B inversions together involve $8 + 1 + (3B - 4) = 3B + 5$ multiplications; i.e., $3 + (5/B)$ multiplications per inversion, as mentioned earlier.
- Iteration (B copies of Figure 5.1): $4B$ conversions $N \to P$ and $2B$ multiplications $P \times P \to N$.

In total there are

- $2B + 9$ multiplications $P \times P \to N$, each having overhead 909;
- $3B - 4$ multiplications $P \times P \to P$, each having overhead 912; and
- $4B + 15$ conversions $N \to P$, each having overhead 325.

The total overhead is $5854B+9408$, i.e., $5854+9408/B$ per iteration. To put this in perspective, the fastest known method for size-131 polynomial multiplication (see [Ber09a]) costs 11961 bit operations, and all of the other operations in the iteration cost 3929 bit operations.

For comparison, Shokrollahi's original approach would have used $5B + 5$ multiplications $N \times N \to N$, each costing $M(132)+1559$. (Shokrollahi's analysis actually says $M(132)+3462$; $M(132) + 1559$ is the result of our own analysis of Shokrollahi's algorithm, and has been computer-verified.) The fastest known methods for size-132 multiplication involve 154 bit operations more than the fastest known methods for size-131 multiplication; if these methods are used then each $N \times N \to N$ multiplication has overhead $154 + 1559 = 1713$, for a total overhead of $8565B + 8565$, i.e., 8565 per iteration.

5.6. Comparison to other normal-basis approaches. Almost all normal-basis papers before [Sho07] used more than $2n^2$ bit operations for each multiplication. Known bounds on

$$\begin{tikzcd}
 & N(x) \arrow[d] & \\
P(x^2) \arrow[ddr] \arrow[ddddddddddddddr, bend right] & N(x^2) \arrow[l, "\text{convert}"'] \arrow[d] & \\
 & N(x^4) \arrow[r, "\text{convert}"] & P(x^4) \arrow[dl] \\
P(x^6) \arrow[ddr] & N(x^6) \arrow[l, "\text{convert}"'] \arrow[d] & \\
 & N(x^{24}) \arrow[r, "\text{convert}"] & P(x^{24}) \arrow[dl] \\
P(x^{30}) \arrow[ddr] & N(x^{30}) \arrow[l, "\text{convert}"'] \arrow[d] & \\
 & N(x^{480}) \arrow[r, "\text{convert}"] & P(x^{480}) \arrow[dl] \\
P(x^{2^9-2}) \arrow[ddr] & N(x^{2^9-2}) \arrow[l, "\text{convert}"'] \arrow[d] & \\
 & N(x^{2^{17}-2^9}) \arrow[r, "\text{convert}"] & P(x^{2^{17}-2^9}) \arrow[dl] \\
P(x^{2^{17}-2}) \arrow[ddr] & N(x^{2^{17}-2}) \arrow[l, "\text{convert}"'] \arrow[d] & \\
 & N(x^{2^{33}-2^{17}}) \arrow[r, "\text{convert}"] & P(x^{2^{33}-2^{17}}) \arrow[dl] \\
P(x^{2^{33}-2}) \arrow[ddr] & N(x^{2^{33}-2}) \arrow[l, "\text{convert}"'] \arrow[d] & \\
 & N(x^{2^{65}-2^{33}}) \arrow[r, "\text{convert}"] & P(x^{2^{65}-2^{33}}) \arrow[dl] \\
 & N(x^{2^{65}-2}) \arrow[d] & \\
 & N(x^{2^{66}-4}) \arrow[r, "\text{convert}"] & P(x^{2^{66}-4}) \arrow[dl] \\
P(x^{2^{66}-2}) \arrow[ddr] & N(x^{2^{66}-2}) \arrow[l, "\text{convert}"'] \arrow[d] & \\
 & N(x^{2^{131}-2^{66}}) \arrow[r, "\text{convert}"] & P(x^{2^{131}-2^{66}}) \arrow[dl] \\
 & P(x^{2^{131}-2}) = P(x^{-1}) &
\end{tikzcd}$$

Fig. 5.3. Core inversion inside the ECC2K-130 iteration function.

$M(131)$ imply that $2 \cdot 131^2 > M(131) + 20000$, so these methods would incur an overhead of more than 100000 bit operations per iteration.

There are two previous methods that are asymptotically better than $2n^2$ bit operations. One method by Gao et al. uses *two* size-n polynomial multiplications and is obviously superseded by Shokrollahi's approach. The other method uses roughly $13n^{1.6}$ bit operations and is also not as fast as Shokrollahi's approach. See [vzGSS07] and [Sho07] for further discussion of both of these methods.

5.7. Comparison to traditional low-weight polynomial bases. The current ECC2K-130 attack software uses our techniques. The original ECC2K-130 attack software instead used a low-weight polynomial basis, specifically the basis $1, z, z^2, \ldots, z^{130}$ of $\mathbf{F}_{2^{131}} = \mathbf{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$. There is no trinomial basis for this field.

The obvious approach to multiplication in this polynomial basis has overhead $4 \cdot 130 = 520$: for example, one eliminates the coefficient of z^{260} by adding it to 4 previous coefficients. However, a closer look shows that 65 of these 520 additions can be reused, thanks to the even spacing of $z^2, z, 1$, reducing the multiplication overhead to 455.

Similarly, a *single* squaring costs just 203. The problem is that there are 21 squarings in Figure 5.1: 10 for x^{2^j} via r^2, s^4, t^{16}; another 10 for y^{2^j}; and another 1 for λ^2. Even worse, one still needs to convert x to $N(x)$ as a stepping-stone to weight$(N(x))$.

The total overhead for 5 multiplications and 21 squarings is $5 \cdot 455 + 21 \cdot 203 = 6538$ per iteration. The basis conversion from x to $N(x)$ can be performed in 3380 bit operations as explained in [Ber09b]. We have seen some ideas for slightly reducing these costs, but nothing that could make this low-weight polynomial basis competitive with the approach explained in this paper.

References

[BBB+09] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. `http://eprint.iacr.org/2009/541`.

[Ber09a] Daniel J. Bernstein. Batch binary Edwards. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2009. `http://cr.yp.to/papers.html#bbe`.

[Ber09b] Daniel J. Bernstein. Optimizing linear maps modulo 2, 2009. `http://cr.yp.to/papers.html#linearmod2`.

[Bra39] Alfred T. Brauer. On addition chains. *Bull. Amer. Math. Soc.*, 45:736–739, 1939.

[Cer97] Certicom. Certicom ECC Challenge. `http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf`, 1997.

[Cli05] Neil Clift. Hansen chains do not always produce optimum addition chains. Posting to sci.math on 31 Jul 2005, `http://sci.tech-archive.net/Archive/sci.math/2005-08/msg00447.html`, 2005.

[Han59] Walter Hansen. Zum Scholz–Brauerschen Problem. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 202:129–136, 1959. In German.

[IT88] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.

[IT89] Toshiya Itoh and Shigeo Tsujii. Structure of parallel multipliers for a class of fields $GF(2^m)$. *Information and Computation*, 83(1):21–40, 1989.

[Knu97a] Donald E. Knuth. *The art of computer programming. Vol. 1, Fundamental algorithms.* Addison-Wesley Publishing Company, Reading, MA, third edition, 1997. Addison-Wesley Series in Computer Science and Information Processing.

[Knu97b] Donald E. Knuth. *The art of computer programming. Vol. 2, Seminumerical algorithms.* Addison-Wesley Publishing Company, Reading, MA, third edition, 1997. Addison-Wesley Series in Computer Science and Information Processing.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[MOVW89] Ronald C. Mullin, I. M. Onyszchuk, Scott A. Vanstone, and R. M. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22(2):149–161, 1989.

[Nöc01] Michael Nöcker. *Data structures for parallel exponentiation in finite fields.* PhD thesis, Universität Paderborn, 2001. `http://math-www.uni-paderborn.de/~aggathen/Publications/noc01.ps`.

[Sho07] Jamshid Shokrollahi. *Efficient implementation of elliptic curve cryptography on FPGAs.* PhD thesis, Universität Bonn, 2007. `http://hss.ulb.uni-bonn.de/diss_online/math_nat_fak/2007/shokrollahi_jamshid/0960.pdf`.

[vzGN99] Joachim von zur Gathen and Michael Nöcker. Computing special powers in finite fields (extended abstract). In *ISSAC*, pages 83–90, 1999.

[vzGSS07] Joachim von zur Gathen, Amin Shokrollahi, and Jamshid Shokrollahi. Efficient multiplication using type 2 optimal normal bases. In Claude Carlet and Berk Sunar, editors, *WAIFI*, volume 4547 of *Lecture Notes in Computer Science*, pages 55–68. Springer, 2007.