McBits: fast constant-time code-based cryptography

Daniel J. Bernstein[1,2], Tung Chou[2], and Peter Schwabe[3]

[1] Department of Computer Science
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, the Netherlands
blueprint@crypto.tw
[3] Digital Security Group
Radboud University Nijmegen, Mailbox 47, P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
peter@cryptojedi.org

Abstract. This paper presents extremely fast algorithms for code-based public-key cryptography, including full protection against timing attacks. For example, at a 2^{128} security level, this paper achieves a reciprocal decryption throughput of just 60493 cycles (plus cipher cost etc.) on a single Ivy Bridge core. These algorithms rely on an additive FFT for fast root computation, a transposed additive FFT for fast syndrome computation, and a sorting network to avoid cache-timing attacks.

Keywords: McEliece, Niederreiter, CFS, bitslicing, software implementation.

1 Introduction

This paper presents new software speed records for public-key cryptography: for example, more than 400000 decryptions per second at a 2^{80} security level, or 200000 per second at a 2^{128} security level, on a $215 4-core 3.4GHz Intel Core i5-3570 CPU. These speeds are fully protected against simple timing attacks, cache-timing attacks, branch-prediction attacks, etc.: all load addresses, all store addresses, and all branch conditions are public.

The public-key cryptosystem used here is a code-based cryptosystem with a long history, a well-established security track record, and even post-quantum security: namely, Niederreiter's dual form [49] of McEliece's hidden-Goppa-code cryptosystem [46]. This cryptosystem is well known to provide extremely fast

This work was supported by the Cisco University Research Program, by the National Science Foundation under grant 1018836, and by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005. Permanent ID of this document: e801a97c500b3ac879d77bcecf054ce5. Date: 2013.06.11.

encryption and *reasonably* fast decryption. Our main contributions are new decryption techniques that are (1) much faster and (2) fully protected against timing attacks, including the attacks by Strenzke in [63], [64], and [65].

The main disadvantage of this cryptosystem is that public keys are quite large: for example, 64 kilobytes for the 2^{80} security level mentioned above. In some applications the benefits of very fast encryption and decryption are outweighed by the costs of communicating and storing these keys. We comment that our work allows a tradeoff between key size and decryption time: because decryption is so fast we can afford "combinatorial list decoding", using many trial decryptions to guess a few error positions, which allows the message sender to add a few extra error positions (as proposed by Bernstein, Lange, and Peters in [15]), which increases security for the same key size, which allows smaller keys for the same security level.

We also present new speed records for generating signatures in the CFS code-based public-key signature system. Our speeds are an order of magnitude faster than previous work. This system has a much larger public key but is of interest for its short signatures and fast verification.

We will put all software described in this paper into the public domain.

To bitslice, or not to bitslice. The literature contains several success stories for bitsliced cryptographic computations, but those stories are for small S-boxes or large binary fields, while code-based cryptography relies on medium-size fields and seems to make much more efficient use of table lookups. The fastest previous software [19] for McEliece/Niederreiter decryption uses input-dependent table lookups for fast field arithmetic, uses input-dependent branches for fast root-finding, etc.

Despite this background we use bitslicing for the critical decoding step inside McEliece/Niederreiter decryption. Our central observation is that this decoding step is bottlenecked not by separate operations in a medium-size finite field, but by larger-scale polynomial operations over that finite field; state-of-the-art approaches to those polynomial operations turn out to interact very well with bitslicing. Our decoding algorithms end up using a surprisingly small number of bit operations, and as a result a surprisingly small number of cycles, setting new speed records for code-based cryptography, in some cases an order of magnitude faster than previous work.

The most important steps in our decoding algorithm are an "additive FFT" for fast root computation (Section 3) and a transposed additive FFT for fast syndrome computation (Section 4). It is reasonable to predict that the additive FFT will also reduce the energy consumed by *hardware* implementations of code-based cryptography. We also use a sorting network to efficiently simulate secret-index lookups in a large table (Section 5); this technique may be of independent interest for other computations that need to be protected against timing attacks.

Results: the new speeds. To simpify comparisons we have chosen to report benchmarks on a very widely available CPU microarchitecture, specifically the Ivy Bridge microarchitecture from Intel, which carries out one 256-bit vector arithmetic instruction per cycle. We emphasize, however, that our techniques

are not limited to this platform. Older Intel and AMD CPUs perform two or three 128-bit vector operations per cycle; common tablet/smartphone ARMs with NEON perform one or two 128-bit vector operations per cycle (exploited by Bernstein and Schwabe in [16], although not with bitslicing); the same techniques will also provide quite respectable performance using 64-bit registers, 32-bit registers, etc.

Table 1.1 reports our decoding speeds for various code parameters. Decoding time here is computed as 1/256 of the total latency measured for 256 simultaneous decoding operations. Decryption time is slightly larger, because it requires hashing, checking a MAC, and applying a secret-key cipher; see Section 6. We comment that the software supports a separate secret key for each decryption (although many applications do not need this), and that the latency of 256 decryptions is so small as to be unnoticeable in typical applications.

We use the usual parameter notations for code-based cryptography: $q = 2^m$ is the field size, n is the code length, t is the number of errors corrected, and $k = n - mt$. "Bytes" is the public-key size $\lceil k(n-k)/8 \rceil$; the rows are sorted by this column. "Total" is our cycle count (measured by the Ivy Bridge cycle counter with Turbo Boost and hyperthreading disabled) for decoding, including overhead beyond vector operations. This cycle count is partitioned into five stages: "perm" for initial permutation (Section 5), "synd" for syndrome computation (Section 4), "key eq" for solving the key equation (standard Berlekamp–Massey), "root" for root-finding (Section 3), and "perm" again for final permutation.

Some of the parameters in this table are taken from [15], which says that these parameters were designed to optimize security level subject to key sizes of 2^{16}, 2^{17}, 2^{18}, 2^{19}, and 2^{20} bytes. Some parameters are from [37]. Some parameters are from [19], and for comparison we repeat the Core 2 cycle counts reported in [19]. (We comment that the "cycles/byte" in [19] are cycles divided by $(k + \lfloor \lg \binom{n}{t} \rfloor)/8$.) Our speedups are much larger than any relevant differences between the Core 2 and the Ivy Bridge that we used for benchmarking; we will report Core 2 cycle counts for our software in a subsequent online update of this paper. "Sec" is the approximate security level reported by the `https://bitbucket.org/cbcrypto/isdfq` script from Peters [54], rounded to the nearest integer.

Some of the parameter choices from [19] are uninteresting in all of our metrics: they are beaten by other parameter choices in key size, speed, and security level. For these parameter choices we mark our cycle count in gray. Note that we have taken only previously published parameter sets; in particular, we have not searched for parameters that sacrifice key size to improve speed for the same security level.

Previous speeds for public-key cryptography. The eBATS benchmarking system [14] includes seven public-key encryption systems: `mceliece`, a McEliece implementation from Biswas and Sendrier (with $n = 2048$ and $t = 32$, slightly above a 2^{80} security level); `ntruees787ep1`, an NTRU implementation (2^{256} security) from Mark Etzel; and five sizes of RSA starting from `ronald1024` (2^{80} security). None of these implementations claim to be protected against timing attacks. On `h9ivy`, an Ivy Bridge CPU (Intel Core i5-3210M), the fastest

						Our speeds						
$q = 2^m$	n	t	k	bytes	sec	perm	synd	key eq	root	perm	total	[19]
2048	2048	27	1751	65006	81	3333	8414	3120	5986	3199	24051	
2048	1744	35	1359	65402	83	3301	9199	5132	6659	3145	27434	
2048	2048	32	1696	74624	87	3326	9081	4267	6699	3172	26544	445599
2048	2048	40	1608	88440	95	3357	9412	6510	6852	3299	29429	608172
4096	4096	21	3844	121086	87	8661	17496	2259	11663	8826	48903	288649
4096	2480	45	1940	130950	105	8745	21339	9276	14941	8712	63012	
4096	2690	56	2018	169512	119	8733	22898	14199	16383	8789	71000	
4096	4096	41	3604	221646	129	8622	20846	7714	14794	8520	60493	693822
8192	8192	18	7958	232772	91	23331	49344	3353	37315	23339	136679	317421
4096	3408	67	2604	261702	146	8983	24308	19950	17790	8686	79715	
8192	8192	29	7815	368282	128	22879	56336	7709	44727	22753	154403	540952
16384	16384	15	16174	424568	90	60861	99360	2337	79774	60580	302909	467818
8192	4624	95	3389	523177	187	22693	76050	70696	59409	22992	251838	
8192	6624	115	5129	958482	252	23140	83127	102337	65050	22971	296624	
8192	6960	119	5413	1046739	263	23020	83735	109805	66453	23091	306102	

Table 1.1. Number of cycles for decoding for various code parameters. See text for description.

encryption (for 59-byte messages) is 46940 cycles for `ronald1024` followed by 61440 cycles for `mceliece`, several more RSA results, and finally 398912 cycles for `ntruees787ep1`. The fastest decryption is 700512 cycles for `ntruees787ep1`, followed by 1219344 cycles for `mceliece` and 1340040 cycles for `ronald1024`.

A followup paper [19] by Biswas and Sendrier reports better decryption performance, 445599 cycles on a Core 2 for $n = 2048$ and $t = 32$. Sendrier says (private communication) that he now has better performance, below 300000 cycles. However, our speed of 26544 cycles for $n = 2048$ and $t = 32$ improves upon this by an order of magnitude, and also includes full protection against timing attacks.

eBATS also includes many Diffie–Hellman systems. One can trivially use Diffie–Hellman for public-key encryption; the decryption time is then the Diffie–Hellman shared-secret time plus some fast secret-key cryptography, and the encryption time is the same plus the Diffie–Hellman key-generation time. When we submitted this paper the fastest Diffie–Hellman shared-secret time reported from `h9ivy` was 182632 cycles (side-channel protected), set by the `curve25519` implementation from Bernstein, Duif, Lange, Schwabe, and Yang in [13]. The fastest time now is 77468 cycles (not side-channel protected), set by `gls254` from Oliveira, López, Aranha, and Rodríguez-Henríquez; see [50]. Our software takes just 60493 cycles (side-channel protected) for decryption with $n = 4096$ and $t = 41$ at the same 2^{128} security level.

We have found many claims that NTRU is orders of magnitude faster than RSA and ECC, but we have also found no evidence that NTRU can match our speeds. The fastest NTRU decryption report that we have found is from

Hermans, Vercauteren, and Preneel in [36]: namely, 24331 operations per second on a GTX 280 GPU.

Heyse and Güneysu in [37] report 17012 Niederreiter decryption operations per second on a Virtex6-LX240T FPGA for $n = 2048$ and $t = 27$. The implementation actually uses only 10% of the FPGA slices, so presumably one can run several copies of the implementation in parallel without running into place-and-route difficulties. A direct speed comparison between such different platforms does not convey much information, but we point out several ways that our decryption algorithm improves upon the algorithm used in [37]: we use an additive FFT rather than separate evaluations at each point ("Chien search"); we use a transposed additive FFT rather than applying a syndrome-conversion matrix; we do not even need to store the syndrome-conversion matrix, the largest part of the data stored in [37]; and we use a simple hash (see Section 6) rather than a constant-weight-word-to-bit-string conversion.

2 Field arithmetic

We construct the finite field $\mathbb{F}_{2^m}$ as $\mathbb{F}_2[x]/f$, where f is a degree-m irreducible polynomial. We use trinomial choices of f when possible. We use pentanomials for $\mathbb{F}_{2^{13}}$ and $\mathbb{F}_{2^{16}}$.

Addition. Addition in $\mathbb{F}_{2^m}$ is simply a coefficient-wise xor and costs m bit operations.

Multiplication. A field multiplication is composed of a multiplication in $\mathbb{F}_2[x]$ and reduction modulo f. We follow the standard approach of optimizing these two steps separately, and we use standard techniques for the second step. Note, however, that this two-step optimization is not necessarily optimal, even if each of the two steps is optimal.

For the first step we started from Bernstein's straight-line algorithms from http://binary.cr.yp.to/m.html. The mth algorithm is a sequence of XORs and ANDs that multiplies two m-coefficient binary polynomials. The web page shows algorithms for m as large as 1000; for McEliece/Niederreiter we use m between 11 and 16, and for CFS (Section 7) we use $m = 20$. These straight-line algorithms are obtained by combining different multiplication techniques as explained in [10]; for $10 \leq m \leq 20$ the algorithms use somewhat fewer bit operations than schoolbook multiplication. We applied various scheduling techniques (in some cases sacrificing some bit operations) to improve cycle counts.

Squaring. Squaring of a polynomial does not require any bit operations. The square of an m-coefficient polynomial $f = \sum_{i=0}^{m-1} a_i x^i$ is simply $f^2 = \sum_{i=0}^{m-1} a_i x^{2i}$. The only bit operations required for squaring in $\mathbb{F}_{2^m}$ are thus those for reduction. Note that half of the high coefficients are known to be zero; reduction after squaring takes only about half the bit operations of reduction after multiplication.

Inversion. We compute reciprocals in $\mathbb{F}_{2^m}$ as (2^m-2)nd powers. For $\mathbb{F}_{2^{20}}$ we use an addition chain consisting of 19 squarings and 6 multiplications. For smaller fields we use similar addition chains.

3 Finding roots: the Gao–Mateer additive FFT

This section considers the problem of finding all the roots of a polynomial over a characteristic-2 finite field. This problem is parametrized by a field size $q = 2^m$ where m is a positive integer. The input is a sequence of coefficients $c_0, c_1, \ldots, c_t \in \mathbb{F}_q$ of a polynomial $f = c_0 + c_1x + \cdots + c_tx^t \in \mathbb{F}_q[x]$ of degree at most t. The output is a sequence of q bits b_α indexed by elements $\alpha \in \mathbb{F}_q$ in a standard order, where $b_\alpha = 0$ if and only if $f(\alpha) = 0$.

Application to decoding. Standard decoding techniques have two main steps: finding an "error-locator polynomial" f of degree at most t, and finding all the roots of the polynomial in a specified finite field $\mathbb{F}_q$. In the McEliece/Niederreiter context it is traditional to take the field size q as a power of 2 and to take t on the scale of $q/\lg q$, typically between $0.1q/\lg q$ and $0.3q/\lg q$; a concrete example is $(q, t) = (2048, 40)$. In cases of successful decryption this polynomial will in fact have exactly t roots at the positions of errors added by the message sender.

Multipoint evaluation. In coding theory, and in code-based cryptography, the most common way to solve the root-finding problem is to simply try each possible root: for each $\alpha \in \mathbb{F}_q$, evaluate $f(\alpha)$ and then OR together the bits of $f(\alpha)$ in a standard basis, obtaining 0 if and only if $f(\alpha) = 0$.

The problem of evaluating $f(\alpha)$ for every $\alpha \in \mathbb{F}_q$, or more generally for every α in some set S, is called multipoint evaluation. Separately evaluating $f(\alpha)$ by Horner's rule for every $\alpha \in \mathbb{F}_q$ costs qt multiplications in $\mathbb{F}_q$ and qt additions in $\mathbb{F}_q$; if t is essentially linear in q (e.g., q or $q/\lg q$) then the total number of field operations is essentially quadratic in q. "Chien search" is an alternative method of evaluating each $f(\alpha)$, also using qt field additions and qt field multiplications.

There is an extensive literature on more efficient multipoint-evaluation techniques. Most of these techniques (for example, the "dcmp" method recommended by Strenzke in [65]) save at most small constant factors. Some of them are much more scalable: in particular, a 40-year-old FFT-based algorithm [21] by Borodin and Moenck evaluates an n-coefficient polynomial at any set of n points using only $n^{1+o(1)}$ field operations. On the other hand, the conventional wisdom is that FFTs are particularly clumsy for characteristic-2 fields, and in any case are irrelevant to the input sizes that occur in cryptography.

Additive FFT: overview. For multipoint evaluation we use a characteristic-2 "additive FFT" algorithm introduced in 2010 [32] by Gao and Mateer (improving upon previous algorithms by Wang and Zhu in [66], Cantor in [24], and von zur Gathen and Gerhard in [33]), together with some new improvements described below. This algorithm evaluates a polynomial at every element of $\mathbb{F}_q$, or more generally every element of an $\mathbb{F}_2$-linear subspace of $\mathbb{F}_q$. The algorithm

uses an essentially linear number of field operations; most of those operations are additions, making the algorithm particularly well suited for bitslicing.

The basic idea of the algorithm is to write f in the form $f_0(x^2-x)+xf_1(x^2-x)$ for two half-degree polynomials $f_0, f_1 \in \mathbb{F}_q[x]$; this is handled efficiently by the "radix conversion" described below. This form of f shows a large overlap between evaluating $f(\alpha)$ and evaluating $f(\alpha+1)$. Specifically, $(\alpha+1)^2-(\alpha+1) = \alpha^2-\alpha$, so

$$f(\alpha) = f_0(\alpha^2-\alpha) + \alpha f_1(\alpha^2-\alpha),$$
$$f(\alpha+1) = f_0(\alpha^2-\alpha) + (\alpha+1) f_1(\alpha^2-\alpha).$$

Evaluating both f_0 and f_1 at $\alpha^2-\alpha$ produces both $f(\alpha)$ and $f(\alpha+1)$ with just a few more field operations: multiply the f_1 value by α, add the f_0 value to obtain $f(\alpha)$, and add the f_1 value to obtain $f(\alpha+1)$.

The additive FFT applies this idea recursively. For example, if $\beta^2-\beta = 1$ then evaluating f at $\alpha, \alpha+1, \alpha+\beta, \alpha+\beta+1$ reduces to evaluating f_0 and f_1 at $\alpha^2-\alpha$ and $\alpha^2-\alpha+1$, which in turn reduces to evaluating four polynomials at $\alpha^4-\alpha$. One can handle any subspace by "twisting", as discussed below.

For comparison, a standard multiplicative FFT writes f in the form $f_0(x^2)+xf_1(x^2)$ (a simple matter of copying alternate coefficients of f), reducing the computation of both $f(\alpha)$ and $f(-\alpha)$ to the computation of $f_0(\alpha^2)$ and $f_1(\alpha^2)$. The problem in characteristic 2 is that α and $-\alpha$ are the same. The standard workaround is a radix-3 FFT, writing f in the form $f_0(x^3)+xf_1(x^3)+x^2f_2(x^3)$, but this is considerably less efficient.

We comment that the additive FFT, like the multiplicative FFT, is suitable for small hardware: it can easily be written as a highly structured iterative algorithm rather than a recursive algorithm, and at a small cost in arithmetic it can be written to use very few constants.

Additive FFT: detail. Consider the problem of evaluating a 2^m-coefficient polynomial f at all subset sums ($\mathbb{F}_2$-linear combinations) of $\beta_1, \ldots, \beta_m \in \mathbb{F}_q$: i.e., computing $f(0)$, $f(\beta_1)$, $f(\beta_2)$, $f(\beta_1+\beta_2)$, etc. Gao and Mateer handle this problem as follows.

If $m = 0$ then the output is simply $f(0)$. Assume from now on that $m \geq 1$.

If $\beta_m = 0$ then the output is simply two copies of the output for $\beta_1, \ldots, \beta_{m-1}$. (The algorithm stated in [32] is slightly less general: it assumes that $\beta_1, \ldots, \beta_m$ are linearly independent, excluding this case.) Assume from now on that $\beta_m \neq 0$.

Assume without loss of generality that $\beta_m = 1$. To handle the general case, compute $g(x) = f(\beta_m x)$, and observe that the output for $f, \beta_1, \beta_2, \ldots, \beta_m$ is the same as the output for $g, \beta_1/\beta_m, \beta_2/\beta_m, \ldots, 1$. (This is the "twisting" mentioned above. Obviously the case $\beta_m = 1$ is most efficient; the extent to which this case can be achieved depends on how many powers of 2 divide $\lg q$.)

Apply the radix conversion described below to find two 2^{m-1}-coefficient polynomials $f_0, f_1 \in \mathbb{F}_q[x]$ such that $f = f_0(x^2-x)+xf_1(x^2-x)$. Recursively evaluate f_0 at all subset sums of $\delta_1, \ldots, \delta_{m-1}$, where $\delta_i = \beta_i^2-\beta_i$. Also recursively evaluate f_1 at all subset sums of $\delta_1, \ldots, \delta_{m-1}$.

Observe that each subset sum $\alpha = \sum_{i \in S} \beta_i$ with $S \subseteq \{1, 2, \ldots, m-1\}$ has $\alpha^2-\alpha = \gamma$ where $\gamma = \sum_{i \in S} \delta_i$. Compute $f(\alpha)$ as $f_0(\gamma) + \alpha f_1(\gamma)$, and compute

$f(\alpha+1)$ as $f(\alpha)+f_1(\gamma)$. Note that these evaluation points α and $\alpha+1$ cover all subset sums of $\beta_1,\beta_2,\ldots,\beta_m$, since $\beta_m=1$.

The radix-conversion subroutine. Here is how to write a 2^m-coefficient polynomial $f=c_0+c_1x+\cdots+c_{2^m-1}x^{2^m-1}$ in the form $f_0(x^2-x)+xf_1(x^2-x)$, where f_0 and f_1 are 2^{m-1}-coefficient polynomials.

If $m=1$, simply take $f_0=c_0$ and $f_1=c_1$. Assume from now on that $m\geq 2$.

Abbreviate 2^{m-2} as n; then $f=c_0+c_1x+\cdots+c_{4n-1}x^{4n-1}$. Divide f by the power $(x^2-x)^n=x^{2n}-x^n$, obtaining a quotient Q and a remainder R: explicitly,

$$\begin{aligned}Q&=(c_{2n}+c_{3n})+\cdots+(c_{3n-1}+c_{4n-1})x^{n-1}+c_{3n}x^n+\cdots+c_{4n-1}x^{2n-1},\\ R&=(c_0)+\cdots+(c_{n-1})x^{n-1}\\&\qquad+(c_n+c_{2n}+c_{3n})x^n+\cdots+(c_{2n-1}+c_{3n-1}+c_{4n-1})x^{2n-1}.\end{aligned}$$

This takes $2n=2^{m-1}$ additions; note that $c_{2n}+c_{3n}$ etc. from Q are reused in R.

Recursively write Q in the form $Q_0(x^2-x)+xQ_1(x^2-x)$, and recursively write R in the form $R_0(x^2-x)+xR_1(x^2-x)$. Finally compute $f_0=R_0+x^nQ_0$ and $f_1=R_1+x^nQ_1$.

This procedure is a special case of a general radix-conversion method credited to Schönhage in [41, page 638]. The standard method to convert an integer or polynomial to radix r is to divide it by r, output the remainder, and recursively handle the quotient. Schönhage's method is to divide by a power of r and handle both the quotient and remainder recursively. The division is particularly efficient when the power of r is sparse, as in the case of $(x^2-x)^n=x^{2n}-x^n$.

Improvement: 1-coefficient polynomials. Gao and Mateer show that for $q=2^m$ this additive-FFT algorithm uses $2q\lg q-2q+1$ multiplications in $\mathbb{F}_q$ and $(1/4)q(\lg q)^2+(3/4)q\lg q-(1/2)q$ additions in $\mathbb{F}_q$. The $\beta_m=1$ optimization removes many multiplications when it is applicable.

We do better by generalizing from one parameter to two, separating the maximum polynomial degree t from the number 2^m of evaluation points. Our main interest is not in the case $t+1=2^m$, but in the case that t is smaller than 2^m by a logarithmic factor.

The adjustments to the algorithm are straightforward. We begin with a polynomial having $t+1$ coefficients. If $t=0$ then the output is simply 2^m copies of $f(0)$, which we return immediately without any additions or multiplications. If $t\geq 1$ then we continue as in the algorithm above; f_0 has $\lceil (t+1)/2\rceil$ coefficients, and f_1 has $\lfloor (t+1)/2\rfloor$ coefficients. Note that $t+1$ and 2^m each drop by a factor of approximately 2 in the recursive calls.

It is of course possible to zero-pad a $(t+1)$-coefficient polynomial to a 2^m-coefficient polynomial and apply the original algorithm, but this wastes considerable time manipulating coefficients that are guaranteed to be 0.

Improvement: 2-coefficient and 3-coefficient polynomials. We further accelerate the case that t is considerably smaller than 2^m, replacing many multiplications with additions as follows.

Recall that the last step of the algorithm involves 2^{m-1} multiplications of the form $\alpha f_1(\gamma)$. Here α runs through all subset sums of $\beta_1, \beta_2, \ldots, \beta_{m-1}$, and $\gamma = \alpha^2 - \alpha$. The multiplication for $\alpha = 0$ can be skipped but all other multiplications seem nontrivial.

Now consider the case that $t \in \{1, 2\}$. Then f_1 has just 1 coefficient, so the recursive evaluation of f_1 produces 2^{m-1} copies of $f_1(0)$, as discussed above. The products $\alpha f_1(\gamma) = \alpha f_1(0)$ are then nothing more than subset sums of $\beta_1 f_1(0), \beta_2 f_1(0), \ldots, \beta_{m-1} f_1(0)$. Instead of $2^{m-1} - 1$ multiplications we use just $m - 1$ multiplications and $2^{m-1} - m$ additions.

Results. Table 3.1 displays the speed of the additive FFT, including these improvements, for an illustrative sample of field sizes $q = 2^m$ and degrees t taken from our applications to decoding.

Other algorithms. We briefly mention a few alternative root-finding algorithms.

In the standard McEliece/Niederreiter context, f is known in advance to have t distinct roots (for valid ciphertexts). However, in the signing context of Section 7 and the "combinatorial list decoding" application mentioned in Section 6, one frequently faces, and wants to discard, polynomials f that do not have t distinct roots. One can usually save time by checking whether $x^q - x \bmod f = 0$ before applying a root-finding algorithm. There are other applications where one wants all the roots of a polynomial f that has no reason to have as many as $\deg f$ distinct roots; for such applications it is usually helpful to replace f with $\gcd\{f, x^q - x\}$.

There are other root-finding techniques (and polynomial-factorization techniques) that scale well to very large finite fields $\mathbb{F}_q$ when t remains small, such as Berlekamp's trace algorithm [6]. If t is as large as q then all of these techniques are obviously slower than multipoint evaluation with the additive FFT, but our experiments indicate that the t cutoff is above the range used in code-based signatures (see Section 7) and possibly within the range used in code-based encryption. Our main reason for not using these methods is that they involve many data-dependent conditional branches; as far as we can tell, all of these methods become much slower when the branches are eliminated.

There is a generalization of the additive FFT that replaces $x^2 - x$ with $x^t - x$ if q is a power of t. Gao and Mateer state this generalization only in the extreme case that $\lg q$ and $\lg t$ are powers of 2; we are exploring the question of whether the generalization produces speedups for other cases.

4 Syndrome computation: transposing the additive FFT

Consider the problem of computing the vector $(\sum_\alpha r_\alpha, \sum_\alpha r_\alpha \alpha, \ldots, \sum_\alpha r_\alpha \alpha^d)$, given a sequence of q elements $r_\alpha \in \mathbb{F}_q$ indexed by elements $\alpha \in \mathbb{F}_q$, where $q = 2^m$. This vector is called a "syndrome". One can compute $\sum_\alpha r_\alpha \alpha^i$ separately for each i with approximately $2dq$ field operations. We do better in this section by merging these computations across all the values of i.

$m = 11$	t	27	32	35	40	53	63	69	79		
	adds	5.41	5.60	5.75	5.99	6.47	6.69	6.84	7.11		
	mults	1.85	2.12	2.13	2.16	2.40	2.73	2.77	2.82		
$m = 12$	t	21	41	45	56	67	81	89	111	133	
	adds	5.07	6.01	6.20	6.46	6.69	7.04	7.25	7.59	7.86	
	mults	1.55	2.09	2.10	2.40	2.64	2.68	2.70	2.99	3.28	
$m = 13$	t	18	29	35	57	95	115	119	189	229	237
	adds	4.78	5.45	5.70	6.44	7.33	7.52	7.56	8.45	8.71	8.77
	mults	1.52	1.91	2.04	2.38	2.62	2.94	3.01	3.24	3.57	3.64

Table 3.1. Number of field operations/point in the additive FFT for various field sizes $q = 2^m$ and various parameters t. The total number of field additions is q times "adds"; the total number of field multiplications is q times "mults". For comparison, Horner's rule uses qt additions and qt multiplications; i.e., for Horner's rule, "adds" and "mults" are both t. Chien search also uses qt additions and qt multiplications.

Application to decoding. The standard Berlekamp decoding algorithm computes the syndrome shown above, and then solves a "key equation" to compute the error-locator polynomial mentioned in Section 3. When Berlekamp's algorithm is applied to decoding Goppa codes using a degree-t polynomial g as described in Section 6, the inputs r_α are a received word divided by $g(\alpha)^2$, and d is $2t-1$. Many other decoding algorithms begin with the same type of syndrome computation, often with d only half as large.

Note that there are only $n \leq q$ bits in the received word. The $(d+1)m = 2tm$ syndrome bits are $\mathbb{F}_2$-linear functions of these n input bits. Standard practice in the literature is to precompute the corresponding $2tm \times n$ matrix (or a $tm \times n$ matrix for Patterson's algorithm), and to multiply this matrix by the n input bits to obtain the syndrome. These $2tmn$ bits are by far the largest part of the McEliece/Niederreiter secret key. Our approach eliminates this precomputed matrix, and also reduces the number of bit operations once t is reasonably large.

Syndrome computation as the transpose of multipoint evaluation. Notice that the syndrome $(c_0, c_1, \ldots, c_d)$ is an $\mathbb{F}_q$-linear function of the inputs r_α. The syndrome-computation matrix is a "transposed Vandermonde matrix": the coefficient of r_α in c_i is α^i.

For comparison, consider the multipoint-evaluation problem stated in the previous section, producing $f(\alpha)$ for every $\alpha \in \mathbb{F}_q$ given a polynomial $f = c_0 + c_1 x + \cdots + c_d x^d$. The multipoint-evaluation matrix is a "Vandermonde matrix": the coefficient of c_i in $f(\alpha)$ is α^i.

To summarize, the syndrome-computation matrix is exactly the transpose of the multipoint-evaluation matrix. We show below how to exploit this fact to obtain a fast algorithm for syndrome computation.

Transposing linear algorithms. A *linear algorithm* expresses a linear computation as a labeled acyclic directed graph. Each edge in the graph is labeled by a constant (by default 1 if no label is shown), multiplies its incoming vertex by that constant, and adds the product into its outgoing vertex; some vertices with-

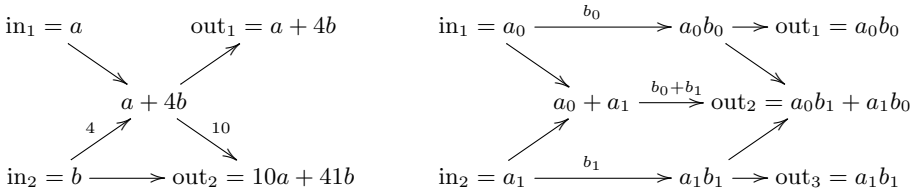

Fig. 4.1. An **R**-linear algorithm to compute $a, b \mapsto a + 4b, 10a + 41b$ using constants $4, 10$, and an $\mathbb{F}_{2^m}$-linear algorithm to compute $a_0, a_1 \mapsto a_0b_0, a_0b_1 + a_1b_0, a_1b_1$ using constants $b_0, b_0 + b_1, b_1$.

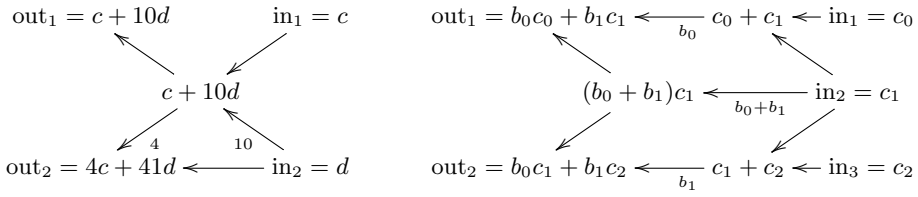

Fig. 4.2. Transposing the algorithms in Figure 4.1.

out incoming edges are labeled as inputs, and some vertices without outgoing edges are labeled as outputs. Figure 4.1 displays two examples: a computation of $a + 4b, 10a + 41b$ given a, b, using constants 4 and 10; and a computation of $a_0b_0, a_0b_1 + a_1b_0, a_1b_1$ given a_0, a_1, using constants $b_0, b_0 + b_1, b_1$.

The *transposition principle* states that if a linear algorithm computes a matrix M (i.e., M is the matrix of coefficients of the inputs in the outputs) then reversing the edges of the linear algorithm, and exchanging inputs with outputs, computes the transpose of M. This principle was introduced by Bordewijk in [20], and independently by Lupanov in [45] for the special case of Boolean matrices. This reversal preserves the number of multiplications (and the constants used in those multiplications), and preserves the number of additions plus the number of nontrivial outputs, as shown by Fiduccia in [29, Theorems 4 and 5] after preliminary work in [28].

For example, Figure 4.2 displays the reversals of the linear algorithms in Figure 4.1. The first reversal computes $c + 10d, 4c + 41d$ given c, d. The second reversal computes $b_0c_0 + b_1c_1, b_0c_1 + b_1c_2$ given c_0, c_1, c_2.

Transposing the additive FFT. In particular, since syndrome computation is the transpose of multipoint evaluation, reversing a fast linear algorithm for multipoint evaluation produces a fast linear algorithm for syndrome computation.

We started with our software for the additive FFT, including the improvements discussed in Section 3. This software is expressed as a sequence of additions

in $\mathbb{F}_q$ and multiplications by various constants in $\mathbb{F}_q$. We compiled this sequence into a directed acyclic graph, automatically renaming variables to avoid cycles. We then reversed the edges in the graph and converted the resulting graph back into software expressed as a sequence of operations in $\mathbb{F}_q$, specifically C code with vector intrinsics.

This procedure produced exactly the desired number of operations in $\mathbb{F}_q$ but was unsatisfactory for two reasons. First, there were a huge number of nodes in the graph, producing a huge number of variables in the final software. Second, this procedure eliminated all of the loops and functions in the original software, producing a huge number of lines of code in the final software. Consequently the C compiler, `gcc`, became very slow as m increased and ran out of memory around $m = 13$ or $m = 14$, depending on the machine we used for compilation.

We then tried the `qhasm` register allocator [8], which was able to produce working code for larger values of m using the expected number of variables (essentially q), eliminating the first problem. We then wrote our own faster straight-line register allocator. We reduced code size by designing a compact format for the sequence of $\mathbb{F}_q$ operations and interpreting the sequence at run time. There was, however, still some performance overhead for this interpreter.

We considered more advanced compilation techniques to reduce code size: the language introduced in [26], for example, and automatic compression techniques to recognize repeated subgraphs of the reversed graph. In the end we eliminated the compiler, analyzed the interaction of transposition with the structure of the additive FFT, and designed a compact transposed additive FFT algorithm.

The original additive FFT algorithm A has steps of the form B, A_1, A_2, C, where A_1 and A_2 are recursive calls. The transpose $A^\intercal$ has steps $C^\intercal, A_2^\intercal, A_1^\intercal, B^\intercal$, preserving the recursions. The main loop in the additive FFT takes a pair of variables v, w (containing $f_0(\alpha^2 - \alpha)$ and $f_1(\alpha^2 - \alpha)$ respectively), operates in place on those variables (producing $f(\alpha)$ and $f(\alpha + 1)$ respectively), and then moves on to the next pair of variables; transposition preserves this loop structure and simply transposes each operation. This operation replaces v by $v + w \cdot \alpha$ and then replaces w by $w + v$; the transposed operation replaces v by $v + w$ and then replaces w by $w + v \cdot \alpha$.

Improvement: transposed additive FFT on scaled bits. Recall that, in the decoding context, the inputs are not arbitrary field elements: r_α is a received bit divided by $g(\alpha)^2$. We take advantage of this restriction to reduce the number of bit operations in syndrome computation.

The first step of the transposed additive FFT operates on each successive pair of inputs v, w as described above: it replaces v by $v + w$ and then replaces w by $w + v \cdot \alpha$. Assume that before this v, w are computed as scaled bits $b_v \cdot s_v, b_w \cdot s_w$, where $b_v, b_w \in \mathbb{F}_2$ are variables and $s_v, s_w \in \mathbb{F}_q$ are constants. Computing $b_v \cdot s_v$ and $b_w \cdot s_w$ takes $2m$ bit operations; computing $w \cdot \alpha$ takes one field multiplication; computing $v + w \cdot \alpha$ takes m bit operations; computing $w + v$ takes m bit operations.

If the multiplication by α takes more than $2m$ bit operations then we do better by computing the final v and w directly as $b_v \cdot s_v + b_w \cdot s_w$ and $b_v \cdot s_v \cdot \alpha +$

$b_w \cdot s_w \cdot (\alpha + 1)$ respectively. This takes just $6m$ bit operations: we precompute $s_v, s_w, s_v \cdot \alpha, s_w \cdot (\alpha + 1)$.

The same idea can be used for more levels of recursion, although the number of required constants grows rapidly. Using this idea for all levels of recursion is tantamount to the standard approach mentioned earlier, namely precomputing a $2tm \times n$ matrix.

5 Secret permutations without secret array indices: odd-even sorting

Section 3 presented an algorithm that, given a polynomial f, outputs bits b_α for all $\alpha \in \mathbb{F}_q$ *in a standard order* (for example, lexicographic order using a standard basis), where $b_\alpha = 0$ if and only if $f(\alpha) = 0$. However, in the McEliece/Niederreiter context, one actually has the elements $(\alpha_1, \alpha_2, \ldots, \alpha_q)$ of $\mathbb{F}_q$ *in a secret order* (or, more generally, $(\alpha_1, \ldots, \alpha_n)$ for some $n \leq q$), and one needs to know for each i whether $f(\alpha_i) = 0$, i.e., whether $b_{\alpha_i} = 0$. These problems are not exactly the same: one must apply a secret permutation to the q bits output by Section 3. Similar comments apply to Section 4: one must apply the inverse of the same secret permutation to the q bits input to Section 4.

This section considers the general problem of computing a permuted q-bit string $b_{\pi(0)}, b_{\pi(1)}, \ldots, b_{\pi(q-1)}$, given a q-bit string $b_0, b_1, \ldots, b_{q-1}$ and a sequence of q distinct integers $\pi(0), \pi(1), \ldots, \pi(q-1)$ in $\{0, 1, \ldots, q-1\}$. Mapping the set $\{0, 1, \ldots, q-1\}$ to $\mathbb{F}_q$ in a standard order, and viewing α_{i+1} as either $\pi(i)$ or $\pi^{-1}(i)$, covers the problems stated in the previous paragraph.

The obvious approach is to compute $b_{\pi(i)}$ for $i = 0$, then for $i = 1$, etc. We require all load and store addresses to be public, so we cannot simply use the CPU's load instruction (with appropriate masking) to pick up the bit $b_{\pi(i)}$. Bitslicing can simulate this load instruction, essentially by imitating the structure of physical RAM hardware, but this is very slow: it means performing a computation involving every element of the array. We achieve much better bitslicing speeds by batching all of the required loads into a single large operation as described below.

Sorting networks. A "sorting network" uses a sequence of "comparators" to sort an input array S. A comparator is a data-independent pair of indices (i, j); it swaps $S[i]$ with $S[j]$ if $S[i] > S[j]$. This conditional swap is easily expressed as a data-independent sequence of bit operations: first some bit operations to compute the condition $S[i] > S[j]$, then some bit operations to overwrite $(S[i], S[j])$ with $(\min\{S[i], S[j]\}, \max\{S[i], S[j]\})$.

There are many sorting networks in the literature. We use a standard "odd-even" sorting network by Batcher [3], which uses exactly $(m^2 - m + 4)2^{m-2} - 1$ comparators to sort an array of 2^m elements. This is more efficient than other sorting networks such as Batcher's bitonic sort [3] or Shell sort [61]. The odd-even sorting network is known to be suboptimal when m is very large (see [2]), but we are not aware of noticeably smaller sorting networks for the range of m used in code-based cryptography.

Precomputed comparisons. We treat this section's $b_{\pi(i)}$ computation as a sorting problem: specifically, we use a sorting network to sort the key-value pairs $(\pi^{-1}(0), b_0), (\pi^{-1}(1), b_1), \ldots$ according to the keys. Note that computing $(\pi^{-1}(0), \pi^{-1}(1), \ldots)$ from $(\pi(0), \pi(1), \ldots)$ can be viewed as another sorting problem, namely sorting the key-value pairs $(\pi(0), 0), (\pi(1), 1), \ldots$ according to the keys.

We do better by distinguishing between the b-dependent part of this computation and the b-independent part of this computation: we precompute everything b-independent before b is known. In the context of code-based cryptography, the permutations π and π^{-1} are known at key-generation time and are the same for every use of the secret key. The only computations that need to be carried out for each decryption are computations that depend on b.

Specifically, all of the comparator conditions $S[i] > S[j]$ depend only on π, not on b; the conditional swaps of π values also depend only on π, not on b. We record the $(m^2 - m + 4)2^{m-2} - 1$ comparator conditions obtained by sorting π, and then apply those conditional swaps to the b array once b is known. Conditionally swapping $b[i]$ with $b[j]$ according to a bit c uses only 4 bit operations ($y \leftarrow b[i] \oplus b[j]$; $y \leftarrow cy$; $b[i] \leftarrow b[i] \oplus y$; $b[j] \leftarrow b[j] \oplus y$), for a total of $4((m^2-m+4)2^{m-2}-1)$ bit operations. Note that applying the same conditional swaps in reverse order applies the inverse permutation.

Permutation networks. A "permutation network" (or "rearrangeable permutation network" or "switching network") uses a sequence of conditional swaps to apply an arbitrary permutation to an input array S. Here a conditional swap is a data-independent pair of indices (i, j) together with a permutation-dependent bit c; it swaps $S[i]$ with $S[j]$ if $c = 1$.

A sorting network, together with a permutation, produces a limited type of permutation network in which the condition bits are computed by data-independent comparators; but there are other types of permutation networks in which the condition bits are computed in more complicated ways. In particular, the Beneš permutation network [4] uses only $2^m(m-1/2)$ conditional swaps to permute 2^m elements for $m \geq 1$.

The main challenge in using the Beneš permutation network is to compute the condition bits in constant time; see Section 6 for further discussion of timing-attack protection for key generation. We have recently completed software for this condition-bit computation but have not yet integrated it into our decoding software. We will report the details of this computation, and the resulting speeds, in an online update of this paper.

Alternative: random condition bits. In code-based cryptography we choose a permutation at random; we then compute the condition bits for a permutation network, and later (during each decryption) apply the conditional swaps. An alternative is to first choose a random sequence of condition bits for a permutation network, then compute the corresponding permutation, and later apply the conditional swaps.

This approach reduces secret-key size but raises security questions. By definition a permutation network can reach every permutation, but perhaps it is

much more likely to reach some permutations than others. Perhaps this hurts security. Perhaps not; perhaps a nearly uniform distribution of permutations is unnecessary; perhaps it is not even necessary to reach all permutations; perhaps a network half the size of the Beneš network would produce a sufficiently random permutation; but these speculations need security analysis. Our goals in this paper are more conservative, so we avoid this approach: we are trying to reduce, not increase, the number of questions for cryptanalysts.

6 A complete code-based cryptosystem

Code-based cryptography is often presented as encrypting fixed-length plaintexts. McEliece encryption multiplies the public key (a matrix) by a k-bit message to produce an n-bit codeword and adds t random errors to the codeword to produce a ciphertext. The Niederreiter variant (which has several well-known advantages, and which we use) multiplies the public key by a weight-t n-bit message to produce an $(n-k)$-bit ciphertext. If the t-error decoding problem is difficult for the public code then both of these encryption systems are secure against passive attackers who intercept valid ciphertexts for random plaintexts.

What users want, however, is to be able to encrypt *non-random* plaintexts of *variable length* and to be secure against *active* attackers who observe the receiver's responses to *forged* ciphertexts. The literature contains several different ways to convert the McEliece encryption scheme into this more useful type of encryption scheme, with considerable attention paid to

- the ciphertext overhead (ciphertext length minus plaintext length) and
- the set of attacks that are proven to be as difficult as the t-error decoding problem (e.g., generic-hash IND-CCA2 attacks in [42]).

However, much less attention has been paid to

- the cost in encryption time,
- the cost in decryption time, and
- security against timing attacks.

The work described in previous sections of this paper, speeding up t-error decoding and protecting it against timing attacks, can easily be ruined by a conversion that is slow or that adds its own timing leaks. We point out, for example, that straightforward implementations of any of the decryption procedures presented in [42] would abort if the "$\mathcal{D}^{\text{McEliece}}$" step fails; the resulting timing leak allows all of the devastating attacks that [42] claims to eliminate.

This section specifies a fast code-based public-key encryption scheme that provides high security, including security against timing attacks. This section also compares the scheme to various alternatives.

Parameters. The system parameters are positive integers m, q, n, t, k such that $n \leq q = 2^m$, $k = n - mt$, and $t \geq 2$. For example, one can take $m = 12$, $n = q = 4096$, $t = 41$, and $k = 3604$.

Key generation. The receiver's secret key has two parts: first, a sequence $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ of distinct elements of $\mathbb{F}_q$; second, a squarefree degree-t polynomial $g \in \mathbb{F}_q[x]$ such that $g(\alpha_1)g(\alpha_2)\cdots g(\alpha_n) \neq 0$. These can of course be generated dynamically from a much smaller secret.

The receiver computes the $t \times n$ matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix}$$

over $\mathbb{F}_q$. The receiver then replaces each entry in this matrix by a column of m bits in a standard basis of $\mathbb{F}_q$ over $\mathbb{F}_2$, obtaining an $mt \times n$ matrix H over $\mathbb{F}_2$. The kernel of H, i.e., the set of $c \in \mathbb{F}_2^n$ such that $Hc = 0$, is a vector space of dimension at least $n - mt = k$, namely the Goppa code $\Gamma = \Gamma_2(\alpha_1, \ldots, \alpha_n, g)$.

At this point one can compute the receiver's public key K by applying Gaussian elimination (with partial pivoting) to H. Specifically, K is the result of applying a sequence of elementary row operations to H (adding one row to another row), and is the unique result in systematic form, i.e., the unique result whose left $tm \times tm$ submatrix is the identity matrix. One can trivially compress K to $(n - mt)mt = k(n - k)$ bits by not transmitting the identity matrix; this compression was introduced by Niederreiter in [49], along with the idea of using a systematic parity-check matrix for Γ instead of a random parity-check matrix for Γ. If Gaussian elimination fails (i.e., if the left $tm \times tm$ submatrix of H is not invertible) then the receiver starts over, generating a new secret key; approximately 3 tries are required on average.

The standard approach to Gaussian elimination is to search for a 1 in the first column (aborting if there is no 1), then swap that row with the first row, then subtract that row from all other rows having a 1 in the first column, then continue similarly through the other columns. This approach has several timing leaks in the success cases. (It also takes variable time in the failure cases, but those cases are independent of the final secret.) We eliminate the timing leaks in the success cases as follows, with only a small constant-factor overhead. We add $1 - b$ times the second row to the first row, where b is the first entry in the first row; and then similarly (with updated b) for the third row etc. We then add b times the first row to the second row, where b is the first entry in the second row; and then similarly for the third row etc. We then continue similarly through the other columns.

An alternate strategy is to first apply a reasonably long sequence of elementary row operations to H, using a public sequence of rows but secret random multiples. Here "reasonably long" is chosen so that the output is negligibly different from a uniform random parity-check matrix for the same code. That parity-check matrix can safely be made public, so one can feed it to any Gaussian-elimination routine to obtain K, even if the Gaussian-elimination routine leaks information about its input through timing.

One can argue that key generation provides the attacker only a single timing trace (for the secret key that ends up actually being used), and that this single trace is not enough information to pinpoint the secret key. However, this argument relies implicitly on a detailed analysis of how much information the attacker actually obtains through timing. By systematically eliminating all timing leaks we eliminate the need for such arguments and analyses.

Encryption. To encrypt a variable-length message we generate a random 256-bit key for a stream cipher and then use the cipher to encrypt the message. AES-CTR has fast constant-time implementations for some platforms but not for others, so we instead choose Salsa20 [9] as the stream cipher. To eliminate malleability we also generate a random 256-bit key for the Poly1305 MAC [7], which takes time dependent only on the message length, and use this MAC to authenticate the ciphertext.

To generate these two secret keys we generate a random weight-t vector $e \in \mathbb{F}_2^n$ and then hash the vector to 512 bits. For the moment we use SHA-512 as the hash function; according to [17] it is still not yet clear exactly which Keccak variants will be specified for SHA-3. All of these hash functions take constant time for fixed n.

To transmit the vector e to the receiver we compute and send $w = Ke \in \mathbb{F}_2^{tm}$. The ciphertext overhead is tm bits for w, plus 128 bits for the authenticator.

Note that we are following Shoup's "KEM/DEM" approach (see [62]) rather than the classic "hybrid" approach. The hybrid approach (see, e.g., [51, Section 5.1]) is to first generate random secret keys, then encode those secret keys (with appropriate padding) as a weight-t vector e. The KEM/DEM approach is to first generate a weight-t vector e and then hash that vector to obtain random secret keys. The main advantage of the KEM/DEM approach is that there is no need for the sender to encode strings injectively as weight-t vectors, or for the receiver to decode weight-t vectors into strings. The sender does have to generate a random weight-t vector, but this is relatively easy since there is no requirement of injectivity.

A security proof for Niederreiter KEM/DEM appeared very recently in Persichetti's thesis [53]. The proof assumes that the t-error decoding problem is hard; it also assumes that a decoding failure for w is indistinguishable from a subsequent MAC failure. This requires care in the decryption procedure; see below.

Decryption. A ciphertext has the form (a, w, c) where $a \in \mathbb{F}_2^{128}$, $w \in \mathbb{F}_2^{tm}$, and $c \in \mathbb{F}_2^*$. The receiver decodes w (as discussed below) to obtain a weight-t vector $e \in \mathbb{F}_2^n$ such that $w = Ke$, hashes e to obtain a Salsa20 key and a Poly1305 key, verifies that a is the Poly1305 authenticator of c, and finally uses Salsa20 to decrypt c into the original plaintext.

Our decoding procedure is a constant-time sequence of bit operations and always outputs a vector e, even if w does not actually have the form Ke. With a small extra cost we also compute, in constant time, an extra bit indicating whether decoding succeeded. We continue through the hashing and authenticator verification in all cases, mask the authenticator-valid bit with the decoding-

succeeded bit, and finally return failure if the result is 0. This procedure rejects all forgeries with the same sequence of bit operations; there is no visible distinction between decoding failures and authenticator failures.

Finding a weight-t vector e given $w = Ke$ is the problem of syndrome decoding for K. We follow one of the standard approaches to syndrome decoding: first compute *some* vector $v \in \mathbb{F}_2^n$ such that $w = Kv$, and then find a codeword at distance t from v; this codeword must be $v - e$, revealing e. We use a particularly simple choice of v, taking advantage of K having systematic form: namely, v is w followed by $n - mt$ zeros. (This choice was recommended to us by Nicolas Sendrier; we do not know where it was first used in code-based cryptography.) This choice means that the receiver does not need to store K. We also point out that some of the conditional swaps in Section 5 are guaranteed to take $0, 0$ as input and can therefore be skipped.

There are two standard methods to find a codeword at distance t from v: Berlekamp's method [5] and Patterson's method [52]. To apply Berlekamp's method one first observes that $\Gamma = \Gamma_2(\alpha_1, \ldots, \alpha_n, g^2)$, and then that Γ is the $\mathbb{F}_2$-subfield subcode of the generalized Reed–Solomon code $\Gamma_q(\alpha_1, \ldots, \alpha_n, g^2)$. Berlekamp's method decodes generalized Reed–Solomon codes by computing a syndrome (Section 4), then using the Berlekamp–Massey algorithm to compute an error-locator polynomial, then computing the roots of the error-locator polynomial (Section 3).

Many authors have stated that Patterson's method is somewhat faster than Berlekamp's method. Patterson's method has some extra steps, such as computing a square root modulo g, but has the advantage of using g instead of g^2, reducing some computations to half size. On the other hand, Berlekamp's method has several advantages. First, as mentioned in Section 1, combinatorial list-decoding algorithms decode more errors, adding security for the same key size, by guessing a few error positions; in this case most decoding attempts fail (as in Section 7), and the analysis in [44] suggests that this makes Berlekamp's method faster than Patterson's method. Second, Berlekamp's method generalizes to *algebraic* list-decoding algorithms more easily than Patterson's method; see, e.g., [11]. Third, Berlekamp's method is of interest in a wider range of applications. Fourth, Berlekamp's method saves code size. Finally, Berlekamp's method is easier to protect against timing attacks.

7 New speed records for CFS signatures

CFS is a code-based public-key signature system proposed by Courtois, Finiasz, and Sendrier in [25]. The main drawbacks of CFS signatures are large public-key sizes and inefficient signing; the main advantages are short signatures, fast verification, and post-quantum security. This section summarizes the CFS signature system and reports our CFS speeds.

Review of CFS. System parameters are m, q, n, t, k as in Section 6, with two extra requirements: $n = q$, and g is irreducible. Key generation works as in the encryption scheme described in Section 6.

The basic idea of signing is simple. To sign a message M, first hash this message to a syndrome. If this syndrome belongs to a word at distance $\leq t$ from a codeword, use the secret decoding algorithm to obtain the error positions and send those positions as the signature. The verifier simply adds the columns of the public-key matrix indexed by these positions and checks whether the result is equal to the hash of M.

Unfortunately, a random syndrome has very low chance of being the syndrome of a word at distance $\leq t$ from a codeword. CFS addresses this problem using combinatorial list decoding: guess δ error positions and then proceed with decoding. If decoding fails, guess a different set of δ error positions. Finding a decodable syndrome requires many guesses; as shown in [25] the average number of decoding attempts is very close to $t!$. The decoding attempts for different guesses are independent; we can thus make efficient use of bitslicing in a *single* signature computation.

We actually use parallel CFS, a modification of CFS proposed by Finiasz in [30]. The idea is to compute λ different hashes of the message M and compute a CFS signature for each of these hashes. This increases the security level of CFS against a 2004 Bleichenbacher attack; see generally [51] and [30].

Previous CFS speeds. Landais and Sendrier in [44] describe a software implementation of parallel CFS with various parameters that target the 80-bit security level. Their best performance is for parameters $m = 20$, $t = 8$, $\delta = 2$ and $\lambda = 3$. With these parameters they compute a signature in 1.32 seconds on average on an Intel Xeon W3670 (Westmere microarchitecture) running at 3.2GHz, i.e., $4.2 \cdot 10^9$ cycles per signature on average.

New CFS software. Our CFS software uses the same set of parameters. For most of the computation we also use the same high-level algorithms as the software described in [44]: in particular, we use the Berlekamp–Massey algorithm to compute the error-locator polynomial f, and we test whether this polynomial splits into linear factors by checking whether $x^{2^m} \equiv x \pmod{f}$.

The most important difference in our implementation is the bitsliced field arithmetic. This has two advantages: it is faster and it does not leak timing information. Some parts of the computation are performed on only one stream of data (since we sign one message at a time), but even in those parts we continue using constant-time field arithmetic rather than the lookup-table-based arithmetic used in [44].

We do not insist on the entire signing procedure taking constant time, but we do guarantee that the signing time (and all lower-level timing information) is independent of all secret data. Specifically, to guarantee that an attacker has no information about the guessed error positions that did not allow successful decoding, we choose $\delta = 2$ *random* elements of $\mathbb{F}_{2^m}$ and compute the corresponding public-key columns, rather than running through guesses in a predictable order. These columns are at *some* positions in the public key; we compute these positions (in constant time) if decoding is successful.

There are three main bottlenecks in generating a signature:

- pick $e_1, e_2 \in \mathbb{F}_{2^m}$ at random and compute the corresponding public-key columns;
- use Berlekamp–Massey to obtain an error-locator polynomial f;
- test whether $x^{2^m} \equiv x \pmod f$.

Once such a polynomial f has been found, we multiply it by $(x - e_1)(x - e_2)$ to obtain a degree-10 error-locator polynomial. We then find all roots of this polynomial and output the set of corresponding support positions as the signature. We split the root-finding problem into 256 separate 2^{12}-point evaluation problems, again allowing fast constant-time bitsliced arithmetic for a *single* signature.

New CFS speeds. Our software signs in than $0.425 \cdot 10^9$ Ivy Bridge cycles on average; the median is $0.391 \cdot 10^9$ Ivy Bridge cycles. This cycle count is an order of magnitude smaller than the cycle count in [44]. We measured this performance across 100000 signature computations on random 59-byte messages on one core of an otherwise idle Intel Core i5-3210M with Turbo Boost and hyperthreading disabled.

It is common to filter out variations in cycle counts by reporting the median cycle count for many computations. Note, however, that the average is noticeably higher than the median for this type of random process. Similar comments apply to, e.g., RSA key generation.

Most of the $0.425 \cdot 10^9$ cycles are used by the three steps described above:

- picking e_1 and e_2 and computing the corresponding columns takes 52792 cycles for a batch of 256 iterations;
- the Berlekamp–Massey step takes 189900 cycles for a batch of 256 iterations;
- testing whether $x^{2^m} \equiv x \pmod f$ takes 436008 cycles for a batch of 256 iterations.

These computations account for $(52792 + 189900 + 436008)(t!\lambda + 128)/256 \approx 0.32 \cdot 10^9$ cycles on average. Root-finding, repeated λ times, accounts for another $0.05 \cdot 10^9$ cycles. A small number of additional cycles are consumed by hashing, converting to bitsliced form, multiplying the degree-8 error-locator polynomial f by $(x - e_1)(x - e_2)$, et al.

We also have extremely fast software for signature verification, taking only 2176 cycles. This count is obtained as the median of 1000 signature verifications for 59-byte messages. Furthermore we have software for Intel and AMD processors that do not feature the AVX instruction set and that instead uses SSE instructions on 128-bit vectors. This software generates a signature in $0.658 \cdot 10^9$ cycles on average and verifies a signature in only 2790 cycles on one core of an Intel Core 2 Quad Q6600 CPU.

References

[1] — (no editor), *AFIPS conference proceedings, volume 32: 1968 Spring Joint Computer Conference, Reston, Virginia*, Thompson Book Company, 1968.

[2] Miklós Ajtai, János Komlós, Endre Szemerédi, *An $O(n \log n)$ sorting network*, in STOC 1983 [38] (1983), 1–9.

[3] Kenneth E. Batcher, *Sorting networks and their applications*, in [1] (1968), 307–314.
[4] Václav E. Beneš, *Mathematical theory of connecting networks and telephone traffic*, Academic Press, 1965.
[5] Elwyn R. Berlekamp, *Algebraic coding theory*, McGraw-Hill, 1968.
[6] Elwyn R. Berlekamp, *Factoring polynomials over large finite fields*, Mathematics of Computation **24** (1970), 713–715.
[7] Daniel J. Bernstein, *The Poly1305-AES message-authentication code*, in FSE 2005 [34] (2005), 32–49.
[8] Daniel J. Bernstein, `qhasm` *software package* (2007). URL: http://cr.yp.to/qhasm.html.
[9] Daniel J. Bernstein, *The Salsa20 family of stream ciphers*, in [59] (2008), 84–97.
[10] Daniel J. Bernstein, *Batch binary Edwards*, in Crypto 2009 [35] (2009), 317–336.
[11] Daniel J. Bernstein, *Simplified high-speed high-distance list decoding for alternant codes*, in PQCrypto 2011 [67] (2011), 200–216.
[12] Daniel J. Bernstein, Johannes Buchmann, Erik Dahmen (editors), *Post-quantum cryptography*, Springer, 2009.
[13] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*, in CHES 2011 [57] (2011).
[14] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, accessed 10 June 2013 (2013). URL: http://bench.cr.yp.to.
[15] Daniel J. Bernstein, Tanja Lange, Christiane Peters, *Attacking and defending the McEliece cryptosystem*, in PQCrypto 2008 [23] (2008), 31–46.
[16] Daniel J. Bernstein, Peter Schwabe, *NEON crypto*, in CHES 2012 [58] (2012), 320–339.
[17] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, *Keccak and the SHA-3 standardization* (2013). URL: http://csrc.nist.gov/groups/ST/hash/sha-3/documents/Keccak-slides-at-NIST.pdf.
[18] Alex Biryukov, Guang Gong, Douglas R. Stinson (editors), *Selected areas in cryptography—17th international workshop, SAC 2010, Waterloo, Ontario, Canada, August 12–13, 2010, revised selected papers*, LNCS, 6544, Springer, 2011.
[19] Bhaskar Biswas, Nicolas Sendrier, *McEliece cryptosystem implementation: theory and practice*, in [23] (2008), 47–62.
[20] J. L. Bordewijk, *Inter-reciprocity applied to electrical networks*, Applied Scientific Research B: Electrophysics, Acoustics, Optics, Mathematical Methods **6** (1956), 1–74.
[21] Allan Borodin, Robert T. Moenck, *Fast modular transforms*, Journal of Computer and System Sciences **8** (1974), 366–386; older version, not a subset, in [48]. ISSN 0022–0000.
[22] Colin Boyd (editor), *Advances in cryptology—ASIACRYPT 2001, proceedings of the 7th international conference on the theory and application of cryptology and information security held on the Gold Coast, December 9–13, 2001*, LNCS, 2248, Springer, 2001.
[23] Johannes Buchmann, Jintai Ding (editors), *Post-quantum cryptography, second international workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17–19, 2008, proceedings*, LNCS, 5299, Springer, 2008.
[24] David G. Cantor, *On arithmetical algorithms over finite fields*, Journal of Combinatorial Theory, Series A **50** (1989), 285–300.
[25] Nicolas Courtois, Matthieu Finiasz, Nicolas Sendrier, *How to achieve a McEliece-based digital signature scheme*, in Asiacrypt 2001 [22] (2001), 157–174.

[26] Luca De Feo, Éric Schost, `transalpyne`*: a language for automatic transposition* (2010). URL: http://www.prism.uvsq.fr/~dfl/talks/plmms-08-07-10.pdf.

[27] Erwin Engeler, B. F. Caviness, Yagati N. Lakshman (editors), *Proceedings of the 1996 international symposium on symbolic and algebraic computation, ISSAC '96, Zurich, Switzerland, July 24–26, 1996*, Association for Computing Machinery, 1996.

[28] Charles M. Fiduccia, *On obtaining upper bounds on the complexity of matrix multiplication*, in [47] (1972), 31–40.

[29] Charles M. Fiduccia, *On the algebraic complexity of matrix multiplication*, Ph.D. thesis, Brown University, 1973.

[30] Matthieu Finiasz, *Parallel-CFS—strengthening the CFS McEliece-based signature scheme*, in SAC 2010 [18] (2011), 159–170.

[31] Steven Galbraith, Mridul Nandi (editors), *Progress in cryptology—Indocrypt 2012—13th international conference on cryptology in India, Kolkata, India, December 9–12, 2012, proceedings*, LNCS, 7668, Springer, 2012.

[32] Shuhong Gao, Todd Mateer, *Additive fast Fourier transforms over finite fields*, IEEE Transactions on Information Theory **56** (2010), 6265–6272.

[33] Joachim von zur Gathen, Jürgen Gerhard, *Arithmetic and factorization of polynomials over* $\mathbb{F}_2$ *(extended abstract)*, in ISSAC '96 [27] (1996), 1–9.

[34] Henri Gilbert, Helena Handschuh (editors), *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers*, LNCS, 3557, Springer, 2005.

[35] Shai Halevi (editor), *Advances in cryptology—CRYPTO 2009, 29th annual international cryptology conference, Santa Barbara, CA, USA, August 16–20, 2009, proceedings*, LNCS, 5677, Springer, 2009.

[36] Jens Hermans, Frederik Vercauteren, Bart Preneel, *Speed records for NTRU*, in CT-RSA 2010 [55] (2010), 73–88.

[37] Stefan Heyse, Tim Güneysu, *Towards one cycle per bit asymmetric encryption: code-based cryptography on reconfigurable hardware*, in CHES 2012 [58] (2012), 340–355.

[38] David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, Joel I. Seiferas (editors), *Proceedings of the 15th annual ACM symposium on theory of computing, 25–27 April, 1983, Boston, Massachusetts, USA*, Association for Computing Machinery, 1983.

[39] Richard M. Karp (chairman), *13th annual symposium on switching and automata theory*, IEEE Computer Society, 1972.

[40] Kwangjo Kim (editor), *Public key cryptography: proceedings of the 4th international workshop on practice and theory in public key cryptosystems (PKC 2001) held on Cheju Island, February 13–15, 2001*, LNCS, 1992, Springer, 2001.

[41] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 3rd edition, Addison-Wesley, 1997.

[42] Kazukuni Kobara, Hideki Imai, *Semantically secure McEliece public-key cryptosystems—conversions for McEliece PKC*, in PKC 2001 [40] (2001), 19–35.

[43] Grégory Landais, Nicolas Sendrier, *CFS software implementation* (2012); see also newer version [44].

[44] Grégory Landais, Nicolas Sendrier, *Implementing CFS*, in Indocrypt 2012 [31] (2012), 474–488; see also older version [43].

[45] O. B. Lupanov, *On rectifier and contact-rectifier circuits*, Doklady Akademii Nauk SSSR **111** (1956), 1171–1174. ISSN 0002–3264.

[46] Robert J. McEliece, *A public-key cryptosystem based on algebraic coding theory*, JPL DSN Progress Report (1978), 114–116.
[47] Raymond E. Miller, James W. Thatcher (editors), *Complexity of computer computations*, Plenum Press, 1972.
[48] Robert T. Moenck, Allan Borodin, *Fast modular transforms via division*, in [39] (1972), 90–96; newer version, not a superset, in [21].
[49] Harald Niederreiter, *Knapsack-type cryptosystems and algebraic coding theory*, Problems of Control and Information Theory **15** (1986), 159–166.
[50] Thomaz Oliveira, Juilo López, Diego F. Aranha, Francisco Rodríguez-Henríquez, *Two is the fastest prime* (2013). URL: http://eprint.iacr.org/2013/131.
[51] Raphael Overbeck, Nicolas Sendrier, *Code-based cryptography*, in [12] (2009), 95–145.
[52] Nicholas J. Patterson, *The algebraic decoding of Goppa codes*, IEEE Transactions on Information Theory **21** (1975), 203–207.
[53] Edoardo Persichetti, *Improving the efficiency of code-based cryptography*, Ph.D. thesis, University of Auckland, 2012.
[54] Christiane Peters, *Information-set decoding for linear codes over* $\mathbf{F}_q$, in PQCrypto 2010 [60] (2010), 81–94.
[55] Josef Pieprzyk (editor), *Topics in cryptology—CT-RSA 2010, the cryptographers' track at the RSA Conference 2010, San Francisco, CA, USA, March 1–5, 2010, proceedings*, LNCS, 5985, Springer, 2010.
[56] Josef Pieprzyk, Ahmad-Reza Sadeghi, Mark Manulis (editors), *Cryptology and network security—11th international conference, CANS 2012, Darmstadt, Germany, December 12–14, 2012, proceedings*, LNCS, 7712, Springer, 2012.
[57] Bart Preneel, Tsuyoshi Takagi (editors), *Cryptographic hardware and embedded systems—CHES 2011, 13th international workshop, Nara, Japan, September 28–October 1, 2011, proceedings*, LNCS, Springer, 2011.
[58] Emmanuel Prouff, Patrick Schaumont (editors), *Cryptographic hardware and embedded systems—CHES 2012—14th international workshop, Leuven, Belgium, September 9–12, 2012, proceedings*, LNCS, 7428, Springer, 2012.
[59] Matthew Robshaw, Olivier Billet (editors), *New stream cipher designs*, LNCS, 4986, Springer, 2008.
[60] Nicolas Sendrier (editor), *Post-quantum cryptography, third international workshop, PQCrypto, Darmstadt, Germany, May 25–28, 2010*, LNCS, 6061, Springer, 2010.
[61] Donald L. Shell, *A high-speed sorting procedure*, Communications of the ACM **2** (1959), 30–32.
[62] Victor Shoup, *A proposal for an ISO standard for public key encryption (version 2.1)* (2001). URL: http://www.shoup.net/papers.
[63] Falko Strenzke, *A timing attack against the secret permutation in the McEliece PKC*, in PQCrypto 2010 [60] (2010), 95–107.
[64] Falko Strenzke, *Timing attacks against the syndrome inversion in code-based cryptosystems* (2011). URL: http://eprint.iacr.org/2011/683.pdf.
[65] Falko Strenzke, *Fast and secure root finding for code-based cryptosystems*, in CANS 2012 [56] (2012), 232–246.
[66] Yao Wang, Xuelong Zhu, *A fast algorithm for Fourier transform over finite fields and its VLSI implementation*, IEEE Journal on Selected Areas in Communications **6** (1988), 572–577.
[67] Bo-Yin Yang (editor), *Post-quantum cryptography, fourth international workhop, PQCrypto, Taipei, Taiwan, November 29–December 02, 2011*, LNCS, 7071, Springer, 2011.