McBits Revisited

Tung Chou(✉)

Graduate School of Engineering, Osaka University Japan,
1-1, Yamadaoka, Suita, Osaka Prefecture 565-0871, Japan
blueprint@crypto.tw

Abstract. This paper presents a constant-time fast implementation for a high-security code-based encryption system. The implementation is based on the "McBits" paper by Bernstein, Chou, and Schwabe in 2013: we use the same FFT algorithms for root finding and syndrome computation, similar algorithms for secret permutation, and bitslicing for low-level operations. As opposed to McBits, where a high decryption throughput is achieved by running many decryption operations in parallel, we take a different approach to exploit the internal parallelism in one decryption operation for the use of more applications. As the result, we manage to achieve a slightly better decryption throughput at a much higher security level than McBits. As a minor contribution, we also present a constant-time implementation for encryption and key-pair generation, with similar techniques used for decryption.

Keywords: McEliece · Niederreiter · Bitslicing · Software implementation

1 Introduction

In recent years, due to the advance in quantum computing, cryptographers are paying more and more attention to post-quantum cryptography. In particular, NIST's call for proposal [16] serves as an announcement to declare that post-quantum cryptography is going to be reality, and the whole world needs to be prepared for that. Among other things, we need post-quantum public-key encryption schemes, and the most promising candidates today are from code-based cryptography and lattice-based cryptography.

In 1978, McEliece proposed his hidden-Goppa-code cryptosystem [13] as the first code-based encryption system. Until today, almost 40 years of research has been invested on cryptanalyzing the system, yet nothing has really shaken its security. It has thus become one of the most confidence-inspiring post-quantum encryption systems we have today, and it is important to evaluate how practical the system is for deployment.

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: a6d277b6724b21ae996418cbec02d682. Date: 2017.06.26.

W. Fischer and N. Homma (Eds.): CHES 2017, LNCS 10529, pp. 213–231, 2017.
DOI: 10.1007/978-3-319-66787-4_11

Table 1. Number of cycles for decoding for McBits and our software.

reference	m	n	t	bytes	sec	perm	synd	key eq	root	all	arch
McBits [3]	13	6624	115	958482	252	23140	83127	102337	65050	444971	IB
	13	6960	119	1046739	263	23020	83735	109805	66453	456292	IB
This paper	13	8192	128	1357824	297	3783	62170	170576	53825	410132	IB
						3444	36076	127070	34491	275092	HW

In 2013, Bernstein, Chou, and Schwabe published the "McBits" paper [3], which presents a software implementation of Niederreiter's dual form [15] of the McEliece cryptosystem. McBits features (1) a very high decoding (and thus decryption) throughput which is an order of magnitude faster than the previous implementation by Biswas and Sendrier [8], and (2) full protection against timing attacks. These features are achieved by bitslicing non-conventional algorithms for decoding: they use the Gao–Mateer additive FFT [11] for the root-finding, the corresponding "transposed" FFT for syndrome computation, and a sorting network for secret permutation.

The decryption throughput McBits achieves, however, relies on the assumption that there are many decryption operations that can be run at the same time. This is a reasonable assumption for some applications, but not for the all applications. The user would be glad to have an implementation that is capable of decrypting efficiently, even when there is only one decryption operation at the moment.

The main contribution of this paper is that we show the assumption is NOT a requirement to achieve a high decryption throughput. Even better, our software actually achieves a slightly better decryption throughput than McBits, at a much higher security level. To achieve this, we need to have a deep understanding about the data flow in each stage of decoding algorithm in order to figure out what kind of internal parallelism there is and how it can be exploited.

Speeds. The decoding speed of our software, as well as those for the highest-security parameters in [3, Table 1], are listed in Table 1. Most notations here are the same as in [3, Table 1]: we use m to indicate the field size 2^m, n to denote the code length, and t to denote the number of errors. "Bytes" is the size of public keys in bytes; "Sec" is the (pre-quantum) security level reported by the https://bitbucket.org/cbcrypto/isdfq script from Peters [17], rounded to the nearest integer. We list the cycle counts for each stage of the decoding process as in [3, Table 1]: "perm" for secret permutation, "synd" for syndrome computation, "key eq" for key-equation solving, and "root" for root finding. In [3, Table 1] there are two columns for "perm": one stands for the initial permutation and one stands for the final permutation, but the cycle counts are essentially the same (we pick the timing for the initial permutation). Note that the column "all", which serves as an estimation for the KEM decryption time, is computed as

$$\text{"perm"} \times 2 + \text{"synd"} \times 2 + \text{"key eq"} + \text{"root"} \times 2.$$

Table 2. Cycle counts for key generation, encryption (for 59-byte messages), and decryption.

key-generation	encryption	decryption	arch
1552717680	312135	492404	IB
1236054840	289152	343344	HW

This is different from the "total" column in [3, Table 1] for decoding time, which is essentially

$$\text{“perm”} \times 2 + \text{“synd”} + \text{“key eq”} + \text{“root”}.$$

The difference is explained in Sect. 6 in detail. "Arch" indicates the microarchitecture of the platform: "IB" for Ivy Bridge and "HW" for Haswell.

We comment that the way we exploit internal parallelism brings some overhead that can be avoided when using external parallelism. In general such an overhead is hard to avoid since the data flow of the algorithm is not necessarily friendly for bitslicing internally. This is exactly the main reason why our software is slower in "key eq" than McBits (a minor reason is that we are using a larger t). Despite the extra overhead, we still perform better when it comes to "synd" and "root". The improvement on "perm" is mainly because of our use of an asymptotically faster algorithm. Our "all" speed ends up being better than McBits. We emphasize that the timings for McBits are actually 1/256 of the timings for 256 parallel decryption operations, while the timings for our software involve only one decryption operation.

For completeness, we also implement the complete KEM/DEM-like ([19]) encryption system as described in [3, Sect. 6]. The corresponding cycle counts for key generation, encryption, and decryption are presented in Table 2.

For comparison with lattice-based cryptosystems, NTRU Prime [4], which appears to be the fastest high-security NTRU-type system (that has a constant-time implementation) at the moment, takes

- 1 multiplications in $\mathbb{F}_{9829}[x]/(x^{739} - x - 1)$ for encryption and
- 2 multiplications in $\mathbb{F}_{9829}[x]/(x^{739} - x - 1)$ plus
  1 multiplication in $\mathbb{F}_3[x]/(x^{739} - x - 1)$ for decryption,

where each multiplication in $\mathbb{F}_{9829}[x]/(x^{739} - x - 1)$ takes around 50000 Haswell cycles. As other lattice-based cryptosystems, NTRU Prime has a relatively small public key size of 1232 bytes. Our system has a ciphertext overhead of only 224 bytes, while NTRU Prime takes at least 1141 bytes.

Parameter Selection. As shown in Table 1, we implement one specific parameter set $(m, n, t) = (13, 8192, 128)$, with 1357824-byte public keys and a 2^{297} security level. We explain below the reasons to select this parameter set.

The Gao–Mateer additive FFT evaluates the input polynomial at a predefined $\mathbb{F}_2$-linear subspace of $\mathbb{F}_{2^m}$. The parameter n indicates the size of the list of

field elements that we need to evaluate at, so for $n = 2^m$ we can simply define the subspace as $\mathbb{F}_{2^m}$. In the case of $n < 2^m$, however, there is no way to define the subspace to fit arbitrary choice of the field elements (which is actually a part of the secret key), so the best we can do is still evaluate at the whole $\mathbb{F}_{2^m}$. In other words, having $n < 2^m$ would result in some redundant computation.

The parameter n also indicates the number of elements that we need to apply secret permutations on. The permutation algorithm we use, in its original form, requires that the number of elements to be a power of 2. The algorithm can be "truncated" to deal with an arbitrary number of elements, but this makes implementation difficult.

Having t close to the register size is convenient for bitslicing the FFT algorithms and the Berlekamp–Massey algorithm. We choose $t = 128$ to match the size of XMM registers in SSE-supporting architectures, as well as the size of the vector registers in the ARM-NEON architectures. Not having t close to the register size will not really affect the performance of FFTs: the algorithms are dominated by the t-irrelevant part as long as t is much smaller than 2^m. A bad value for t has more impact on the performance of the Berlekamp–Massey algorithm since we might waste many bits in the registers. Choosing $t = 128$ (after choosing $n = 2^m$) also forces the number of rows mt and number of columns $n - mt$ of the public-key matrix to be multiples of 128, which is convenient for implementing the encryption operation.

For the reasons stated above, some other nice parameters for (m, n, t) are

- $(12, 4096, 64)$ with 319488-byte public keys and a 2^{159} security level,
- $(12, 4096, 128)$ with 491520-byte public keys and a 2^{189} security level, and
- $(13, 8192, 64)$ with 765440-byte public keys and a 2^{210} security level.

We decided to select a parameter set that achieves at least a 2^{256} pre-quantum security level and thus presumably at least a 2^{128} post-quantum security level.

The reader might argue that such a high security level is not required for real applications. Indeed, even if quantum algorithms can take a square root on the security level, it still means that our system has a roughly 2^{150} post-quantum security level. In fact, we even believe that quantum algorithms will not be able to take a square root on the security: we believe there is a overhead of more than 2^{20} that needs to be added upon the square root. However, before the post-quantum security of our system is carefully analyzed, we think it is not a bad idea to implement a parameter set that is very likely to be an overkill and convince users that the system achieves a decent speed even in this case. Once careful analysis is done, our implementation can then be truncated to fit the parameters. The resulting implementation will have at least the same speed and a smaller key size.

Organization. The rest of this paper is organized as follows. Section 2 introduces the low-level building blocks used in our software. Section 3 describes how we implement the Beneš networks for secret permutations. Section 4 describes how we implement the Gao–Mateer FFT for root finding and the corresponding "transposed" FFT for syndrome computation. Section 5 introduces how we

implement the Berlekamp–Massey algorithm for key-equation solving. Finally, Sect. 6 introduces how the components in Sects. 3, 4, 5 are combined to form the complete decryption, as well as how key generation and encryption are implemented.

2 Building Blocks

This section describes the low-level building blocks used in our software. We will use these building blocks as black boxes in the following sections. The implementation techniques behind these building blocks are not new. In particular, this section presents (1) how to use bitslicing to perform several field operations in parallel and (2) how to perform bit-matrix transposition in software. Readers who are familiar with these techniques may skip this section.

Individual Field Operations. The finite field $\mathbb{F}_{2^{13}}$ is constructed as $\mathbb{F}_2[x]/(g)$, where $g = x^{13}+x^4+x^3+x+1$. Let $z = x+(g)$. Each field element $\sum_{i=0}^{12} a_i z^i$ can then be represented as the integer $(a_{12}a_{11}\cdots a_0)_2$ in software. Field additions are carried out by XORs between integers. Field multiplications are carried out by the following C function.

```
typedef uint16_t gf;
gf gf_mul(gf in0, gf in1)
{
        uint64_t i, tmp, t0=in0, t1=in1, t;
        tmp = t0 * (t1 & 1);
        for (i = 1; i < 13; i++) tmp ^= (t0 * (t1 & (1 << i)));
        t = tmp & 0x1FF0000;
        tmp ^= (t >> 9) ^ (t >> 10) ^ (t >> 12) ^ (t >> 13);
        t = tmp & 0x000E000;
        tmp ^= (t >> 9) ^ (t >> 10) ^ (t >> 12) ^ (t >> 13);
        return tmp & ((1 << 13)-1);
}
```

The squaring function is written in a similar way. Computing the inverse of a field element is carried out by raising the element to the power $2^{13}-2$ using 12 squarings and 4 multiplications.

Bitsliced Field Operations. The field multiplication function `gf_mul` and the field addition shown above are rather inefficient. The reason is that each logical instruction deals with only a small number of bits. For the algorithms used in our software, however, most of the time several field operations can be performed in parallel. We thus "bitslice" the field operations. The idea of bitslicing is to use bitwise logical operations to simulate w copies of a combinational circuit: the data for the ith copy is stored in the ith bits of the registers. In this way, the number of bits involved in each instruction can be improved to w. Bitslicing is also heavily used in [3]. We emphasize that for [3], the w copies are from w different decryption operations. For our software, the w copies are all from the same decryption operation.

```
void vec64_mul(uint64_t *h, uint64_t *f, uint64_t *g)
{
        int i, j;
        uint64_t r[2*13 - 1];
        for (i = 0; i < 2*13 - 1; i++)
                r[i] = 0;
        for (i = 0; i < 13; i++)
        for (j = 0; j < 13; j++)
                r[i+j] ^= r[i+j] ^ (f[i] & g[j]);
        for (i = 2*13-2; i >= 13; i--)
        {
                r[i - 9] ^= r[i];
                r[i - 10] ^= r[i];
                r[i - 12] ^= r[i];
                r[i - 13] ^= r[i];
        }
        for (i = 0; i < 13; i++) h[i] = r[i];
}
```

Fig. 1. The C function for bitsliced multiplications in $\mathbb{F}_{2^{13}}[x]/(x^{13}+x^4+x^3+x+1)$ using 64-bit words.

The function `vec64_mul` for bitsliced field multiplications using 64-bit words is shown in Fig. 1. One can of course use 128-bit or 256-bit words instead. According to Fog's well-known performance survey [10], on the Ivy Bridge architecture, the bitwise AND/XOR/OR instructions on the 128-bit registers (XMM registers) have a throughput of 3 per cycle, while for the 256-bit registers (YMM registers) the throughput is only 1. On Haswell, the instructions for the 256-bit registers have a throughput of 3 per cycle. We thus use the corresponding function `vec128_mul` for Ivy Bridge and use `vec256_mul` as much as possible for Haswell. Since both functions are heavily used in our software, they are written in `qhasm` [2] code for the best performance.

Many CPUs nowadays support the `pclmulqdq` instruction. The instruction essentially performs a multiplication between two 64-coefficient polynomials in $\mathbb{F}_2[x]$, so it can be used for field multiplications. Our multiplication function `vec256_mul` takes 138 Haswell cycles, which means a throughput of 1.86 field multiplications per cycle. The `pclmulqdq` instruction has a throughput of 1/2 on Haswell. We may perform 2 multiplications between 13-coefficient polynomials using one `pclmulqdq` instruction. However, non-bitsliced representations make it expensive to perform reductions modulo the irreducible polynomial g. On Ivy Bridge the throughput for `pclmulqdq` is only 1/8, which makes it even less favorable.

Transposing Bit Matrices. Bit-matrix transposition appears to be a well-known technique in computer programming. Perhaps due to the simplicity of the method, it is hard to trace who the credit belongs to. Below we give a brief review on the idea.

The task is to transpose a $w \times w$ bit matrix M, where w is a power of 2. The idea is to first divide the matrix into 4 $w/2 \times w/2$ submatrices, i.e., the left upper, right upper, left bottom, and right bottom submatrices. Then a "coarse-grained transposition" is performed on M, which simply interchanges the left bottom and right upper submatrices. Finally each block is transposed recursively, until we reach 1×1 matrices. The idea is depicted below.

$$M = \begin{pmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{pmatrix} \implies M' = \begin{pmatrix} M_{00} & M_{10} \\ M_{01} & M_{11} \end{pmatrix} \implies \begin{pmatrix} M_{00}^T & M_{10}^T \\ M_{01}^T & M_{11}^T \end{pmatrix} = M^T$$

The benefit of this approach is that it can be carried out efficiently in software. Suppose we are working on a w-bit machine, where the matrix is naturally represented as an array of w w-bit words in a row-major fashion. Observe that each of the first $w/2$ rows of M' is the concatenation of the first halves of two rows in M. Similarly, each of the second $w/2$ rows is the concatenation of the second halves of two rows in M. Therefore, each row in M' can be generated using a few logical operations. After this, in order to carry out operations in the recursive calls efficiently, the operations involving the upper two blocks can be handled together using logical operations on w-bit words. The same applies for the bottom two blocks. The C code for transposing 64×64 matrices is shown in Fig. 2.

```
const uint64_t mask[6][2] =
{
  {0X5555555555555555, 0XAAAAAAAAAAAAAAAA},
  {0X3333333333333333, 0XCCCCCCCCCCCCCCCC},
  {0X0F0F0F0F0F0F0F0F, 0XF0F0F0F0F0F0F0F0},
  {0X00FF00FF00FF00FF, 0XFF00FF00FF00FF00},
  {0X0000FFFF0000FFFF, 0XFFFF0000FFFF0000},
  {0X00000000FFFFFFFF, 0XFFFFFFFF00000000}
};
for (j = 5; j >= 0; j--)
{
  s = 1 << j;
  for (p = 0; p < 32/s; p++)
  for (i = 0; i < s; i++)
  {
    idx0 = p*2*s + i;
    idx1 = p*2*s + i + s;
    x = (in[idx0] & mask[j][0]) | ((in[idx1] & mask[j][0]) << s);
    y = ((in[idx0] & mask[j][1]) >> s) | (in[idx1] & mask[j][1]);
    in[idx0] = x;
    in[idx1] = y;
  }
}
```

Fig. 2. The C code for transposing 64×64 bit matrices. The matrix to be transposed is stored in the array `in`. The transposition is performed in-place.

The same technique can be easily generalized to deal with non-square matrices. Our software makes use of functions for transposing 64×128 and 128×64 matrices, where instructions such as `psrlq`, `psllq`, `psrld`, `pslld`, `psrlw`, and `psllw` are used to shift the 128-bit registers.

3 The Beneš Network

As described in [3], a "permutation network" uses a sequence of conditional swaps to apply an arbitrary permutation to an input array S. Each conditional swap is a permutation-independent pair of indices (i, j) together with a permutation-dependent bit c; it swaps $S[i]$ with $S[j]$ if $c = 1$. Our software uses a specific type of permutation network, called the Beneš network [1], to perform secret permutations for the code-based encryption system.

The McBits paper uses a "sorting network" for the same purpose but notes that it takes asymptotically more conditional swaps than the Beneš network: $O(n \log^2 n)$ versus $O(n \log n)$ for array size $n = 2^m$. We found that the Beneš network is more favorable for our implementation because it is easier to use the internal parallelism due to its simple structure. This section introduces the structure of the Beneš network, as well as how it is implemented in our software.

Conditional Swaps: Structure. The Beneš network for 2^m elements consists of a sequence of $2m - 1$ stages, where each stage consists of exactly 2^{m-1} conditional swaps. The set of index pairs for these 2^{m-1} conditional swaps is defined as

$$\left\{(\alpha \cdot 2^{s+1} + \beta,\ \alpha \cdot 2^{s+1} + 2^s + \beta) \ \mid\ 0 \le \alpha < 2^{m-1-s},\ 0 \le \beta < 2^s\right\},$$

where s is stage-dependent. The sequence of s is defined as

$$m-1, m-2, \ldots, 1, 0, 1, \ldots, m-2, m-1.$$

To visualise the structure, the size-16 Beneš network is depicted in Fig. 3.

The Beneš network is often defined in a recursive way, in which case the size-2^m Beneš network is viewed as the combination of the first and last stage, plus 2 size-2^{m-1} Beneš networks in the middle. Also note that in some materials the sequence of s is defined as

$$0, 1, \ldots, m-2, m-1, m-2, \ldots, 1, 0.$$

The two ways to define the sequence for s are equivalent up to a permutation of the array indices.

Conditional Swaps: Implementation. Consider the Beneš network for an array S of 2^m bits for some even m. We may consider S as a $m/2 \times m/2$ matrix M such that

$$M_{i,j} = S[i \cdot 2^{m/2} + j].$$

In each of the first and last $m/2$ stages, the index pairs always have an index difference that is a multiple of $2^{m/2}$. This implies that in each of these stages,

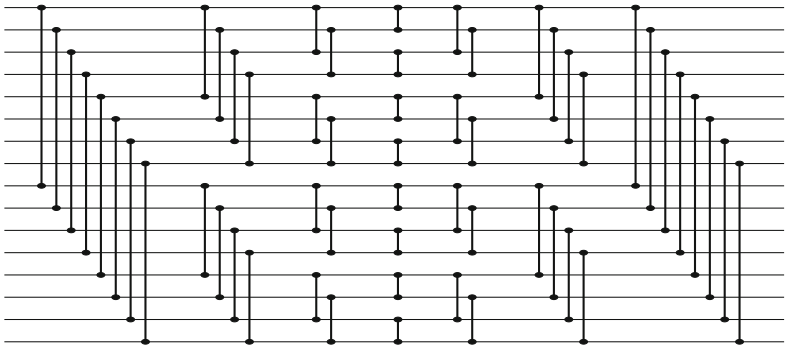

Fig. 3. The size-16 Beneš network with 7 stages. Each horizontal line represents an element in the array. Each vertical line segment illustrates a conditional swap involving the array elements corresponding to the end points.

$M_{i,j}$ is always conditionally swapped with $M_{i',j}$, where i' is a function of i. This implies that the conditional swaps can be carried out by performing bitwise logical operations between the rows (and the vectors formed by the corresponding conditions): a conditional swap between $M_{i,j}$ and $M_{i',j}$ with condition bit c can be carried out by 4 bit operations

$$(y \leftarrow M_{i,j} \oplus M_{i',j}; y \leftarrow cy; M_{i,j} \leftarrow M_{i,j} \oplus y; M_{i',j} \leftarrow M_{i',j} \oplus y),$$

as mentioned in [3]. Likewise, the $m-1$ stages in the middle can be carried out by using bitwise logical operations between columns.

The Beneš network can be easily implemented on a machine with $m/2$-bit registers. The matrix M can be represented using an array of $m/2$ $m/2$-bit words in a row-major fashion. With such a representation, the conditional swaps between the rows can be performed by bitwise logical instructions between the words. To deal with the $m-1$ stages in the middle, we transpose the bit matrix right after the first $m/2$ stages and right before the last $m/2$ stages (using the technique described in Sect. 2), to maintain a column-major representation of M during the $m-1$ stages.

For our system it is required to permute $2^m = 2^{13}$ bits. We store these bits in a 64×128 matrix, and the same technique described above still applies.

4 The Gao–Mateer Additive FFT

Given a predefined $\mathbb{F}_2$-linear basis $\{\beta_1, \beta_2, \ldots, \beta_k\} \subset \mathbb{F}_{2^m}$ and an ℓ-coefficient input polynomial $f = \sum_{i=0}^{\ell-1} f_i x^i \in \mathbb{F}_{2^m}[x]$ such that $\ell \leq 2^k \leq 2^m$, the Gao–Mateer FFT evaluates f at all the subset sums of the basis. In other words, the FFT outputs the sequence $f(e_1), f(e_2), \ldots, f(e_{2^k})$, where

$$(e_1, e_2, e_3, e_4, e_5, \ldots) = (0, \beta_1, \beta_2, \beta_1 + \beta_2, \beta_3, \ldots).$$

Such an FFT will be called a size-2^k FFT.

Assuming that $\beta_k = 1$. The idea is to compute two polynomials $f^{(0)}$ and $f^{(1)}$ such that

$$f = f^{(0)}(x^2 + x) + xf^{(1)}(x^2 + x),$$

using the "radix conversion" described in [3, Sect. 3] (this is called "Taylor expansion" in [11]). Note that $f^{(0)}$ is a $\lceil \ell/2 \rceil$-coefficient polynomial, while $f^{(1)}$ is a $\lfloor \ell/2 \rfloor$-coefficient polynomial. Observe that $\alpha^2 + \alpha = (\alpha + 1)^2 + (\alpha + 1)$. This implies that once $t_0 = f^{(0)}(\alpha^2 + \alpha)$ and $t_1 = f^{(1)}(\alpha^2 + \alpha)$ are computed, $f(\alpha)$ can be computed as $t_0 + \alpha \cdot t_1$, and $f(\alpha + 1)$ can be computed as $f(\alpha) + t_1$. Observe that the output of the FFT is the sequence

$$f(e_1), f(e_2), \ldots, f(e_{2^{k-1}}), \quad f(e_1 + 1), f(e_2 + 1), \ldots, f(e_{2^{k-1}} + 1),$$

and $e_1, \ldots, e_{2^{k-1}}$ forms all subset sums of $\{\beta_1, \ldots, \beta_{k-1}\}$. Therefore, two FFT recursive calls are carried out to evaluate $f^{(0)}$ and $f^{(1)}$ at all subset sums of $\{\beta_1^2 + \beta_1, \ldots, \beta_{k-1}^2 + \beta_{k-1}\}$. Finally, $f(e_i)$ and $f(e_i + 1)$ are computed by using $f^{(0)}(e_i^2 + e_i)$ and $f^{(1)}(e_i^2 + e_i)$ from the recursive calls, for all i from 1 to 2^{k-1}.

In the case where $\beta_k \neq 1$, the task is reconsidered as evaluating $f(\beta_k x)$ at the subset sums of $\{\beta_1/\beta_k, \beta_2/\beta_k, \ldots, 1\}$. This is called "twisting" in [3]. Note that it takes $\ell - 1$ multiplications to compute $f(\beta_k x)$. To sum up, the Gao–Mateer additive FFT consists of 4 steps: (1) twisting, (2) radix conversion, (3) two FFT recursive calls, and (4) combining outputs from the recursive calls.

In order to find the roots of an error locator, we need to evaluate at every field element in $\mathbb{F}_{2^{13}}$. The corresponding basis is defined as

$$\{\beta_1 = z^{12}, \beta_2 = z^{11}, \ldots, \beta_{13} = 1\}.$$

Having $\beta_{13} = 1$ means that the first twisting can be skipped. Since we use $t = 128$, the error locator for our system is a 129-coefficient polynomial. However, for implementation of the FFT algorithm it is more convenient to have a 128-coefficient input polynomial. We therefore consider the error locator as $x^{128} + f$ and compute $\alpha^{128} + f(\alpha)$ for all $\alpha \in \mathbb{F}_{2^{13}}$. Below we explain how the Gao–Mateer additive FFT for root finding, as well as the corresponding "transposed" FFT for syndrome computation, are implemented in our software.

Radix Conversions and Twisting. As described in [3], the first step of the radix conversion is to compute polynomials Q and R from the $4n$-coefficient (n is a power of 2) input polynomial $f = \sum_{i=0}^{4n-1} f_i x^i$:

$$\begin{aligned} Q &= (f_{2n} + f_{3n}) + \cdots + (f_{3n-1} + f_{4n-1})x^{n-1} + f_{3n}x^n + \cdots + f_{4n-1}x^{2n-1}, \\ R &= (f_0) + \cdots + (f_{n-1})x^{n-1} \\ &\quad + (f_n + f_{2n} + f_{3n})x^n + \cdots + (f_{2n-1} + f_{3n-1} + f_{4n-1})x^{2n-1}, \end{aligned}$$

so that $f = Q(x^{2n} + x^n) + R$. Then Q and R are fed into recursive calls to obtain the corresponding $R^{(0)}, R^{(1)}, Q^{(0)}, Q^{(1)}$. Finally, the routine outputs $f^{(0)} = R^{(0)} + x^n Q^{(0)}$ and $f^{(1)} = R^{(1)} + x^n Q^{(1)}$. The recursion ends when we reach a 2-coefficient polynomial $f_0 + f_1 x$, in which case $f^{(0)} = f_0$ and $f^{(1)} = f_1$.

Here is a straightforward way to implement the routine. First of all, represent the input polynomial f as a $4n$-element array `in` of datatype `gf` (see Sect. 2) such that f_i is stored in `in[i]`. Then perform $4n$ XORs

```
for (i = 0; i < n; i++) in[2*n+i] ^= in[3*n+i];
for (i = 0; i < n; i++) in[1*n+i] ^= in[2*n+i];
```

to store R_i in `in[i]` and Q_i in `in[2*n+i]`. Likewise, the additions in the recursive calls can be carried out by in-place XORs between array elements. Eventually we have $f_i^{(0)}$ in `in[2*i]` and $f_i^{(1)}$ in `in[2*i+1]`.

Representing the polynomials as arrays in `gf` is, however, expensive for twisting: as mentioned in Sect. 2, the function `gf_mul` is not efficient. Therefore in our software the polynomials are represented in bitsliced format. In this case, the additions can be simulated by using bitwise logical instructions and shifts. As a concrete example, let f be a 64-coefficient input polynomial in $\mathbb{F}_{2^{13}}[x]$, which is represented as a 13-element array of type `uint64_t`. Then the following code applies the radix conversion on f.

```
const uint64_t mask[5][2] =
{
  {0x8888888888888888, 0x4444444444444444},
  {0xC0C0C0C0C0C0C0C0, 0x3030303030303030},
  {0xF000F000F000F000, 0x0F000F000F000F00},
  {0xFF000000FF000000, 0x00FF000000FF0000},
  {0xFFFF000000000000, 0x0000FFFF00000000}
};
for (k = 4; k >= 0; k--)
for (i = 0; i < 13; i++)
{
  in[i] ^= (in[i] & mask[k][0]) >> (1 << k);
  in[i] ^= (in[i] & mask[k][1]) >> (1 << k);
}
```

In the end, the coefficients of $f^{(0)}$ are represented by the even bits of the words, while the coefficients of $f^{(1)}$ are represented by the odd bits.

The same technique can also be used to complete the radix conversions in the FFT recursive calls. Since a twisting operation simply multiplies f_i by β_k^i, they are carried out using bitsliced multiplications. See Fig. 4 for the code for all the radix conversions and twisting operations, including those in the FFT recursive calls. Note that the first twisting operation, which should take place before the first radix conversion, is already skipped in the code. Our software uses similar code but replaces 64-bit words by 128-bit words.

Butterflies. The reader might have noticed that the last 4 stages of Fig. 3 are similar to the well-known butterfly diagram for standard multiplicative FFTs. In a standard multiplicative FFT, f is written as $f^{(0)}(x^2) + xf^{(1)}(x^2)$ so that $f(\alpha)$ and $f(-\alpha)$ can be computed using $f^{(0)}(\alpha^2)$ and $f^{(1)}(\alpha^2)$ obtained from

```
for (j = 0; j <= 4; j++)
{
  for (i = 0; i < 13; i++)
  for (k = 4; k >= j; k--)
  {
    in[i] ^= (in[i] & mask[k][0]) >> (1 << k);
    in[i] ^= (in[i] & mask[k][1]) >> (1 << k);
  }
  vec64_mul(in, in, s[j]); // twisting
}
```

Fig. 4. The code for performing the twisting operations and radix conversion in the FFT for a 64-coefficient polynomial $f \in \mathbb{F}_{2^{13}}[x]$.

recursive calls. The similarity (between multiplicative FFTs and additive FFTs) in the ways of rewriting f results in the same "butterfly" structure.

In the case of a "full-size" additive FFT, where $\ell = 2^k$, the whole butterfly diagram has to be carried out. The technique used for carrying out the Beneš network (see Sect. 3) can be easily generalized to carry out the diagram. For decoding, however, ℓ is usually much smaller than $2^k = 2^m$. As the result, we only need to carry out the last $\log_2 \ell$ stages of the complete butterfly diagram.

As described in Sect. 3, we carry out the second half of the Beneš network by using a bit-matrix transposition in the middle. In the case of additive FFT butterflies, there will be m bit-matrix transpositions. The ideal case is that the ℓ is small enough so that the transpositions can be avoided. The corresponding code using 64-bit words for $m = 12$ is presented in Fig. 5. For the parameters $\ell = 128$ and $m = 13$, we are close to this ideal case but need to carry out 1 or 2 extra stages. The extra stages can be carried out by interleaving the 128-bit or 256-bit words.

```
for (i = 0; i <= 5; i++)
{
  s = 1 << i;
  for (j = 0; j < 64; j += 2*s)
  for (k = j; k < j+s; k++)
  {
    vec64_mul(tmp, out[k+s], consts[ consts_ptr + (k-j) ]);
    for (b = 0; b < 13; b++) out[k][b] ^= tmp[b];
    for (b = 0; b < 13; b++) out[k+s][b] ^= out[k][b];
  }
  consts_ptr += (1 << i);
}
```

Fig. 5. Butterflies in the additive FFT.

```
for (i = 5; i >= 0; i--)
{
  s = 1 << i;
  consts_ptr -= s;
  for (j = 0; j < 64; j += 2*s)
  for (k = j; k < j+s; k++)
  {
    for (b = 0; b < 13; b++) out[k][b] ^= out[k+s][b];
    vec64_mul(tmp, out[k], consts[ consts_ptr + (k-j) ]);
    for (b = 0; b < 13; b++) out[k+s][b] ^= tmp[b];
  }
}
                    :
                    :
                    :
for (j = 4; j >= 0; j--)
{
  vec64_mul(in, in, s[j]); // twisting
  for (k = j; k <= 4; k++)
  for (i = 0; i < 13; i++)
  {
    in[i] ^= (in[i] & (mask[k][1] >> (1 << k))) << (1 << k);
    in[i] ^= (in[i] & (mask[k][0] >> (1 << k))) << (1 << k);
  }
}
```

Fig. 6. Transposed FFT code with respect to Figs. 4 and 5.

The Bottom Level of Recursion. As shown in Fig. 4, when carrying out the radix conversions and twisting operations, we maintain a list of ℓ field elements. On the other hand, as shown in Fig. 5, when carrying out the FFT butterflies, we maintain a list of 2^m field elements. Apparently some operations are required to transit from the ℓ-element representation to the 2^m-element representation. This has to do with how the bottom level of recursion is defined.

The straightforward way to end the recursion is to check whether the input polynomial has only 1 coefficient; if so, the output is simply copies of the coefficient (the constant term). This is exactly the case for Figs. 4 and 5: after running the code in Fig. 4, we simply prepare the bitsliced representation of 64 copies of each elements and store them in `out`, and then Fig. 5 can be run to complete the FFT.

We do better by using the idea in [3, Sect. 3] to end the recursion when the input is a 2-coefficient polynomial. Let the input be $f = f_0 + f_1 x$ and the basis be $\{\beta_1, \ldots, \beta_k\}$. The idea is to first prepare a table containing $f_1\beta_i$ for all i, and then each output element can be computed using at most one field addition. To implement the idea, we perform the radix conversions and twisting operations

as in Fig. 4 but stop when we reach 2-coefficient polynomials. At this moment, the $\ell/2$ elements corresponding to f_0 would lie in the lower $\ell/2$ bits of the ℓ-bit words, while those for f_1 would lie in the higher $\ell/2$ bits. The outputs of the lowest-level FFTs can then be obtained by carrying out bitsliced multiplications and additions using bitwise logical operations between the $\ell/2$-bit words.

After this, we have the bitsliced representation (an array of m $\ell/2$-bit words) for the first output elements of the lowest level FFTs, the representation for the second output elements, and so on; in total there are $2^m/(\ell/2)$ such arrays. In order to group the output elements that belong to the same lowest-level FFT, we perform a sequence of m transpositions on $2^m/(\ell/2) \times (\ell/2) = 128 \times 64$ bit matrices, using the technique described in Sect. 2. Finally, the FFT butterflies can be performed using code similar to Fig. 5.

The Transposed Additive FFT. As described in [3, Sect. 4], a *linear algorithm* can be represented as a directed graph, and an algorithm that performs the transposed linear map can be obtained by reversing the edges in the graph. The way we implement the FFT makes it easy to imagine the structure of the graph and program the corresponding transposed FFT. As shown in Figs. 4 and 5, each inner loop in our FFT code essentially applies a simple linear operation on the values in `in` or `out`. In general it suffices to modify the loops to reverse the order that the inner loop is iterated and then replace the inner loop by its transpose. The transposed additive FFT code with respect to Figs. 4 and 5 is shown in Fig. 6 (the code for transposing the bottom level of recursion is skipped).

5 The Berlekamp-Massey Algorithm

The description of the original Berlekamp–Massey algorithm (BM) can be found in [12]. In each iteration of the algorithm, a field inversion has to be carried out. To perform the inversion in constant time, we may use the square-and-multiply algorithm, but this is rather expensive as discussed in Sect. 2. To avoid the problem, our implementation follows the inversion-free version of the algorithm as described in [21].

The algorithm begins with initializing polynomials $\sigma(x) = 1, \beta(x) = x \in \mathbb{F}_{2^m}[x]$, $\ell = 0 \in \mathbb{Z}$, and $\delta = 1 \in \mathbb{F}_{2^m}$. The input syndrome polynomial is denoted as $S(x) = \sum_{i=0}^{2t-1} S_i x^i$. Then in iteration k (from 0 to $2t-1$), the variables are updated using operations in Fig. 7. Note that ℓ and δ are just an integer and a field element, and multiplying a polynomial by x (to update $\beta(x)$) is rather cheap. Therefore the algorithm is bottlenecked by computing d and updating $\sigma(x)$. We explain below how the algorithm is implemented in our software.

General Implementation Strategy. Assume that there are $(t+1)$-bit general-purpose registers on the target machine. For example, one can assume that $t = 63$ and that we are working on a 64-bit machine. We store polynomials $\sigma(x)$ and $\beta(x)$ in the bitsliced format, each using an array of m $(t+1)$-bit words. The constant terms σ_0 and β_0 are stored in the most significant bits of the words; σ_1 and β_1 are stored in the second significant bits; and so on. We also use an array

$$d \leftarrow \sum_{i=0}^{t} \sigma_i S_{k-i}$$

$$\Big[\sigma(x),\ \beta(x),\ \ell,\ \delta\Big] \leftarrow \begin{cases} \Big[\delta\sigma(x) - d\beta(x),\ x\beta(x),\ \ell,\ \delta\Big], & d = 0 \text{ or } k < 2\ell. \\ \Big[\delta\sigma(x) - d\beta(x),\ x\sigma(x),\ k - \ell + 1,\ d\Big], & \text{otherwise.} \end{cases}$$

Fig. 7. Iteration k in the inversion-free BM.

S' of m (t+1)-bit words to store at most $t+1$ coefficients of $S(x)$. This array is maintained so that S_k is stored in the most significant bits of the words; S_{k-1} is stored in the second significant bits; and so on.

To compute d, we first perform a bitsliced field multiplication between $\sigma(x)$ and S'. The result is the bitsliced representation of $\sigma_0 S_k$, $\sigma_1 S_{k-1}$, ..., etc. The element d can then be computed as the parities of the m $(t+1)$-bit words. After this, S_{k+1} is inserted to the most significant bits of the words in S', which will be used in the next iteration.

To update $\sigma(x)$, we need to perform two scalar multiplications $\delta \cdot \sigma(x)$ and $d \cdot \beta(x)$. The bitsliced representations of $t+1$ copies of δ and d are first prepared, and then bitsliced multiplications are carried out to compute the products. Updating $\beta(x)$ is done by conditionally replacing the value of $\beta(x)$ by $\sigma(x)$ (which can be easily represented as logical operations) and then shifting each word to the right by one bit to simulate the multiplication by x.

The implementation strategy pretty much simulates the circuit presented in [21, Fig. 1]. Using the strategy, (each iteration of) the BM algorithm can be represented as a fixed sequence of instructions. In particular, the load and store instructions always use the same memory indices. As the result, the implementation is fully protected against timing attacks.

Haswell Implementation for $t = 128$. Exactly the same implementation strategy cannot be used for $t = 128$ on Haswell for there is no $(128+1)$-bit registers. To solve this problem, our strategy is to store σ_0 and S_k in two variables of datatype `gf`. The elements $\sigma_1, \ldots, \sigma_{128}$ and $S_{k-1}, \ldots, S_0$ are still stored in the bitsliced format, using two arrays of 128-bit words. To compute d, the product $\sigma_0 S_k$ is computed separately. Similarly, to update $\sigma(x)$, the product $\sigma_0 \delta$ is computed separately. Note that β_0 is always 0, so we simply store $\beta_1, \ldots, \beta_{128}$ in the bitsliced format.

We also need a way to update S' and $\beta(x)$ without generic shift instructions for 128-bit registers. Our solution is to make use of the `shrd` instruction. Given 64-bit registers r_1, r_0 as arguments, the `shrd` instruction is able to shift the least significant bit of r_1 into the most significant bit of r_0. Therefore, with 2 `shrd` instructions, we can shift a 128-bit word by one bit to the right. In particular, the second `shrd` shifts one bit into the most significant bit of the 128-bit word.

Therefore, we update S' by setting this bit to bits of S_k and update β by setting this bit to 0 or bits of σ_0 (depending on the condition).

To optimize the speed for Haswell, we combine the two `vec128_mul` function calls for $\delta \cdot \sigma(x)$ and $d \cdot \beta(x)$ to form one `vec256_mul`. As discussed in Sect. 2, this is better because 256-bit logical instructions have the same throughput as the 128-bit ones.

We also use 256-bit logical instructions to accelerate `vec128_mul`. A field multiplication can be viewed as a multiplication between 13-coefficient polynomials, followed by a reduction modulo g. Let the polynomials be f and f'; the idea is to split the polynomial multiplication into two parts $f(f'_0 + \cdots + f'_6 x^6)$ and $f(f'_7 + \cdots + f'_{12} x^5 + 0x^6)$. In this way, we create two bitsliced multiplications for computing d, and the two can be combined as what we do for $\delta \cdot \sigma(x)$ and $d \cdot \beta(x)$. Note that for combining the two products and the reduction part we still use 128-bit logical instructions. By using 256-bit logical instructions, we improve the cycle counts of `vec128_mul` from 137 to 94 Haswell cycles.

As a minor optimization, we also combine the computation of $\sigma_0 S_k$ and $\sigma_0 \delta$. This is achieved by using the upper 32 bits of the 64-bit variables in `gf_mul` for another multiplication. In this way, two field multiplications can be carried out in roughly the same time as `gf_mul`.

As discussed in Sect. 4, the input of the FFT function for root finding is the bitsliced representation of $f_0, \ldots, f_{127}$; f_{128} is not stored since it is assumed to be 1. In fact, at the end of the Berlekamp–Massey algorithm we have $f_i = \sigma_{128-i}$. Therefore we perform a field inversion for σ_0 and bitsliced multiplications to force a monic output polynomial for the Berlekamp–Massy algorithm.

6 The Complete Cryptosystem

In [3, Sect. 6] a complete public-key encryption system is described. The cryptosystem uses a KEM/DEM-like structure, where the KEM is based on the Niederreiter cryptosystem. To send a message, the sender first uses the receiver's Niederreiter public key to compute the syndrome of a random weight-t error vector. Then the error vector is hashed to obtain two symmetric keys. The first symmetric key is used for a stream cipher to encrypt the arbitrary-length message. The second symmetric key is used for a message authentication code to authenticate the output generated by the stream cipher. The syndrome, the stream-cipher output, and the authentication tag are then sent to the receiver.

The receiver first decodes the syndrome using the Niederreiter secret key. The resulting error vector is then hashed to obtain the symmetric keys, and the receiver verifies (using the tag) and decrypts the stream-cipher output. Note that the receiver can fail in decoding or verification. The decryption algorithm should be carefully implemented such that others cannot distinguish (for example, by using timing information) what kind of failure the receiver encounters.

We show below how key-pair generation, KEM encryption, and KEM decryption are implemented in our software.

Private-Key Generation. The private key of the system consists of two parts: (1) a sequence $(\alpha_1, \ldots, \alpha_n)$ of n distinct elements in $\mathbb{F}_{2^m}$ and (2) a square-free degree-t polynomial $g \in \mathbb{F}_{2^m}[x]$ such that $g(\alpha_i) \neq 0$ for all i.

For our implementation, g is generated as a uniform random degree-t monic irreducible polynomial in $\mathbb{F}_{2^m}[\mathrm{x}]$. To generate g, we first generate a random element α in the extension field $\mathbb{F}_{2^{mt}}$. The polynomial g is then defined as the minimal polynomial of α in $\mathbb{F}_{2^m}[\mathrm{x}]$, if the degree is t. To find the minimal polynomial, we view $\mathbb{F}_{2^{mt}}$ as the vector space $(\mathbb{F}_{2^m})^t$ and try to find linear dependency between $1, \alpha, \alpha^2, \ldots, \alpha^t$ using Gaussian elimination. A description of the algorithm can be found in, for example, [20, Sect. 17.2].

The benefit of this approach is that it is easy to make Gaussian elimination constant-time: [3] already shows how this can be achieved in the case of bit matrices. Note that the algorithm can fail to find a degree-t irreducible polynomial when $\alpha \in \mathbb{F}_{2^{mt'}}$ such that t' is a divisor of t. For our parameters $m = 13$ and $t = 128$ the probability of failure is only 2^{-832}.

Recall that we use $n = 2^m$. Let ϕ be a permutation function such that $\phi(e_1, \ldots, e_{2^m}) = (\alpha_1, \ldots, \alpha_{2^m})$, where $(e_1, \ldots, e_{2^m})$ is the stardard order of field elements introducted by the FFT (see Sect. 4). In our software, the permutation function is defined using the condition bits in the corresponding size-2^m Beneš network. Instead of generating the sequence α_i and then figure out the condition bits, the condition bits are generated as random bits in the current implementation; the reader may refer to [3, Sect. 5] for a brief discussion on this approach. We comment that there are $(2m-1)2^{m-1} = m2^m - 2^{m-1}$ condition bits in the Beneš network, while a list of 2^m field elements takes $m2^m$ bits. In other words, representing $(\alpha_1, \ldots, \alpha_n)$ as condition bits actually saves the size of secret keys.

Public-Key Generation. Let H be the bit matrix obtained by replacing each entry in the matrix

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

by a column of m bits from the standard-basis representation. The receiver computes the row-reduced echelon form of H. If the result is of the form $\left[I|H'\right]$, the public-key is set to H'; otherwise a new secret key is generated.

In our implementation, the images $g(e_1), \ldots, g(e_n)$ are first generated using the FFT implementation described in Sect. 4. After this, the inversions of all these images are computed, using Montgomery's trick [14] with bitsliced field multiplications. Now we have the bitsliced representation of the first row of the matrix

$$\begin{pmatrix} 1/g(e_1) & 1/g(e_2) & \cdots & 1/g(e_n) \\ e_1/g(e_1) & e_2/g(e_2) & \cdots & e_n/g(e_n) \\ \vdots & \vdots & \ddots & \vdots \\ e_1^{t-1}/g(e_1) & e_2^{t-1}/g(e_2) & \cdots & e_n^{t-1}/g(e_n) \end{pmatrix}.$$

The remaining rows are then computed one-by-one using bitsliced field multiplications. Since all the rows are represented in the bitsliced format, the matrix can be easily viewed as the corresponding $mt \times n$ bit matrix. Then the Beneš network is applied to each row of the bit matrix to obtain H. Finally we follow [3, Sect. 6] to perform a constant-time Gaussian elimination. The public key is then the row-major representation of H' (one can of course use a column-major representation instead).

KEM Encryption. The KEM encryption begins with generating the error vector e of weight t. This is carried out by first generating a sequence of t random m-bit values, which indicates the positions of the errors. The t values are then checked for repetition. If a repetition is found, we simply regenerate the t random m-bit values; otherwise, we convert the indices into the error vector as a sequence of $n/8$ bytes.

To compute each bit of the syndrome, each 128-bit word in the corresponding row is first ANDed with the corresponding 128-bit word in the error vector. The 128-bit results are then XORed together to form one single 128-bit word. We make use of the `popcnt` instruction to compute the parity of the 128-bit word, and the syndrome bit is set to the parity. Finally, after processing all the rows of the public key, we deal with the identity matrix by XORing the first $mt/8$ bytes of the error vector into the syndrome.

KEM Decryption. As explained in [3], decoding consists of 5 stages: the initial permutation, syndrome computation, key-equation solving, root finding, and the final permutation. This is why the "total" column in [3, Table 1] is essentially

$$\text{“perm”} \times 2 + \text{“synd”} + \text{“key eq”} + \text{“root”}.$$

The "all" column in Table 1, however, is computed as

$$\text{“perm”} \times 2 + \text{“synd”} \times 2 + \text{“key eq”} + \text{“root”} \times 2.$$

In other words, we count one extra "root" and one extra "synd".

The reason we count "root" one more time is a matter of implementation choice. To perform syndrome computation, each of the 2^m input bits is required to be scaled by $1/g(\alpha)^2$, where α is the corresponding point for evaluation. Since $1/g(\alpha)^2$ depends only on g, [3] uses them as pre-computed values. This strategy saves time but enlarges the size of secret keys. We decide to save the size of secret keys and compute all $1/g(\alpha)^2$ on the fly, using "root" for computing $g(\alpha)$, Montgomery's trick for simultaneous inversions [14] with bitsliced multiplications, and bitsliced squarings.

The reason we count "synd" one more time is for re-encryption. A decoding algorithm is only required to decode when the input syndrome corresponds to an error vector of weight t. For KEM, however, we need additionally the ability to reject invalid inputs. We therefore check the weight of the error vector and perform "synd" again to compute the syndrome of the error vector. The decoding is considered successful only if the weight is exactly t and the syndrome matches the output of the first "synd" stage.

References

1. Beneš, V.E.: Mathematical Theory of Connecting Networks and Telephone Traffic. Academic Press, Cambridge (1965). §3
2. Bernstein, D.J.: qhasm software package (2007). http://cr.yp.to/qhasm.html. §2
3. Bernstein, D.J., Chou, T., Schwabe, P.: McBits: fast constant-time code-based cryptography. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 250–272. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40349-1_15. §1, §1, §1, §1, §1, §1, §1, §1, §2, §2, §3, §3, §4, §4, §4, §4, §4, §6, §6, §6, §6, §6, §6, §6
4. Bernstein, D.J., Chuengsatiansup, C., Lange, T., van Vredendaal, C.: NTRU Prime (2016). https://eprint.iacr.org/2016/461.pdf. §1
5. Bertoni, G., Coron, J.-S. (eds.): CHES 2013. LNCS, vol. 8086. Springer, Heidelberg (2013). See [3]
6. Biham, E. (ed.): FSE 1997. LNCS, vol. 1267. Springer, Heidelberg (1997). See [7]
7. Biham, E.: A fast new DES implementation in software, in [6], pp. 260–272 (1997)
8. Biswas, B., Sendrier, N.: McEliece cryptosystem implementation: theory and practice, in [9], pp. 47–62 (2008). §1
9. Buchmann, J., Ding, J. (eds.): Post-Quantum Cryptography. LNCS, vol. 5299. Springer, Heidelberg (2008). See [8]
10. Agner Fog: Instruction tables (2016). http://www.agner.org/optimize/instruction_tables.pdf. §2
11. Gao, S., Mateer, T.: Additive fast Fourier transforms over finite fields. IEEE Trans. Inf. Theory **56**, 6265–6272 (2010). http://www.math.clemson.edu/sgao/pub.html. §1, §4
12. Massey, J.L.: Shift-register synthesis and BCH decoding. IEEE Trans. Inf. Theory **15**, 122–127 (1969). §5
13. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory, JPL DSN Progress report, pp. 114–116 (1978). http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF. §1
14. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathe. Comput. **48**, 243–264 (1987). http://www.jstor.org/stable/pdf/2007888.pdf. §6, §6
15. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Probl. Control Inf. Theory **15**, 159–166 (1986). §1
16. NIST: Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process (2016). http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf. §1
17. Peters, C.: Information-set decoding for linear codes over $\mathbf{F}_q$. In: Sendrier, N. (ed.) PQCrypto 2010 [18]. LNCS, vol. 6061, pp. 81–94. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12929-2_7. §1
18. Sendrier, N. (ed.): PQCrypto 2010. LNCS, vol. 6061. Springer, Heidelberg (2010). See [17]
19. Shoup, V.: A proposal for an ISO standard for public key encryption (version 2.1). http://www.shoup.net/papers. §1
20. Shoup, V. (ed.): A Computational Introduction to Number Theory and Algebra (Version 2). Cambridge University Press, Cambridge (2015). §6
21. Youzhi, X.: Implementation of Berlekamp-Massey algorithm without inversion. IEE Proc. I Commun. Speech Vision **138**, 138–140 (1991). §5, §5