Designing efficient dyadic operations for cryptographic applications

Designing Efficient Dyadic Operations for Cryptographic Applications

Gustavo Banegas[1], Paulo S. L. M. Barreto[2], Edoardo Persichetti[3] and Paolo Santini[4]

[1] *Technische Universiteit Eindhoven, the Netherlands*
[2] *Institute of Technology, University of Washington at Tacoma, USA*
[3] *Department of Mathematical Sciences, Florida Atlantic University, USA*
[4] *Università Politecnica delle Marche, Ancona, Italy*

Abstract

Cryptographic primitives from coding theory are some of the most promising candidates for NIST's Post-Quantum Cryptography Standardization process. In this paper, we introduce a variety of techniques to improve operations on dyadic matrices, a particular type of symmetric matrices that appear in the automorphism group of certain linear codes. Besides the independent interest, these techniques find an immediate application in practice. In fact, one of the candidates for the Key Exchange functionality, called DAGS, makes use of quasi-dyadic matrices to provide compact keys for the scheme.

Keywords: post-quantum cryptography, code-based cryptography, dyadic matrices.

1 Introduction

Post-Quantum Cryptography is the area of research that investigates cryptographic primitives that are deemed secure against attackers equipped with quantum technology. These include schemes based on a variety of mathematical problems, such as finding short vectors in a lattice, or decoding random linear codes. The latter is known as Code-based Cryptography and it is relies more or less directly on the Syndrome Decoding Problem [4], which shows no vulnerabilities to quantum attacks. The first code-based scheme was introduced by McEliece in 1978 [10] and has resisted cryptanalysis, in its original form, for nearly 40 years.

McEliece's cryptosystem has often been ignored in favor of schemes based on number theory problems (such as RSA or El Gamal), mainly due to the size of its public key, which was deemed too large for practical use (especially at the time). However, Shor's algorithm [13] shows that, once quantum computers of an appropriate size are available, the cryptosystems currently in use will become obsolete. It is therefore important to offer a credible alternative to current cryptography, and, with this in mind, NIST has recently launched a call for papers to standardize the public-key primitives of the future [1].

Among the code-based candidates for NIST's call, DAGS [3] is a Key Encapsulation Mechanism (KEM) that uses Quasi-Dyadic (QD) matrices to considerably reduce the size of the public key,

following a McEliece-like approach. The proposal builds on a line of work initiated by Misoczki and Barreto [11] and subsequently developed by Persichetti in [6, 12].

Our Contribution. We analyze two separate aspects of dyadic operations. First, we present three different algorithms for that are aimed specifically at computing multiplication of dyadic matrices. These are, respectively, a "standard" approach that makes use of dyadic signatures, a specialized Karatsuba-like algorithm, and a procedure based on the Fast Walsh-Hadamard Transform (FWHT) [7], also called dyadic convolution. We analyze the performance of all three methods and report our timings.
As a second contribution, we describe a procedure that applies the LUP decomposition [5] to the dyadic case. The method effectively factors every quasi-dyadic matrix into a product of two triangular matrices and a permutation matrix. This leads to the possibility of a very efficient algorithm for computing the inverse of a matrix, which is particularly useful in code-based cryptography, for instance for computing the systematic form of a parity-check (or generator) matrix. According to our measurements, this improved inversion procedure is extremely fast, and provides a very large speedup during DAGS Key Generation.

Organization of the Paper. This paper is organized as follows. We start with some preliminary definitions in Section 2. We then present our main contributions: the various multiplication techniques are described in Section 4 and the improved inversion algorithm is presented in Section 3. We conclude by showing the results obtained when applying our techniques to DAGS; this is done in Section 5.

2 Preliminaries

We introduce dyadic matrices and describe some of their general properties.

Definition 2.1 *Given a ring $\mathcal{R}$ and a vector $\mathbf{h} = (h_0, h_1, \cdots, h_{n-1}) \in \mathcal{R}^n$, with $n = 2^r$ for some $r \in \mathbb{N}$, the* dyadic *matrix $\mathbf{\Delta}(\mathbf{h}) \in \mathcal{R}^{n\times n}$ is the symmetric matrix with components $\Delta_{ij} = h_{i\oplus j}$ where $\oplus$ stands for bitwise exclusive-or. Such a matrix is said to have* order r. *The sequence $\mathbf{h}$ is called* signature *of the matrix $\mathbf{\Delta(h)}$, and corresponds to its first row. The set of dyadic $n \times n$ matrices over $\mathcal{R}$ is denoted $\mathbf{\Delta}(\mathcal{R}^n)$.*

One can alternatively characterize a dyadic matrix recursively: any 1×1 matrix is dyadic of order 0, and any dyadic matrix $\mathbf{M}$ of order $r > 0$ has the form

$$\mathbf{M} = \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{pmatrix} \tag{1}$$

where A and B are two dyadic matrices of order $r-1$. In other words, $\mathbf{\Delta}(\mathcal{R}^n) = \mathbf{\Delta}\left(\mathbf{\Delta}\left(\mathcal{R}^{n/2}\right)\right)$.

Definition 2.2 *A* dyadic permutation *is a dyadic matrix $\mathbf{\Pi}^i \in \mathbf{\Delta}(\{0,1\}^n)$ characterized by the signature $\boldsymbol{\pi}^i = (\delta_{ij} \mid j = 0, \ldots, n-1)$, where δ_{ij} is the Kronecker delta (hence $\boldsymbol{\pi}^i$ corresponds to the i-th row or column of the identity matrix).*

A dyadic permutation is clearly an involution, i.e. $(\mathbf{\Pi}^i)^2 = \mathbf{I}$. The i-th row, or equivalently the i-th column, of the dyadic matrix defined by a signature $\mathbf{h}$ can be written as $\mathbf{\Delta}(\mathbf{h})_i = \mathbf{h}\mathbf{\Pi}^i$.

A dyadic matrix can be efficiently represented by its signature; in particular, all the operations between dyadic matrices can be referred only to the corresponding signatures. Indeed, for any two length-n vectors $\mathbf{a}, \mathbf{b} \in \mathcal{R}$, we have:

$$\mathbf{\Delta}(\mathbf{a}) + \mathbf{\Delta}(\mathbf{b}) = \mathbf{\Delta}(\mathbf{a} + \mathbf{b}) \tag{2}$$

which means that, given two dyadic matrices $\mathbf{A}$ and $\mathbf{B}$, with respective signatures $\mathbf{a}$ and $\mathbf{b}$, their sum is the dyadic matrix described by the signature $\mathbf{a} + \mathbf{b}$.

In an analogous way, the multiplication between dyadic matrices can be done by considering only the corresponding signatures; we will discuss efficient ways for computing multiplications in Section 3. Moreover, it is easy to see that the inverse of a dyadic matrix is also a dyadic matrix; this can be easily computed using Sylvester-Hadamard matrices (see Section 3.2).We will expand on this in Section 4.

Finally, we introduce a relaxed notion of dyadicity, which will be useful throughout the paper.

Definition 2.3 *A* quasi-dyadic *matrix is a (possibly non-dyadic) block matrix whose elements are dyadic submatrices, i.e. an element of* $\mathbf{\Delta}\left(\mathcal{R}^n\right)^{d_1 \times d_2}$.

3 Multiplication of Dyadic Matrices

In this section we consider different methods for computing the multiplication between two dyadic matrices. In fact, we have just mentioned how some matrix operations, like the sum or the inversion, can be efficiently performed in the dyadic case just by considering the signatures. Multiplication can be strongly improved with similar methods, which exploit the particular structure of such matrices. In particular, we analyze three different algorithms and provide estimations for their complexities; we then compare the performance of the various algorithms.
For ease of notation, we will refer to the two $n \times n$ matrices that we want to multiply simply as $\mathbf{A}$ and $\mathbf{B}$, with $\mathbf{a} = [a_0, a_1, \cdots, a_{n-1}]$ and $\mathbf{b} = [b_0, b_1, \cdots, b_{n-1}]$ being the respective signatures. Maintaining the same notation, the product matrix $\mathbf{C} = \mathbf{AB}$, which is also dyadic, will have signature $\mathbf{c} = [c_0, c_1, \cdots, c_{n-1}]$.
In particular, we focus on the special case of quasi-dyadic matrices with elements belonging to a field $\mathbb{F}$ of characteristic 2.

3.1 Standard Multiplication

The first algorithm we analyze is described in Algorithm 1; we refer to it as the *standard multiplication*. The element of $\mathbf{C}$ in position (i, j) is obtained as the multiplication between the i-th row of $\mathbf{A}$ and the j-th column of $\mathbf{B}$. Since dyadic matrices are symmetric, this is equivalent to the inner product between the i-th row of $\mathbf{A}$ and the j-th one of $\mathbf{B}$. The signature $\mathbf{c}$ (i.e., the first row of $\mathbf{C}$) is obtained by inner products involving only $\mathbf{a}$ (i.e., the first row of $\mathbf{A}$). Thus, we can just construct the rows of $\mathbf{B}$, by permutations of the elements in $\mathbf{b}$, and then compute the inner products.

Algorithm 1 Standard multiplication of dyadic matrices

INPUT: $r \in \mathbb{N}$, $n = 2^r$ and $\mathbf{a}, \mathbf{b} \in \mathbb{F}^n$.
OUTPUT: $\mathbf{c} \in \mathbb{F}^n$ such that $\mathbf{\Delta}(\mathbf{c}) = \mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b})$.

$\mathbf{c} \leftarrow$ vector of length n, initialized with null elements.
$c_0 \leftarrow a_0 \cdot b_0$
for $i \leftarrow 1$ **to** $n-1$ **do**
    $c_0 \leftarrow c_0 + a_i b_i$
    $i^{(b)} \leftarrow$ binary representation of i, using n bits.
    for $\{j = 0, 1, \cdots, n-1\}$ **do**
        $j^{(b)} \leftarrow$ binary representation of j, using n bits.
        $\pi^{(b)} \leftarrow i^{(b)} \oplus j^{(b)}$
        $\pi \leftarrow$ conversion of $\pi^{(b)}$ into an integer.
        $c_i \leftarrow c_i + a_i b_\pi$
    end for
end for
return c

The complexity of the algorithm is due to two different types of operations:

1. In order to construct the rows of $\mathbf{B}$, we need the indexes of the corresponding permutations. Each index is computed as the modulo 2 sum of two binary vectors of length r, so can be obtained with a complexity of r binary operations. Thus, considering that we need to repeat this operation for $2^r - 1$ rows (for the first one, no permutation is needed), the complexity of this procedure can be estimated as $r \cdot 2^r \cdot (2^r - 1)$.
2. Each element of $\mathbf{c}$ is obtained as the inner product between two vectors of 2^r elements, assuming values in $\mathbb{F}$. This operation requires 2^r multiplication and $2^r - 1$ sums in $\mathbb{F}$. If we denote as $C_{\texttt{mult}}$ and $C_{\texttt{sum}}$ the costs of, respectively, a multiplication and a sum in $\mathbb{F}$, the total number of binary operations needed to compute 2^r inner products can be estimated as $2^{2r} \cdot C_{\texttt{mult}} + (2^{2r} - 2^r) \cdot C_{\texttt{sum}}$.

The complexity of a standard multiplication between two dyadic signatures can be estimated as:

$$C_{\texttt{std}} = r \cdot \left(2^{2r} - 2^r\right) + 2^{2r} \cdot C_{\texttt{mult}} + (2^{2r} - 2^r) \cdot C_{\texttt{sum}} \tag{3}$$

3.2 Dyadic Convolution

Definition 3.1 *The* dyadic convolution *of two vectors* $\mathbf{a}, \mathbf{b} \in \mathcal{R}$*, denoted by* $\mathbf{a} \triangle \mathbf{b}$*, is the unique vector of* $\mathcal{R}$ *such that* $\mathbf{\Delta}(\mathbf{a} \triangle \mathbf{b}) = \mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b})$.

Of particular interest to us is the case where $\mathcal{R}^n$ is actually a field $\mathbb{F}$. Dyadic matrices over $\mathbb{F}$ form a commutative subring $\mathbf{\Delta}(\mathbb{F}^n) \subset \mathbb{F}^{n \times n}$, and this property gives rise to efficient arithmetic algorithms to compute the dyadic convolution. In particular, we here consider the fast Walsh-Hadamard transform (FWHT), which is well known [7] but seldom found in a cryptographic context. We describe it here for ease of reference. We firstly recall the FWHT for the case of a field $\mathbb{F}$ such that $\text{char}(\mathbb{F}) \neq 2$, and then describe how this technique can be generalized to consider also the case of $\text{char}(\mathbb{F}) = 2$ (which, again, is the one we are interested in).

Definition 3.2 *Let* $\mathbb{F}$ *be a field with* $\mathrm{char}(\mathbb{F}) \neq 2$. *The* Sylvester-Hadamard matrix $\mathbf{H}_r \in \mathbb{F}^n$ *is recursively defined as*

$$\begin{aligned} \mathbf{H}_0 &= \left[\ 1\ \right], \\ \mathbf{H}_r &= \begin{bmatrix} \mathbf{H}_{r-1} & \mathbf{H}_{r-1} \\ \mathbf{H}_{r-1} & -\mathbf{H}_{r-1} \end{bmatrix}, r > 0. \end{aligned}$$

One can show by straightforward induction that $\mathbf{H}_r^2 = 2^r\mathbf{I}_r$ and hence $\mathbf{H}_r^{-1} = 2^{-r}\mathbf{H}_r$, which can also be expressed recursively as

$$\begin{aligned} \mathbf{H}_0^{-1} &= \left[\ 1\ \right], \\ \mathbf{H}_r^{-1} &= \frac{1}{2}\begin{bmatrix} \mathbf{H}_{r-1}^{-1} & \mathbf{H}_{r-1}^{-1} \\ \mathbf{H}_{r-1}^{-1} & -\mathbf{H}_{r-1}^{-1} \end{bmatrix}, r > 0. \end{aligned}$$

Lemma 3.1 *Let* $\mathbb{F}$ *be a field with* $\mathrm{char}(\mathbb{F}) \neq 2$. *If* $\mathbf{M} \in \mathbb{F}^{n\times n}$ *is dyadic, then* $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$ *is diagonal.*

Proof The lemma clearly holds for $r = 0$. Now let $r > 0$, and write

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix}$$

where $\mathbf{A}$ and $\mathbf{B}$ are dyadic. It follows that

$$\begin{aligned} \mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r &= \frac{1}{2}\begin{bmatrix} \mathbf{H}_{r-1}^{-1} & \mathbf{H}_{r-1}^{-1} \\ \mathbf{H}_{r-1}^{-1} & -\mathbf{H}_{r-1}^{-1} \end{bmatrix}\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix}\begin{bmatrix} \mathbf{H}_{r-1} & \mathbf{H}_{r-1} \\ \mathbf{H}_{r-1} & -\mathbf{H}_{r-1} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{H}_{r-1}^{-1}\mathbf{M}_+\mathbf{H}_{r-1} & \mathbf{O} \\ \mathbf{O} & \mathbf{H}_{r-1}^{-1}\mathbf{M}_-\mathbf{H}_{r-1} \end{bmatrix}, \end{aligned}$$

and since both $\mathbf{M}_+ = \mathbf{A} + \mathbf{B}$ and $\mathbf{M}_- = \mathbf{A} - \mathbf{B}$ are dyadic, $\mathbf{H}_{r-1}^{-1}\mathbf{M}_+\mathbf{H}_{r-1}$ and $\mathbf{H}_{r-1}^{-1}\mathbf{M}_-\mathbf{H}_{r-1}$ are diagonal by induction, as is thus also $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$. □

Lemma 3.1 establishes that Sylvester-Hadamard matrices diagonalize all dyadic matrices. In particular, the factors in a product of dyadic matrices are thus simultaneously diagonalized, suggesting an efficient way to carry out the matrix multiplication, namely, computing $\mathbf{H}_r^{-1}(\mathbf{M}\mathbf{N})\mathbf{H}_r = (\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r)(\mathbf{H}_r^{-1}\mathbf{r}\mathbf{H}_r)$ given the diagonal forms $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$ and $\mathbf{H}_r^{-1}\mathbf{r}\mathbf{H}_r$ of two dyadic matrices $\mathbf{M}$ and $\mathbf{N}$ requires only n multiplications of the diagonal elements.

In fact, it is not necessary to compute $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$ in full to obtain the diagonal form of $\mathbf{M}$, as indicated by the following result:

Lemma 3.2 *Let* $\mathbb{F}$ *be a field with* $\mathrm{char}(\mathbb{F}) \neq 2$. *The diagonal form of a dyadic matrix* $\mathbf{M} \in \mathbb{F}^{n\times n}$ *is the first line of* $\mathbf{M}\mathbf{H}_r$. *In other words,* $\mathbf{H}_r^{-1}\boldsymbol{\Delta}(\mathbf{h})\mathbf{H}_r = \mathrm{diag}(\mathbf{h}\mathbf{H}_r)$.

Proof The lemma clearly holds for $r = 0$. Now let $r > 0$, and with the notation of Lemma 3.1, the diagonal of $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$ is the concatenation of the diagonals of $\mathbf{H}_{r-1}^{-1}\mathbf{M}_+\mathbf{H}_{r-1}$ and $\mathbf{H}_{r-1}^{-1}\mathbf{M}_-\mathbf{H}_{r-1}$. Similarly, since

$$\mathbf{M}\mathbf{H}_r = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix}\begin{bmatrix} \mathbf{H}_{r-1} & \mathbf{H}_{r-1} \\ \mathbf{H}_{r-1} & -\mathbf{H}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{M}_+\mathbf{H}_{r-1} & \mathbf{M}_-\mathbf{H}_{r-1} \\ \mathbf{M}_+\mathbf{H}_{r-1} & -\mathbf{M}_-\mathbf{H}_{r-1} \end{bmatrix},$$

the first line of $\mathbf{M}\mathbf{H}_r$ is the concatenation of the first lines of $\mathbf{M}_+\mathbf{H}_{r-1}$ and $\mathbf{M}_-\mathbf{H}_{r-1}$, which by induction are the diagonals of $\mathbf{H}_{r-1}^{-1}\mathbf{M}_+\mathbf{H}_{r-1}$ and $\mathbf{H}_{r-1}^{-1}\mathbf{M}_-\mathbf{H}_{r-1}$ respectively, yielding the claimed property. □

Corollary 3.2.1 *Computing* $\mathbf{c}$ *such that* $\mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b}) = \mathbf{\Delta}(\mathbf{c})$ *involves only three multiplications of vectors by Sylvester-Hadamard matrices.*

Proof By Lemma 3.2, $\operatorname{diag}(\mathbf{a}\mathbf{H}_r)\operatorname{diag}(\mathbf{b}\mathbf{H}_r) = (\mathbf{H}_r^{-1}\mathbf{\Delta}(\mathbf{a})\mathbf{H}_r)(\mathbf{H}_r^{-1}\mathbf{\Delta}(\mathbf{b})\mathbf{H}_r) = \mathbf{H}_r^{-1}\mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b})\mathbf{H}_r = \mathbf{H}_r^{-1}\mathbf{\Delta}(\mathbf{c})\mathbf{H}_r = \operatorname{diag}(\mathbf{c}\mathbf{H}_r)$. Now simply retrieve $\mathbf{c}$ from $\mathbf{z} = \mathbf{c}\mathbf{H}_r$ as $\mathbf{c} = \mathbf{z}\mathbf{H}_r^{-1} = 2^{-r}\mathbf{z}\mathbf{H}_r$. □

The structure of Sylvester-Hadamard matrices leads to an efficient algorithm to compute $\mathbf{a}\mathbf{H}_r$ for $\mathbf{a} \in \mathbb{F}^n$, which is known as the fast Walsh-Hadamard transform. Let $[\mathbf{a}_0, \mathbf{a}_1]$ be the two halves of $\mathbf{a}$. Thus

$$\mathbf{a}\mathbf{H}_r = [\mathbf{a}_0, \mathbf{a}_1]\begin{bmatrix} \mathbf{H}_{r-1} & \mathbf{H}_{r-1} \\ \mathbf{H}_{r-1} & -\mathbf{H}_{r-1} \end{bmatrix} = [(\mathbf{a}_0 + \mathbf{a}_1)\mathbf{H}_{r-1}, (\mathbf{a}_0 - \mathbf{a}_1)\mathbf{H}_{r-1}].$$

This recursive algorithm, which can be easily written in purely sequential fashion (Algorithm 2), has complexity $\Theta(n \log n)$, specifically, rn additions or subtractions in $\mathbb{F}$. It is therefore somewhat more efficient than the fast Fourier transform, which involves multiplications by n-th roots of unity, when they are available at all (otherwise working in extension fields is unavoidable, and more expensive).

Algorithm 2 The fast Walsh-Hadamard transform (FWHT)

INPUT: $r \in \mathbb{N}$, $n = 2^r$ and $\mathbf{a} \in \mathbb{F}^n$ with $\operatorname{char}(\mathbb{F}) \neq 2$.
OUTPUT: $\mathbf{a}\mathbf{H}_r$.

```
1: v ← 1
2: for j ← 1 to n do
3:     w ← v
4:     v ← 2v
5:     for i ← 0 to n − 1 by v do
6:         for l ← 0 to w − 1 do
7:             s ← a_{i+l}
8:             q ← a_{i+l+w}
9:             a_{i+l} ← s + q
10:            a_{i+l+w} ← s − q
11:        end for
12:    end for
13: end for
14: return a
```

The product of two dyadic matrices $\mathbf{\Delta}(\mathbf{a})$ and $\mathbf{\Delta}(\mathbf{b})$, or equivalently the dyadic convolution $\mathbf{a} \vartriangle \mathbf{b}$, can thus be efficiently computed as described in Algorithm 3. The total cost is $3rn$ additions or subtractions and $2n$ multiplications (half of these by the constant $2^{-r} = 1/n$) in $\mathbb{F}$, with an overall complexity $\Theta(n \log n)$. Notice that this is also the complexity of computing $\det \mathbf{\Delta}(\mathbf{a})$.

The fast Walsh-Hadamard transform itself is not immediately possible on fields of characteristic 2, since it depends on Sylvester-Hadamard matrices which must contain a primitive square root of

Algorithm 3 Dyadic convolution via the FWHT

INPUT: $r \in \mathbb{N}$, $n = 2^r$ and $\mathbf{a}, \mathbf{b} \in \mathbb{F}^n$ with $\mathrm{char}(\mathbb{F}) \neq 2$.
OUTPUT: $\mathbf{a} \triangle \mathbf{b} \in \mathbb{F}^n$ such that $\mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b}) = \mathbf{\Delta}(\mathbf{a} \triangle \mathbf{b})$.

1: $\mathbf{c} \leftarrow$ vector of length n, initialized with null elements.
2: $\tilde{\mathbf{c}} \leftarrow$ vector of length n, initialized with null elements.
3: Compute $\tilde{\mathbf{a}} \leftarrow \mathbf{a}\mathbf{H}_r$ via Algorithm 2. ▷ expansion $1 \rightarrow r+1$
4: Compute $\tilde{\mathbf{b}} \leftarrow \mathbf{b}\mathbf{H}_r$ via Algorithm 2. ▷ expansion $1 \rightarrow r+1$
5: **for** $j \leftarrow 0$ **to** $n-1$ **do**
6: $\quad \tilde{c}_j \leftarrow \tilde{a}_j \tilde{b}_j$ ▷ expansion $r+1 \rightarrow 2r+1$
7: **end for**
8: Compute $\mathbf{c} \leftarrow \tilde{\mathbf{c}}\mathbf{H}_r$ via Algorithm 2. ▷ expansion $2r+1 \rightarrow 3r+1$
9: $\mathbf{c} \leftarrow 2^{-r}\mathbf{c}$
10: **return** $\mathbf{c}$

unity. Yet the FWHT algorithm can be lifted to characteristic 0, namely, from $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ to $\mathbb{Z}$, or more generally from $\mathbb{F}_{2^N} = (\mathbb{Z}/2\mathbb{Z})[x]/P(x)$ (for some irreducible $P(x)$ of degree N) to $\mathbb{Z}[x]$. Algorithm 3 can then be applied, and its output mapped back to the relevant binary field by modular reduction. This incurs a space expansion by a logarithmic factor, though. Each bit from $\mathbb{F}_2$ is mapped to intermediate values that can occupy as much as $3r+1$ bits; correspondingly, each element from $\mathbb{F}_{2^N}$ is mapped to intermediate values that can occupy as much as $(3r+1)N$ bits. Thus the component-wise multiplication in Algorithm 3 becomes more complicated to implement for large N. However, the method remains very efficient for the binary case as long as each expanded integer component fits a computer word. For a typical word size of 32 bits and each binary component being expanded by a factor of $3r+1$, this means that blocks as large as 1024×1024 can be tackled efficiently. On more restricted platforms where the maximum available word size is 16 bits, dyadic blocks of size 32×32 can still be handled with relative ease.

3.3 Karatsuba Multiplication

In this section we propose a method which is inspired by Karatsuba's algorithm for the multiplication of two integers [9]. Let us denote by $\mathbf{a}_0$ and $\mathbf{a}_1$, respectively, the first and second halves of $\mathbf{a}$, i.e.:

$$\begin{aligned} \mathbf{a}_0 &= \left[a_0, a_1, \cdots, a_{\frac{n}{2}-1}\right] \\ \mathbf{a}_1 &= \left[a_{\frac{n}{2}}, a_{\frac{n}{2}+1}, \cdots, a_{n-1}\right]. \end{aligned} \tag{4}$$

The same notation is used for $\mathbf{b}_0$ and $\mathbf{b}_1$ and $\mathbf{c}_0$ and $\mathbf{c}_1$, corresponding to the halves of $\mathbf{B}$ and $\mathbf{C}$. Some straightforward computations show that the following relations hold:

$$\begin{aligned} \mathbf{c}_0 &= \mathbf{a}_0\mathbf{b}_0 + \mathbf{a}_1\mathbf{b}_1 \\ \mathbf{c}_1 &= (\mathbf{a}_0 + \mathbf{a}_1)(\mathbf{b}_0 + \mathbf{b}_1) + \mathbf{c}_0 \end{aligned} \tag{5}$$

The iterative application of equation (5) allows to compute multiplications between dyadic matrices of any size. Let us denote as $C_{\mathtt{mul}}^{(2^z)}$ and $C_{\mathtt{sum}}^{(2^z)}$ the complexities of a multiplication and a sum between

two signatures of length 2^z. For the sum of two dyadic signatures of size 2^z we have:

$$C_{sum}^{(2^z)} = 2^z \cdot C_{\mathtt{sum}}, \tag{6}$$

where $C_{\mathtt{sum}}$ again denotes the complexity of a sum in the finite field.
The complexity of this algorithm can thus be estimated as:

$$\begin{aligned}
C_{\mathtt{Kar}} &= 3 \cdot C_{\mathtt{mul}}^{(2^{r-1})} + 4 \cdot C_{\mathtt{sum}}^{(2^{r-1})} = \\
&= 3 \cdot C_{\mathtt{mul}}^{(2^{r-1})} + 4 \cdot 2^{r-1} \cdot C_{\mathtt{sum}} = \\
&= 3 \cdot \left[3 \cdot C_{\mathtt{mul}}^{(2^{r-2})} + 4 \cdot C_{\mathtt{sum}}^{(2^{r-2})}\right] + 4 \cdot 2^{r-1} \cdot C_{\mathtt{sum}} = \\
&= 3 \left[3 \cdot C_{\mathtt{mul}}^{(2^{r-2})} + 4 \cdot 2^{r-2} \cdot C_{\mathtt{sum}}\right] + 4 \cdot 2^{r-1} \cdot C_{\mathtt{sum}} = \\
&= 3^2 \cdot C_{\mathtt{mul}}^{(2^{r-2})} + 4 \cdot \left[3 \cdot 2^{r-2} + 2^{r-1}\right] C_{\mathtt{sum}} = \\
&= 3^3 \cdot C_{\mathtt{mul}}^{(2^{r-3})} + 4 \cdot \left[3^2 \cdot 2^{r-3} + 3 \cdot 2^{r-2} + 2^{r-1}\right] C_{\mathtt{sum}} = \\
&= \cdots = \\
&= 3^r \cdot C_{\mathtt{mul}} + 4 \cdot \left[\sum_{j=1}^{r} 3^{j-1} 2^{r-j}\right] \cdot C_{\mathtt{sum}} = \\
&= 3^r \cdot C_{\mathtt{mul}} + \frac{4}{3} \cdot 2^r \cdot \left[\sum_{j=1}^{r} \left(\frac{3}{2}\right)^j\right] \cdot C_{\mathtt{sum}}
\end{aligned} \tag{7}$$

Taking into account the well known sum of a geometric series, we have:

$$\begin{aligned}
\sum_{j=1}^{r} \left(\frac{3}{2}\right)^j &= -1 + \sum_{j=0}^{r} \left(\frac{3}{2}\right)^j = \\
&= -1 + \frac{1 - (\frac{3}{2})^{r+1}}{1 - \frac{3}{2}} = \frac{3^{r+1}}{2^r} - 3.
\end{aligned} \tag{8}$$

Considering this result, equation (7) leads to:

$$C_{\mathtt{Kar}} = 3^r \cdot C_{\mathtt{mul}} + 4 \cdot [3^r - 2^r] \cdot C_{\mathtt{sum}} \tag{9}$$

4 Efficient Inversion of Dyadic and Quasi-Dyadic Matrices

In this section we propose an efficient algorithm for computing the inverse of quasi-dyadic matrices. The algorithm in principle is targeted to matrices that are not fully dyadic (even though, obviously, they have to be square). This is because, while it is of course possible to apply our procedure to fully dyadic matrices, these can in general be inverted much more easily, as we will see next.

To begin, remember that by definition of a quasi-dyadic matrix (Definition 2.3) we mean an element of $\mathbf{\Delta}\left(\mathcal{R}^n\right)^{d_1 \times d_2}$.

4.1 Dyadic Matrices

The inverse of a dyadic matrix (i.e. $d_1 = d_2 = 1$) can be efficiently computed, using only the signature, as described by the following Lemma.

Lemma 4.1 *Let $n = 2^r$ for $r \in \mathbb{N}$ and let $\mathbf{\Delta}(\mathbf{a}) \in \mathcal{R}^{n\times n}$ be a dyadic matrix with signature $\mathbf{a}$. Then the inverse $\mathbf{\Delta}(\mathbf{a})^{-1}$ is the dyadic matrix $\mathbf{\Delta}(\mathbf{b})$, for $\mathbf{b} = \frac{1}{2^r}\tilde{\mathbf{b}}\mathbf{H}_r$, where $\tilde{\mathbf{b}}$ is the vector such that* $\mathrm{diag}(\tilde{\mathbf{b}}) = [\mathrm{diag}\,(\mathbf{a}\mathbf{H}_r)]^{-1}$.

Proof We have $\mathbf{\Delta(b)\Delta(a)} = \mathbf{I}_n = \mathbf{\Delta}([1, 0, \cdots, 0])$. The diagonal form of $\mathbf{I}_n$ corresponds to the first row of the product $\mathbf{I}_n\mathbf{H}_r$, and so it is equal to the first row of $\mathbf{H}_r$, that is the length-n vector made of all ones. According to Corollary 3.2.1, we can write:

$$\mathrm{diag}(\mathbf{b})\,\mathrm{diag}(\mathbf{a}) = \mathrm{diag}([1, 1, \cdots, 1]).$$

We then define $\mathbf{a}\mathbf{H}_r = [\lambda_0, \lambda_1, \cdots, \lambda_{n-1}]$, and obtain:

$$\mathrm{diag}(\mathbf{b}) = \mathrm{diag}([1, 1, \cdots, 1])\,\mathrm{diag}^{-1}(\mathbf{a}) = \mathrm{diag}([\lambda_0^{-1}, \lambda_1^{-1}, \cdots, \lambda_{n-1}^{-1}]).$$

Because of Lemma 3.2, we finally have:

$$\mathbf{b} = \mathrm{diag}^{-1}(\mathbf{a})\mathbf{H}_r^{-1} = \tfrac{1}{2^r}\,\mathrm{diag}^{-1}(\mathbf{a})\mathbf{H}_r. \qquad \square$$

As we mentioned before, the above Lemma yields a very simple way for computing the inverse of a dyadic matrix: given a signature $\mathbf{a}$, we just need to compute its diagonalized form as $\mathbf{a}\mathbf{H}_r$, compute the reciprocals of its elements and put it in a vector $\tilde{\mathbf{b}}$. Finally, the inverse of $\mathbf{\Delta(a)}$ can be obtained as $\frac{1}{2^r}\tilde{\mathbf{b}}\mathbf{H}_r$. This property also leads to a very simple way to check the singularity of $\mathbf{\Delta}(\mathbf{a})$: if its diagonalized form contains some null elements, then it is singular.

We now focus on the case of dyadic matrices over a field $\mathbb{F}$ with characteristic 2. One can show by induction that in such a case a dyadic matrix $\mathbf{\Delta}(\mathbf{a})$ of dimension n satisfies $\mathbf{\Delta}(\mathbf{a})^2 = (\sum_i a_i)^2\mathbf{I}$, and hence its inverse, when it exists, is $\mathbf{\Delta}(\mathbf{a})^{-1} = (\sum_i a_i)^{-2}\mathbf{\Delta}(\mathbf{a})$, which can be computed in $O(n)$ steps since it is entirely determined by its first row. It is equally clear that $\det \mathbf{\Delta}(\mathbf{a}) = (\sum_i a_i)^n$, which can be computed with the same complexity (notice that raising to the power of $n = 2^r$ only involves r squarings). Basically, verifying whether a dyadic matrix has full rank or not can be easily done by checking whether the sum of the elements of the signature equals 0.

4.2 Quasi-Dyadic Matrices

Consider a quasi-dyadic matrix $\mathbf{M}$. Since the matrix has to be square, we have $d_1 = d_2 = d$, and the matrix has dimension $dn \times dn$. Such a matrix can be compactly represented just by the signatures of the dyadic blocks. To simplify notation, we can denote the signature of the dyadic-block in position (i, j) as $\hat{\mathbf{m}}_{i,j}$, and store all such vectors in a matrix $\hat{\mathbf{M}} \in \mathcal{R}^{d\times dn}$:

$$\hat{\mathbf{M}} = \begin{pmatrix} \hat{\mathbf{m}}_{0,0} & \hat{\mathbf{m}}_{0,1} & \cdots & \hat{\mathbf{m}}_{0,d-1} \\ \hat{\mathbf{m}}_{1,0} & \hat{\mathbf{m}}_{1,1} & \cdots & \hat{\mathbf{m}}_{1,d-1} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{m}}_{d-1,0} & \hat{\mathbf{m}}_{d-1,1} & \cdots & \hat{\mathbf{m}}_{d-1,d-1} \end{pmatrix}. \tag{10}$$

We focus again on the special case of quasi-dyadic matrices over a field $\mathbb{F}$ with characteristic 2.

The LUP decomposition is a method which factorizes a matrix $\mathbf{M}$ as $\mathbf{LUP}$, where $\mathbf{L}$ and $\mathbf{U}$ are lower triangular and upper triangular matrices, respectively, and $\mathbf{P}$ is a permutation.
Exploiting this factorization, the inverse of $\mathbf{M}$ can thus be expressed as:

$$\mathbf{M}^{-1} = \mathbf{P}^{-1}\mathbf{U}^{-1}\mathbf{L}^{-1}. \tag{11}$$

The advantage of this method is that the inverses appearing in (11) can be easily computed, because of their particular structures. In fact, the inverse of an upper (lower) triangular matrix is obtained via a simple backward (forward) substitution procedure, while the inverse of $\mathbf{P}$ is its transpose.

In in Algorithm 4 we describe how the LUP decomposition can be efficiently applied to any square quasi-dyadic matrix.

Algorithm 4 LUP Decomposition of a Quasi-Dyadic Matrix

INPUT: $d, r \in \mathbb{N}$, $n = 2^r$ and $\hat{\mathbf{M}} \in \mathbb{F}^{d \times dn}$ with $\mathrm{char}(\mathbb{F}) = 2$.
OUTPUT: $\hat{\mathbf{M}} \in \mathbb{F}^{d \times dn}$, $\hat{\mathbf{P}} \in \mathbb{N}^d$.

$\hat{\mathbf{P}} \leftarrow [0, 1, \cdots, d-1]$
$u \leftarrow 0$
for $j \leftarrow 0$ **to** $d-1$ **do**
    Update u, $\hat{\mathbf{M}}$ and $\hat{\mathbf{P}}$ via Algorithm 5. ▷ Pivoting of the signatures in the j-th column
    if $u = 0$ **then**
        return u ▷ $\hat{\mathbf{M}}$ is singular
    end if
    for $i \leftarrow j+1$ **to** d **do**
        $\hat{\mathbf{m}}_{i,j} \leftarrow \hat{\mathbf{m}}_{i,j}\hat{\mathbf{m}}_{j,j}^{-1}$
    end for
    for $i \leftarrow j+1$ **to** $d-1$ **do**
        for $l \leftarrow j+1$ **to** $d-1$ **do**
            $\hat{\mathbf{m}}_{i,l} \leftarrow \hat{\mathbf{m}}_{i,l} + \hat{\mathbf{m}}_{i,j}\hat{\mathbf{m}}_{j,l}$
        end for
    end for
end for
return $\hat{\mathbf{M}}$, $\hat{\mathbf{P}}$

Our proposed procedure consists in using a block decomposition, which works directly on the signatures, in order to exploit the simple and efficient algebra of dyadic matrices. It can be easily shown that, for a quasi-dyadic matrix, its factors $\mathbf{L}$, $\mathbf{U}$ and $\mathbf{P}$ are in quasi-dyadic form as well: as we have done for the matrix $\mathbf{M}$, we refer to their compact representations as $\hat{\mathbf{L}}$, $\hat{\mathbf{U}}$ and $\hat{\mathbf{P}}$, respectively.

The algorithm takes as input a matrix $\hat{\mathbf{M}}$, as in (10), and computes its LUP factorization; outputs of the algorithm are the modified matrix $\hat{\mathbf{M}}$, having as elements the ones of its factors $\hat{\mathbf{L}}$ and $\hat{\mathbf{U}}$, and the permutation $\hat{\mathbf{P}}$. As in (10), we denote as $\hat{\mathbf{m}}_{i,j}$ the signature in position (i, j) in the output matrix $\hat{\mathbf{M}}$. The matrices $\hat{\mathbf{L}}$ and $\hat{\mathbf{U}}$ can then be expressed as:

$$\hat{\mathbf{L}} = \begin{pmatrix} \hat{\mathbf{1}} & \hat{\mathbf{0}} & \hat{\mathbf{0}} & \cdots & \hat{\mathbf{0}} \\ \hat{\mathbf{m}}_{1,0} & \hat{\mathbf{1}} & \hat{\mathbf{0}} & \cdots & \hat{\mathbf{0}} \\ \hat{\mathbf{m}}_{2,0} & \hat{\mathbf{m}}_{2,1} & \hat{\mathbf{1}} & \cdots & \hat{\mathbf{0}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{m}}_{d-1,0} & \hat{\mathbf{m}}_{d-1,1} & \hat{\mathbf{m}}_{d-1,2} & \cdots & \hat{\mathbf{1}} \end{pmatrix}, \quad \hat{\mathbf{U}} = \begin{pmatrix} \hat{\mathbf{m}}_{0,0} & \hat{\mathbf{m}}_{0,1} & \hat{\mathbf{m}}_{0,2} & \cdots & \hat{\mathbf{m}}_{0,d-1} \\ \hat{\mathbf{0}} & \hat{\mathbf{m}}_{1,1} & \hat{\mathbf{m}}_{1,2} & \cdots & \hat{\mathbf{m}}_{1,d-1} \\ \hat{\mathbf{0}} & \hat{\mathbf{0}} & \hat{\mathbf{m}}_{2,2} & \cdots & \hat{\mathbf{m}}_{2,d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{0}} & \hat{\mathbf{0}} & \hat{\mathbf{0}} & \cdots & \hat{\mathbf{m}}_{d-1,d-1} \end{pmatrix} \tag{12}$$

where $\hat{\mathbf{1}}$ and $\hat{\mathbf{0}}$ denote, respectively, the signature of the identity matrix and the one of the null matrix (i.e. the length-k vectors $[1, 0, \cdots, 0]$ and $[0, 0, \cdots, 0]$).

The matrix $\hat{\mathbf{P}}$ is represented through a length-d vector $[p_0, p_1, \cdots, p_{d-1}]$, containing a permutation of the integers $[0, 1, \cdots, d-1]$; the rows of $\hat{\mathbf{M}}$ get permuted according to the elements of $\hat{\mathbf{P}}$. In particular, the elements of $\hat{\mathbf{P}}$ are obtained through a block pivoting procedure, which is described in Algorithm 5.

Algorithm 5 Block pivoting

INPUT: $d, j, r \in \mathbb{N}$, $n = 2^r$, $\hat{\mathbf{P}} \in \mathbb{N}^n$ and $\hat{\mathbf{M}} \in \mathbb{F}^{d \times dn}$ with $\text{char}(\mathbb{F}) = 2$, .
OUTPUT: $u \in \mathbb{N}$.

$u \leftarrow 0$
$i \leftarrow j$
while $i \leq d-1$ **do**
  $w \leftarrow \texttt{sum}(\hat{\mathbf{m}}_{i,j})$ ▷ Sum of the elements in $\hat{\mathbf{m}}_{i,j}$
  if $w = 0$ **then**
    $z \leftarrow p_j$
    $p_j \leftarrow p_i$
    $p_i \leftarrow z$
    for $l \leftarrow 0$ **to** $d-1$ **do**
      $z \leftarrow \hat{\mathbf{m}}_{j,l}$
      $\hat{\mathbf{m}}_{j,l} \leftarrow \hat{\mathbf{m}}_{i,l}$
      $\hat{\mathbf{m}}_{i,l} \leftarrow z$
      $i \leftarrow i+1$
    end for
  else
    $i \leftarrow d$
    $u \leftarrow 1$
  end if
end while
return u

This function takes as input $\hat{\mathbf{M}}$, $\hat{\mathbf{P}}$ and an integer j, and searches for a pivot (i.e., a non singular signature) in the j-th column of $\hat{\mathbf{M}}$, starting from $\hat{\mathbf{m}}_{j,j}$, and places it in position (j, j). As the procedure goes on, every time a singular signature is tested, the rows of $\hat{\mathbf{M}}$ get permuted; the elements of $\hat{\mathbf{P}}$ are accordingly modified. If the j-th column contains all singular blocks, this means that the matrix $\hat{\mathbf{M}}$ is singular; in such a case, this event is notified by setting $u = 0$.

We point out that, for the matrices we are considering, we expect Algorithm 4 to be particularly efficient. First of all, as we have already said, this is due to the possibility of efficiently performing

operations involving dyadic matrices; in addition, the dyadic structure should also speed-up the pivoting procedure. In fact, we can consider a signature in $\hat{\mathbf{M}}$ as a collection of k random elements picked from $GF(2^N)$: thus, their sum can be assumed to be a random variable with uniform distribution among the elements of the field $GF(2^N)$. So, the probability of it being equal to 0, which corresponds to the probability of the corresponding signature to be singular, equals 2^{-m}. This probability gets lower as m increases: this fact means that the expected number of operations performed by Algorithm 5 should be particularly low. Basically, most of the times the function will just compute the sum of the elements in $\hat{\mathbf{m}}_{j,j}$ and verify whether it is null or not.

Once the factorization of $\hat{\mathbf{M}}$ has been obtained, we just need to perform the computation of $\mathbf{M}^{-1}$ through (11). Since the inverse of a triangular matrix maintains the original triangular structure, the computation of the inverses $\hat{\mathbf{L}}^{-1}$ and $\hat{\mathbf{U}}^{-1}$ can be efficiently performed. A possible way for computing these matrices is to store the elements of both matrices in just one output matrix $\hat{\mathbf{T}}$. We do this in Algorithm 6.

Algorithm 6 Computation of $\hat{\mathbf{T}}$

INPUT: $d, r \in \mathbb{N}$, $n = 2^r$ and $\hat{\mathbf{M}} \in \mathbb{F}^{d \times dn}$ with $\mathrm{char}(\mathbb{F}) = 2$.
OUTPUT: $\hat{\mathbf{T}} \in \mathbb{F}^{d \times dn}$.

$\hat{\mathbf{T}} \leftarrow \hat{\mathbf{I}}_d$
for $j \leftarrow 0$ **to** $d-1$ **do**
  for $i \leftarrow j+1$ **to** $d-1$ **do**
    for $l \leftarrow j$ **to** $i-1$ **do**
      $\hat{\mathbf{t}}_{i,j} \leftarrow \hat{\mathbf{t}}_{i,j} + \hat{\mathbf{m}}_{i,k}\hat{\mathbf{t}}_{k,j}$
    end for
  end for
  for $i \leftarrow j$ **to** $d-1$ **do**
    for $l \leftarrow j$ **to** $i-1$ **do**
      $\hat{\mathbf{t}}_{j,i} \leftarrow \hat{\mathbf{t}}_{j,i} + \hat{\mathbf{m}}_{k,i}\hat{\mathbf{t}}_{j,k}$
    end for
    $\hat{\mathbf{t}}_{j,i} \leftarrow \hat{\mathbf{t}}_{j,i}\hat{\mathbf{m}}_{i,i}^{-1}$
  end for
end for
return $\hat{\mathbf{T}}$

The matrix $\hat{\mathbf{I}}_d$ is the compact representation of a $dn \times dn$ identity matrix, and so is composed of signatures $\delta_{i,j}\hat{\mathbf{1}}$, where $\delta_{i,j}$ denotes the Kronecker delta.

If we denote as $\hat{\mathbf{t}}_{i,j}$ the signature in position (i,j), we have:

$$\hat{\mathbf{L}}^{-1} = \begin{pmatrix} \hat{\mathbf{1}} & \hat{\mathbf{0}} & \hat{\mathbf{0}} & \cdots & \hat{\mathbf{0}} \\ \hat{\mathbf{t}}_{1,0} & \hat{\mathbf{1}} & \hat{\mathbf{0}} & \cdots & \hat{\mathbf{0}} \\ \hat{\mathbf{t}}_{2,0} & \hat{\mathbf{t}}_{2,1} & \hat{\mathbf{1}} & \cdots & \hat{\mathbf{0}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{t}}_{d-1,0} & \hat{\mathbf{t}}_{d-1,1} & \hat{\mathbf{t}}_{d-1,2} & \cdots & \hat{\mathbf{1}} \end{pmatrix}, \quad \hat{\mathbf{U}}^{-1} = \begin{pmatrix} \hat{\mathbf{t}}_{0,0} & \hat{\mathbf{t}}_{0,1} & \hat{\mathbf{t}}_{0,2} & \cdots & \hat{\mathbf{t}}_{0,d-1} \\ \hat{\mathbf{0}} & \hat{\mathbf{t}}_{1,1} & \hat{\mathbf{t}}_{1,2} & \cdots & \hat{\mathbf{t}}_{1,d-1} \\ \hat{\mathbf{0}} & \hat{\mathbf{0}} & \hat{\mathbf{t}}_{2,2} & \cdots & \hat{\mathbf{t}}_{2,d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{0}} & \hat{\mathbf{0}} & \hat{\mathbf{0}} & \cdots & \hat{\mathbf{t}}_{d-1,d-1} \end{pmatrix}. \tag{13}$$

5 Performance Analysis: Application to DAGS

In this section we provide the results of the application of our techniques to DAGS. For completeness, we have included a specification of the three DAGS algorithms in Appendix A, but for our purpose, DAGS is essentially the McEliece cryptosystem, converted to a KEM via a standard transformation [8]. In particular, the Key Generation algorithm is the same as the QD-GS McEliece version described in [12]. In this algorithm, a key role is played by the systematization (i.e. reduced row echelon form) of a quasi-dyadic rectangular matrix, the result of which will in fact be the public key for the scheme. The cost of computing said systematic matrix dwarfs everything else in key generation: according to a static analysis, this takes over 98% of the total cost of key generation. Therefore, a fast procedure to compute the systematic form will have a substantial impact on the overall performance of the algorithm.

Implementation Details. We developed a code in "C" to implement our procedures. In all cases, we use no optimizations apart from the optimization from the GCC compiler ("-O3"). The GCC version used was 7.3.1 20180406, the code was compiled for the processor Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz with 16GB of memory and operating system Arch linux version 2018.05.01 with Kernel 4.16.5. We ran 100 times each piece of code and computed the average of all measurements; to obtain the number of cycles, we used the file "cpucycles.h" from supercop[1].

Fast Multiplication. To compare our methods, we fix a dyadic order r and measure the cost of a multiplication of two matrices of size $n = 2^r$. Relevant dyadic orders for DAGS are for instance $r = 4$ and $r = 5$. We do this over different fields to highlight the difference in performance when changing fields: we tested $\mathbb{F}_{2^5}$ and $\mathbb{F}_{2^6}$ which are the fields currently used by DAGS.

Table 1: Cost of Multiplication between Dyadic Matrices

		Standard	Karatsuba	Dyadic Convolution
$\mathbb{F}_{2^5}$	$r = 4$	$4,833$	$2,194$	$3,899$
	$r = 5$	$21,285$	$5,909$	$12,045$
$\mathbb{F}_{2^6}$	$r = 4$	$5,833$	$2,194$	$4,899$
	$r = 5$	$23,231$	$6,223$	$13,568$

Efficient Inversion. We report here the results of the improved inversion procedure (Algorithms 4, 5 and 6). We compared our procedure with the equivalent portion of the DAGS implementation that we extrapolated from the publicly available source code [2]. In particular, we measured the piece of code that begins with the creation of the Cauchy matrix and ends with the generation of the systematic matrix. Table 2 shows the comparison, measured in cpu cycles.

Table 2: Comparison of Inversion Methods

	DAGS Implementation	LUP Inversion	LUP + Karatsuba
DAGS 1	$1,318,973,209$	$321,771$	$108,117$
DAGS 3	$2,211,076,311$	$557,822$	$198,199$
DAGS 5	$17,925,330,712$	$654,713$	$431,890$

[1] https://bench.cr.yp.to/supercop.html

References

[1] *https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.*

[2] *https://git.dags-project.org/dags/dags.*

[3] G. Banegas, P. S. L. M. Barreto, B. O. Boidje, P. Cayrel, G. N. Dione, K. Gaj, C. T. Gueye, R. Haeussler, J. B. Klamti, O. Ndiaye, D. T. Nguyen, E. Persichetti, and J. E. Ricardini, "DAGS: key encapsulation using dyadic GS codes," *IACR Cryptology ePrint Archive*, vol. 2017, p. 1037, 2017. [Online]. Available: http://eprint.iacr.org/2017/1037

[4] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems (corresp.)," *Information Theory, IEEE Transactions on*, vol. 24, no. 3, pp. 384 – 386, may 1978.

[5] J. R. Bunch and J. E. Hopcroft, "Triangular factorization and inversion by fast matrix multiplication," *Mathematics of Computation*, vol. 28, no. 125, pp. 231–236, 1974.

[6] P. Cayrel, G. Hoffmann, and E. Persichetti, "Efficient implementation of a cca2-secure variant of mceliece using generalized srivastava codes," in *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Fischlin, J. A. Buchmann, and M. Manulis, Eds., vol. 7293. Springer, 2012, pp. 138–155. [Online]. Available: https://doi.org/10.1007/978-3-642-30057-8_9

[7] M. N. Gulamhusein, "Simple matrix-theory proof of the discrete dyadic convolution theorem," *Electronics Letters*, vol. 9, no. 10, pp. 238–239, 1973.

[8] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the Fujisaki-Okamoto transformation," Cryptology ePrint Archive, Report 2017/604, 2017, http://eprint.iacr.org/2017/604.

[9] A. Karatsuba and Y. Ofman, *Multiplication of Multidigit Numbers by Automata*, 01 1963.

[10] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *Deep Space Network Progress Report*, vol. 44, pp. 114–116, Jan. 1978.

[11] R. Misoczki and P. S. L. M. Barreto, "Compact mceliece keys from goppa codes," in *Selected Areas in Cryptography*, 2009, pp. 376–392.

[12] E. Persichetti, "Compact mceliece keys based on quasi-dyadic srivastava codes," *Journal of Mathematical Cryptology*, vol. 6, no. 2, pp. 149–169, 2012.

[13] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

A DAGS Algorithms

We briefly describe the three algorithms that define DAGS. Generalized Srivastava codes are defined by parameters s and t, where in our case $\log s$ is the dyadic order; the codes in use have length $n = n_0 s$ and dimension $k = k_0 s$ where n_0 and k_0 are the number of dyadic blocks. Other parameters are the cardinality of the base field q and the degree of the field extension m. In addition, we have $k = k' + k''$, where k' is arbitrary and set to be "small".

The key generation process uses the following fundamental equation

$$\frac{1}{h_{i \oplus j}} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0} \tag{14}$$

which guarantees we can build a dyadic matrix, with signature $\mathbf{h} = (h_0, h_1, \ldots, h_{n-1})$, which is also a Cauchy matrix, i.e. a matrix $C(\mathbf{u}, \mathbf{v})$ with components $C_{ij} = \frac{1}{u_i - v_j}$. In [11] it is proved that we can use the fundamental equation to choose a support and polynomial for a Goppa code such that this dyadic Cauchy matrix is a parity-check matrix for the code.

Key Generation

1. Generate dyadic signature $\mathbf{h}$ according to the fundamental equation.
2. Build the vectors $(\mathbf{u}, \mathbf{v})$ that define the Cauchy matrix (again using the equation).
3. Form Cauchy matrix $\hat{H}_1 = C(\mathbf{u}, \mathbf{v})$.
4. Build $\hat{H}_i$, $i = 2, \ldots t$, by raising each element of $\hat{H}_1$ to the power of i.
5. Superimpose blocks $\hat{H}_i$ in ascending order to form matrix $\hat{H}$.
6. Generate scaling vector $\mathbf{z}$ by sampling elements z_i in $\mathbb{F}_{q^m}$ with $z_{is+j} = z_{is}$ for $i = 0, \ldots, n_0 - 1$, $j = 0, \ldots, s-1$.
7. Set $y_j = \dfrac{z_j}{\prod_{i=0}^{s-1} (u_i - v_j)^t}$ for $j = 0, \ldots, n-1$ and $\mathbf{y} = (y_0, \ldots, y_{n-1})$.
8. Form $H = \hat{H} \cdot \mathrm{Diag}(\mathbf{z})$.
9. Project H onto $\mathbb{F}_q$ using the co-trace function: call this H_{base}.
10. Write H_{base} in systematic form $(A \mid \mathbf{I}_{n-k})$.
11. The public key is the generator matrix $G = (\mathbf{I}_k \mid A^T)$.
12. The private key is the pair $(\mathbf{v}, \mathbf{y})$.

Note that all matrices involved in key generation are quasi-dyadic (with blocks of size $s \times s$), namely $\hat{H}, H, H_{base}$ and its systematic form, and the final matrix G which is the public key. Step 10 is the systematization process which is impacted by our improved inversion algorithm.

The encapsulation and decapsulation algorithms make use of three hash functions $\mathcal{G} : \mathbb{F}_q^{k'} \to \mathbb{F}_q^k$, $\mathcal{H} : \mathbb{F}_q^{k'} \to \mathbb{F}_q^{k'}$ and $\mathcal{K} : \{0,1\}^* \to \{0,1\}^\ell$, where ℓ is the desired length of the key to be shared.

Encapsulation

1. Choose $\mathbf{m} \xleftarrow{\$} \mathbb{F}_q^{k'}$.
2. Compute $\mathbf{r} = \mathcal{G}(\mathbf{m})$ and $\mathbf{d} = \mathcal{H}(\mathbf{m})$.
3. Parse $\mathbf{r}$ as $(\rho \parallel \sigma)$ then set $\mu = (\rho \parallel \mathbf{m})$.
4. Generate error vector $\mathbf{e}$ of length n and weight w from σ.
5. Compute $\mathbf{c} = \mu G + \mathbf{e}$.
6. Compute $\mathbf{k} = \mathcal{K}(\mathbf{m})$.
7. Output ciphertext $(\mathbf{c}, \mathbf{d})$; the encapsulated key is $\mathbf{k}$.

The decapsulation algorithm is essentially a run of the decoding algorithm to decode the noisy codeword received as part of the ciphertext, plus a number of integrity checks.

Decapsulation

1. Recover parity-check matrix H' in alternant form from private key.
2. Use H' to decode $\mathbf{c}$ and obtain codeword $\mu' G$ and error $\mathbf{e}'$.
3. Output $\perp$ if decoding fails or $\mathsf{wt}(\mathbf{e}') \neq w$
4. Recover μ' and parse it as $(\rho' \parallel \mathbf{m}')$.
5. Compute $\mathbf{r}' = \mathcal{G}(\mathbf{m}')$ and $\mathbf{d}' = \mathcal{H}(\mathbf{m}')$.
6. Parse $\mathbf{r}'$ as $(\rho'' \parallel \sigma')$.
7. Generate error vector $\mathbf{e}''$ of length n and weight w from σ'.
8. If $\mathbf{e}' \neq \mathbf{e}'' \vee \rho' \neq \rho'' \vee \mathbf{d} \neq \mathbf{d}'$ output $\perp$.
9. Else compute $\mathbf{k} = \mathcal{K}(\mathbf{m}')$.
10. The decapsulated key is $\mathbf{k}$.