Software Implementation of a Code-Based Key Encapsulation Mechanism from Binary QD Generalized Srivastava Codes

Boly Seck[1,3(✉)], Cheikh Thiécoumba Gueye[2], Gilbert Ndollane Dione[2], Jean Belo Klamti[4], Pierre-Louis Cayrel[3], Idy Diop[1], and Ousmane Ndiaye[2]

[1] École Eupérieure Polytechnique, Laboratoire d'imagerie médicale et de Bio-informatique, Dakar, Senegal
seck.boly@ugb.edu.sn, idy.diop@esp.sn
[2] Université Cheikh Anta Diop, Dakar, Senegal
{cheikht.gueye,gilbertndollane.dione,ousmane3.ndiaye}@ucad.edu.sn
[3] Univ Lyon, UJM-Saint-Etienne, CNRS, Laboratoire Hubert Curien UMR 5516, 42023 Saint-Etienne, France
pierre.louis.cayrel@univ-st-etienne.fr
[4] Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
jbklamti@uwaterloo.ca

Abstract. In the NIST Post-Quantum Cryptography (PQC) standardization process, among 17 candidates for code-based public-key encryption (PKE), signature or key encapsulation mechanism (KEM), only three are in the 4th evaluation round. The remaining code-based candidates are Classic McEliece [CCUGLMMNPP+20], BIKE [ABBBBDGGGM+17] and HQC [MABBBBDDGL+20]. Cryptographic primitives from coding theory are some of the most promising candidates and their security is based on the well-known problems of post-quantum cryptography. In this paper, we present an efficient implementation of a secure KEM based on binary quasi-dyadic generalized Srivastava (QD-GS) codes. With QD-GS codes defined for an extension degree $m > 2$, this key establishment scheme is protected against the attacks of Barelli-Couvreur Bardet *et al.*. We also provide parameters that are secure against folding technique and FOPT attacks. Finally, we compare the performance of our implementation in runtime with the NIST finalists based on codes for the 4th round.

Keywords: NIST PQC Standardization · QD-GS codes · Code-based KEM · Binary DAGS

1 Introduction

As a reminder, Faugère *et al.* had introduced an attack known in the literature as FOPT against scheme using quasi-cyclic or quasi-dyadic algebraic codes

J.-C. Deneuville (Ed.): CBCrypto 2022, LNCS 13839, pp. 77–89, 2023.
https://doi.org/10.1007/978-3-031-29689-5_5

[FOPT10]. Their attack exploits the algebraic structure to build a system of equations and then uses the Grobnër bases techniques to solve it efficiently. Therefore, with FOPT attack, proposals based on quasi-cyclic algebraic codes are compromised. However, techniques using a quasi-dyadic approach need to be treated with caution for a proper choice of parameters concerning the dimension of space of solutions. That means that it is possible to design secure schemes using for instance binary Goppa codes, or Generalized Srivastava (GS) codes. Separately, note that, during the first round of the NIST PQC standardization process, Banegas *et al.* [BBBCDGGHKN+18] introduced a KEM scheme based on nonbinary QD-GS codes. This scheme was broken by Barelli and Couvreur [BC18]. They used in their attack the norm and trace codes technique which works only for code designed on an extension field with extension degree m equal to 2. It was shown that a simple parameters variation of the base field could avoid this attack [BC18, Section 5.3]. Lately, Banegas *et al.* introduced a new version of their scheme called DAGS reloaded [BBBCDGGHKN+19]. At the same time, Bardet *et al.* [BBCO19] introduced a hybrid version of the Barelli-Couvreur attack against the updated parameters. However, due to a proper choice of parameters, their attack worked only for the parameters of NIST security level 1 and not for the two others.

As part of this work, we show that the Barelli-Couvreur and Bardet *et al.* attacks have no effect against the code-based KEM using binary QD-GS codes that we call binary DAGS. We provide parameters that are secure against folding technique [FOPDPT15] and FOPT attacks. The main difference between the binary DAGS and the version submitted to the first round of NIST PQC standardization process [BBBCDGGHKN+18] is that the base fields are nonbinary. In addition, the underlying cryptosystem in the binary DAGS is that of Niederreiter instead of McEliece.

Contribution: In this work, we focus on the fast software implementation of the secure binary DAGS.

First, we show that the Barelli-Couvreur and Bardet *et al.* attacks have no effect against the code-based KEM using binary QD-GS codes that we called binary DAGS. We provide parameters that are secure against folding technique [FOPDPT15] and FOPT attacks.

Second, we perform an efficient software implementation of the binary DAGS. For that, we use techniques from [BBPS20] that specifically aim to improve the multiplication of QD matrices. These involve a version of the Karatsuba multiplication algorithm and an application of the LUP version in order to compute the product and inverse of matrices more efficiently. These improvements allow us to achieve better runtimes than previous DAGS implementations.

Finally, we show that our implementation is competitive in terms of execution time with the NIST candidates for advanced evaluation.

Organisation: The paper is organized as follows. In Sect. 2, we focus on the required prerequisites for this paper. In Sect. 3, we give the description of the code-based KEM from binary QD-GS codes. We also propose a set of parameters. In Sect. 4, we present the technical details about the software implementation and results. Finally, we conclude this paper in Sect. 5.

2 Prerequisites

2.1 Notations

In this paper we use the following notations:

$\mathbb{F}_q$	finite field of size $q = 2^m$
$\boldsymbol{A}$	matrix
$\boldsymbol{I}_r$	identity matrix of size $r \times r$
$\boldsymbol{a}$	vector
$\mathsf{wt}(\boldsymbol{a})$	Hamming weight of $\boldsymbol{a}$
$d(\boldsymbol{x}, \boldsymbol{y})$	Hamming distance between $\boldsymbol{x}$ and $\boldsymbol{y}$
$(\boldsymbol{x} \| \boldsymbol{y})$	concatenation of vector $\boldsymbol{x}$ and $\boldsymbol{y}$
$(\boldsymbol{A} \| \boldsymbol{B})$	concatenation of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$
$\mathcal{H}$	hash function
$\mathbf{Diag}(\boldsymbol{a})$	Diagonal matrix from vector $\boldsymbol{a}$
$\mathcal{S}_{w,n}$	Set of binary vectors of length n and Hamming weight w

2.2 Coding Theory

Let $\mathbb{F}_q$ be a finite field with $q = 2^m$. A linear code $\mathcal{C}$ of length n and dimension k over $\mathbb{F}_q$ is a subspace of $\mathbb{F}_q^n$ of dimension k. Elements of $\mathcal{C}$ are called code words. A generator matrix of $\mathcal{C}$ is a matrix $\boldsymbol{G} \in \mathbb{F}_q^{k \times n}$ such that

$$\mathcal{C} = \left\{ \boldsymbol{m}\boldsymbol{G} \quad \text{s.t} \quad \boldsymbol{m} \in \mathbb{F}_q^k \right\}$$

and a parity check matrix of $\mathcal{C}$ is a matrix $\boldsymbol{H}$ such $\boldsymbol{H}\boldsymbol{c}^T = \boldsymbol{0}$ for all $\boldsymbol{c} \in \mathcal{C}$.

Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n$ be two vectors. The Hamming weight of $\boldsymbol{x}$ denoted by $\mathsf{wt}(\boldsymbol{x})$ corresponds to the number of nonzero components of $\boldsymbol{x}$. The Hamming distance between $\boldsymbol{x}$ and $\boldsymbol{y}$ denoted by $d(\boldsymbol{x}, \boldsymbol{y})$ is the Hamming weight of the vector $\boldsymbol{x} - \boldsymbol{y}$. The minimal distance of a code $\mathcal{C}$ denoted by $d(\mathcal{C})$ is the minimal distance between different code words. For more details on coding theory refer to [MS77].

Let $n = 2^r$ be an integer and $\boldsymbol{a} = (a_0, a_1, ..., a_{n-1}) \in \mathbb{F}_q^n$ be a nonzero vector. A square matrix $\boldsymbol{A} = (a_{i,j}) \in \mathbb{F}_q^{n \times n}$ is said dyadic of signature $\boldsymbol{a} \in \mathbb{F}_q^n$ if it is defined by:

$$a_{i,j} = a_{i \oplus j}$$

where $\oplus$ is the bitwise operation. A matrix is said quasi-dyadic when it is a block matrix where each block is a dyadic matrix. A linear code $\mathcal{C}$ is said quasi-dyadic when one of its parity check matrices is in the quasi-dyadic form.

Let $\boldsymbol{u} = (u_0, ..., u_{n-1}) \in \mathbb{F}_{2^m}^n$ and $\boldsymbol{v} = (v_0, ..., v_{s-1}) \in \mathbb{F}_{2^m}^s$ be two vectors with pairwise distinct coefficients such that $u_i - v_j \neq 0$ for all $0 \leq i \leq n-1$ and $0 \leq j \leq s-1$. The matrix $C(\boldsymbol{u}, \boldsymbol{v}) = (c_{ij})_{0 \leq i \leq n-1, 0 \leq j \leq s-1}$ such that

$c_{ij} = 1/(u_i - v_j)$ is called a Cauchy matrix. This matrix plays an important role in the design of quasi-dyadic Goppa codes. Indeed, it was shown that Goppa with a monic generator polynomial without multiple zeros has a parity check in the Cauchy form [MS77]. Moreover, recently, Barreto et Misoczki established how to design a binary Goppa code in Cauchy and dyadic form [MB09, Section 3].

A Generalized Srivastava code $\mathcal{C}$ over a $\mathbb{F}_{2^m}$ is an alternant code with a parity check matrix in the form:

$$\boldsymbol{H} = \begin{pmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \\ \vdots \\ \boldsymbol{H}_{s-1} \end{pmatrix}. \tag{1}$$

The matrices $\boldsymbol{H}_i$ are defined by (2) from $n+s$ different elements $\alpha_0, \alpha_1, ..., \alpha_{n-1}$ and $w_0, w_1, ..., w_{s-1}$ of $\mathbb{F}_{q^m}$, and n nonzero elements $z_0, z_1, ..., z_{n-1}$ of $\mathbb{F}_q$ with $n \leq q^m - s$.

$$\boldsymbol{H}_i = \begin{pmatrix} \frac{z_1}{\alpha_0 - w_i} & \cdots & \frac{z_{n-1}}{\alpha_{n-1} - w_i} \\ \frac{z_1}{(\alpha_0 - w_i)^2} & \cdots & \frac{z_{n-1}}{(\alpha_{n-1} - w_i)^2} \\ \vdots & & \vdots \\ \frac{z_1}{(\alpha_0 - w_i)^t} & \cdots & \frac{z_{n-1}}{(\alpha_{n-1} - w_i)^t} \end{pmatrix} \tag{2}$$

Dimension k and minimal distance d of $\mathcal{C}$ verify $k \geq n - mst$ and $d \geq st + 1$. It is important to note that when $t = 1$ GS codes are Goppa codes. Moreover, by reordering rows of the matrix $\boldsymbol{H}$ defined by (1) we can see that generalized Srivasta codes could be defined by a parity check matrix in the form

$$\tilde{\boldsymbol{H}} = \begin{pmatrix} \tilde{\boldsymbol{H}}_1 \\ \boldsymbol{H}_2 \\ \vdots \\ \tilde{\boldsymbol{H}}_t \end{pmatrix} \text{ where } \tilde{\boldsymbol{H}}_i = \begin{pmatrix} \frac{z_1}{(\alpha_0 - w_0)^i} & \cdots & \frac{z_{n-1}}{(\alpha_{n-1} - w_0)^i} \\ \frac{z_1}{(\alpha_0 - w_1)^i} & \cdots & \frac{z_{n-1}}{(\alpha_{n-1} - w_1)^i} \\ \vdots & & \vdots \\ \frac{z_1}{(\alpha_0 - w_{s-1})^i} & \cdots & \frac{z_{n-1}}{(\alpha_{n-1} - w_{s-1})^i} \end{pmatrix} \text{ for } i = 1, ..., t. \tag{3}$$

We can see that for constructing a generalized Srivastava code from (3), we need to compute the matrix $\tilde{\boldsymbol{H}}_1$ and the other matrices $\tilde{\boldsymbol{H}}_i$ could be obtained by raising each coefficient of $\tilde{\boldsymbol{H}}_1$ to the power of i. For more information about GS codes see [MS77].

2.3 Key Encapsulation Mechanism

A KEM is a set of four algorithms (Setup, KeyGen, Encapsulation, Decapsulation) described as follows:

- $\mathsf{Setup}(1^\lambda)$: Setup is a probabilistic algorithm that takes as in input a security parameter λ and returns public parameters PP
- $\mathsf{KeyGen}(\mathsf{PP})$: KeyGen is the key generation algorithm. It takes as input public parameters and returns a couple $(\mathsf{sk}, \mathsf{pk})$ of secret and public keys.

- Encapsulation(PP, pk): This algorithm takes as input public parameters and public key. It first generates a session key $\boldsymbol{k}$ then computes its encapsulated value $\boldsymbol{c}$. Finally, it returns $\boldsymbol{c}$.
- Decapsulation(sk, $\boldsymbol{c}$): Decapsulation is the algorithm consisting to recover a session key from an encapsulation $\boldsymbol{c}$. It takes as input an encapsulation $\boldsymbol{c}$ and a secret key sk. It returns either a session key $\boldsymbol{k}$ or the failed symbol $\perp$.

3 KEM from Binary QD-GS Codes

In this section, we first describe the key encapsulation mechanism from binary QD-GS codes and then make its security analysis.

3.1 Description

It is important to note that the scheme is built upon the Niederreiter cryptosystem thus the public key is a systematic parity check matrix. In the key generation algorithm, the process is similar to that of DAGS reloaded [BBBCDGGHKN+19]. However, the main difference is in the fact that the base field is the binary field $\mathbb{F}_2$ instead of an extension $\mathbb{F}_{2^r}$. The key generation, encapsulation, and decapsulation algorithms of binary DAGS are defined as follows:

Algorithm 1. Key Generation

Input: A finite field $\mathbb{F}_{2^m}$, nonzero integers $n = n_0 s$ and t.
Output: A public key pk and a secret key sk

1. Generate the dyadic signature $\boldsymbol{h}$
2. Construct the Cauchy support $(\boldsymbol{u}, \boldsymbol{v})$
3. Compute the Cauchy matrix $\boldsymbol{H}_1 = C(\boldsymbol{u}, \boldsymbol{v})$
4. Compute $\boldsymbol{H}_i$ for $i = 2, ..., t$ by computing the power of i of each coefficient of the matrix $\boldsymbol{H}_1$
5. Compute a vector $\boldsymbol{z}$ by sampling uniformly at random elements in $\mathbb{F}_{2^m}$ with the restriction $z_{is+j} = z_{is}$ for $i = 0, ..., n_0 - 1$, $j = 0, ..., s - 1$.
6. Compute the matrix $\tilde{\boldsymbol{H}} = \begin{pmatrix} \boldsymbol{H}_1 \\ \boldsymbol{H}_2 \\ \vdots \\ \boldsymbol{H}_t \end{pmatrix} \mathbf{Diag}(\boldsymbol{z})$
7. Split $\boldsymbol{H}_{bin}$ as $(\boldsymbol{B} \| \boldsymbol{A})$ such that $\boldsymbol{A}$ is a $mst \times mst$ invertible matrix.
8. Compute the systematic form $\tilde{\boldsymbol{H}}_{bin} = (\boldsymbol{M} \| \boldsymbol{I}) = \boldsymbol{A}^{-1} \boldsymbol{H}_{bin}$
9. Choose randomly a binary string $\boldsymbol{r} \in \mathbb{F}_2^n$.
10. Return $\mathsf{pk} = \boldsymbol{M}$ and $\mathsf{sk} = (\boldsymbol{u}, \boldsymbol{A}, \boldsymbol{r}, \boldsymbol{M})$

In the key generation algorithm the dyadic signature $\boldsymbol{h}$ is computed according to the work of Barreto and Misoczki [MB09]. The integer s represents the order of the quasi-dyadic matrix $\boldsymbol{H}_1$. Finally, n is the length of the code and t is the number of block rows in the parity check matrix of the generalized Srivastava code.

The Cauchy support $(\boldsymbol{u}, \boldsymbol{v})$ is constructed as follows:

- Choose a random offset $w \xleftarrow{\$} \mathbb{F}_{2^m}$;
- Compute $\boldsymbol{u}_i = \frac{1}{h_i} + w$ and $\boldsymbol{v}_j = \frac{1}{h_j} + \frac{1}{h_0} + w$ for $i = 0, \cdots, s-1$ and $j = 0, \cdots, n-1$;
- Set $u = (\boldsymbol{u}_0, \cdots, \boldsymbol{u}_{s-1})$ and $\boldsymbol{v} = (\boldsymbol{v}_0, \cdots, \boldsymbol{v}_{n-1})$.

Algorithm 2. Encapsulation

Input: The public key $\mathsf{pk} = \boldsymbol{M}$ where $\tilde{\boldsymbol{H}}_{bin} = (\boldsymbol{M} \| \boldsymbol{I})$ is a binary parity check matrix of a QD-GS code.
Output: A session key $\boldsymbol{k}$ and its encapsulation $\boldsymbol{c}$

1. Choose randomly an error vector $\boldsymbol{e} \xleftarrow{\$} \mathcal{S}_{w,n}$
2. Compute $\boldsymbol{c}_0 = \boldsymbol{e}_1 + \boldsymbol{M}\boldsymbol{e}_0$ and $\boldsymbol{c}_1 = \mathcal{H}(2, \boldsymbol{e})$ where $\boldsymbol{e}$ is parse as $(\boldsymbol{e}_0 \| \boldsymbol{e}_1)$.
3. Set $\boldsymbol{c} = (\boldsymbol{c}_0 \| \boldsymbol{c}_1)$
4. Compute $\boldsymbol{k} = \mathcal{H}(1, \boldsymbol{e}, \boldsymbol{c})$
5. Return the encapsulation $\boldsymbol{c}$

Algorithm 3. Decapsulation

Input: The secret key $\mathsf{sk} = (\boldsymbol{u}, \boldsymbol{A}, \boldsymbol{r}, \boldsymbol{M})$, encapsulation $\boldsymbol{c}$
Output: A session key $\boldsymbol{k}$

1. Parse $\boldsymbol{c}$ into $\boldsymbol{c} = (\boldsymbol{c}_0 \| \boldsymbol{c}_1)$
2. Obtain the syndrome $\boldsymbol{c}'_0 \in \mathbb{F}_q$ from $\boldsymbol{c}_0$.
3. Compute $\boldsymbol{e}' = \mathsf{Decode}(\mathsf{sk}, \boldsymbol{c}'_0)$ where Decode is a decoding algorithm for alternant code.
4. If decoding failed or $wt(\boldsymbol{e}') \neq w$ set $b = 0$ and $\eta = \boldsymbol{r}$
5. If $\tilde{\boldsymbol{H}}_{bin}\boldsymbol{e}' = \boldsymbol{c}_0$ and $\boldsymbol{c}_1 = \mathcal{H}(2, \boldsymbol{e}')$.
   Set $b = 1$ and $\eta = \boldsymbol{e}'$
6. Else:
   Set $b = 0$ and $\eta = \boldsymbol{r}$
7. Return $\boldsymbol{k} = \mathcal{H}(b, \eta, \boldsymbol{c})$

Description of the Decoding Algorithm
The input to the decoding algorithm is not, as commonly, a noisy codeword, but a syndrome.

The main step in the decoding algorithm involves reconstructing the alternant matrix $\boldsymbol{H}_{alt}$ and a syndrome $\boldsymbol{c}'_0$ corresponding to the alternant code. For this reconstruction, we first compute $\boldsymbol{A}\tilde{\boldsymbol{H}}_{bin} = \boldsymbol{H}_{bin}$; $\boldsymbol{A}\boldsymbol{c}_0 = \tilde{\boldsymbol{c}_0}$. Then we use the inverse of the co-trace function to transform respectively $\boldsymbol{H}_{bin}$ and $\tilde{\boldsymbol{c}_0}$ into matrix $\tilde{\boldsymbol{H}}$ and syndrome $\tilde{\boldsymbol{c}}$ with coefficients in the extension field $\mathbb{F}_{2^m}$. Finally, we compute $\boldsymbol{H}_{alt} = \boldsymbol{C}^{-1}\tilde{\boldsymbol{H}}$ and $\boldsymbol{c}'_0 = \boldsymbol{C}^{-1}\tilde{\boldsymbol{c}}$, where $\boldsymbol{C}$ is a $r \times r$ matrix such that its

r rows correspond to the coefficients of the polynomials $g_1(x), g_2(x), \cdots, g_r(x)$ defined by:

$$g_{(l-1)t+i} = \frac{\prod_{j=1}^{s}(x - \boldsymbol{u}_j)^t}{(x - \boldsymbol{u}_l)^i}$$

for $l = 1, \cdots, s$ and $i = 1, \cdots, t$. For more details on the alternant matrix reconstruction and the corresponding alternant syndrome, refer to [BBBCDGGHKN+19].

3.2 Security Analysis

Decoding Attack. In code-based cryptography, the main efficient and known decoding attack is the information set decoding (ISD) technique introduced by E. Prange [Pra62]. Other approaches such as statistical decoding [Jab01] are considered as less efficient. For a given linear code of length n and dimension k, the main idea behind the information-set decoding algorithm is to find a set of k coordinates of a garbled vector that are error-free and such that the restriction of the code's generator matrix to these positions is invertible. Then, the original message can be computed by multiplying the encrypted vector by the inverse of the submatrix.

Thus, those k bits determine the codeword uniquely, and hence the set is called an information set. It is sometimes difficult to draw the exact resistance to this type of attack. However, they are always lower-bounded by the ratio of information sets without errors to total possible information sets, i.e.,

$$R_{\text{ISD}} = \frac{\binom{n-\omega}{k}}{\binom{n}{k}},$$

where ω is the Hamming weight of the error vector. Therefore, well-chosen parameters can avoid these non-structural attacks.

Algebraic Attack. In code-based cryptography, the main key recovery attack against schemes using structured codes are that of Faugère *et al.* denoted in the literature by FOPT attack [FOPT10]. Their attack was originally aimed at two variants of McEliece-like schemes, introduced respectively in [BCGO09] and [MB09]. The first scheme based on quasi-cyclic is completely broken. The second variant, instead, only considered quasi-dyadic Goppa codes. Most of the parameters proposed in [MB09], have also been broken very easily, except for the binary case code. This is not connected to the base field but is due to the fact that with a small base field, authors provided a higher extension degree. The extension degree m plays an important role in the attack, as it defines the dimension of the solution space, which is equal, in fact, exactly to $m - 1$. However, in [FOPT13], the authors provided a bound of the complexity of their attack and showed that schemes for which this dimension is less or equal to 20 should be within the scope of the attack.

In this paper, the underlying code is Generalized Srivastava code and in [Per12], the author showed that the dimension of the solution space is $mt - 1$ instead of $m-1$. Therefore the choice of a good parameter could avoid this attack. Recently, in [FOPDPT16] an improvement of FOPT attack was introduced. Authors introduced a new technique called folding to reduce the complexity of the FOPT attack to that of attacking a smaller code (i.e. the folded code) by using the strong properties of the automorphism group of the alternant codes. However, it is important to note that there is not a clear application of this attack against GS codes and furthermore, the authors do not propose a concrete bound, but only experimental results.

During the NIST process for standardization of post-quantum public key schemes, there is only one scheme based on quasi-dyadic generalized Srivastava code presented by Banegas *et al.* [BBBCDGGHKN+18]. The specificity of this scheme is that the authors used a non-binary based field and a small degree extension compared to the scheme of [MB09]. Some proposal parameters of Banegas *et al.*'s scheme were attacked by Barelli and Couvreur in [BC18].

The Barelli-Couvreur attack is based on a novel construction called Norm-Trace Code. The construction of these codes is given explicitly only for the specific case $m = 2$ which is the case in all parameters proposed in [BBBCDGGHKN+18]. However, it is possible to avoid this attack by modifying a single parameter, namely the size q of the base field i.e by changing this value from 2^6 to 2^8. To address of the Barelli-Couvreur attack, Banegas *et al.* provided updated parameters in [BBBCDGGHKN+19] while keeping the size of the base field corresponding to the level 1 of NIST security to $q = 2^6$. This leads to a new attack introduced by Bardet *et al.* [BBCO19].

In Bardet *et al.*'s paper, they first applied the Barelli-Couvreur attack and after, presented a hybrid attack combining exhaustive search and Gröbner basis to attack the updated parameters of level 1 in [BBBCDGGHKN+19]. However, note that their attack did not concern updated parameters of NIST security levels 2 and 3 where the size of the base field is $q = 2^8$. In addition, in [BBCO19] authors showed that when the reduced system is undetermined, both attacks (i.e. of [BC18] and [BBCO19]) have no effect. Note that among all aforementioned attacks only that of Faugère *et al.*'s [FOPT10] concerns binary codes. As it is mentioned, to avoid it we need to choose parameters to have a dimension of the solution space satisfying $mt - 1 > 20$.

Proposal Parameters. The parameters used for this implementation (Table 1) are that proposed in [BBBCDGGHKN+19]. These parameters are chosen to be secure against information decoding attack as well as the folding technique [FOPDPT15] and FOPT attacks. They are chosen as follows:

- *Information set decoding attack*: for avoiding the information set decoding attack, the parameters n and k are chosen such that the ration $\frac{n}{k}$ is closed to $\frac{1}{2}$.

Table 1. Proposal parameters in [BBBCDGGHKN+19]

Security level	q	n	k	s	t	w
1	2^{13}	6400	3072	2^7	2	128
3	2^{14}	11520	4352	2^8	2	256
5	2^{14}	14080	6912	2^8	2	256

- *FOPT attack*: considering the fact that the underlying quasi-dyadic code is a generalized Srivastava code which is a generalization of Goppa code, the parameters m and t are chosen such that the dimension of solution space is larger than 20 as recommended in [FOPT13].
- *Folding technique attack*: for avoiding the folding technique attack, the quasi-dyadic order s of the underlying generalized Srivastava code is chosen such that it is not very large.

  Despite these parameter adjustments, binary DAGS still has a small public key size compared to Classic McEliece [CCUGLMMNPP+20] and BIKE [ABBBBDGGGM+17] (Table 2).

4 Efficient Implementation

In this software implementation for binary DAGS ($\mathsf{DAGS}_{\mathsf{bin}}$), we will exploit the particular structure of QD matrices to improve the multiplication of two QD matrices. Some matrix operations, such as sum or inversion, can be performed efficiently in the dyadic case by simply considering the signatures as in [BBPS20]. The multiplication operations can be significantly improved by means of LUP decomposition and the Karatsuba multiplication in the QD case. This method provides a fast software implementation of key generation and decapsulation in the binary DAGS compared to the standard method.

4.1 Implementation Details

For Key Generation. We realized that the systematization of the matrix $\boldsymbol{H}_{bin}$ represents almost the total cost of the key generation. To reduce this, we first performed a trick in our implementation. By combining Steps 6 and 7 into one and with the projection of the matrix $\tilde{\boldsymbol{H}}$ onto $\mathbb{F}_2$, we obtain a QD matrix $mst \times n$ $\boldsymbol{H}_{bin}$. Then the systematic form $\tilde{\boldsymbol{H}}_{bin}$ of $\boldsymbol{H}_{bin}$ is also a QD matrix. Therefore, instead of considering the complete $mst \times n$ matrix $\boldsymbol{H}_{bin}$, we just need the signature of each block. Thus, the first row of $\boldsymbol{H}_{bin}$ is composed of the signatures of the first blocks, the second row is obtained from the signatures of the second blocks, and so on. Finally, $\boldsymbol{H}_{bin}$ is $mt \times n$ matrix, with n_0 block-rows, where $n_0 = n/s$. As a result, the time required to systematize the matrix $\boldsymbol{H}_{bin}$ decreases significantly.

Second, in Step 9, we used the efficient inversion of the secret matrix $\boldsymbol{A}$ as in [BBPS20] by merging the LUP decomposition and Karatsuba multiplication.

This improved inversion method reduces the execution time of the key generation by almost 10 times for the level security 1 ($\mathsf{DAGS}_{\mathsf{bin}}\_1$) for example.

Table 2. Public key size in bytes for level security 3

Algorithm	BIKE [ABBBBDGGGM+17]	HQC [MABBBBDDGL+20]	Classic McEliece [CCUGLMMNPP+20]	DAGS reloaded	$\mathsf{DAGS}_{\mathsf{bin}}\_3$
Public Key	24 659	4 522	524 160	11 264	15 232

For Encapsulation. This is faster than the key generation and the decapsulations algorithms. It is just composed by a binary matrix-vector product and hash computation. We just cleaned up the C code compared to the previous implementations.

For Decapsulation. For the decapsulation we need to reconstruct the alternant syndrome $\boldsymbol{c}_0'$ and the alternant secret matrix $\boldsymbol{H}_{alt}$ from the secret key [BBBCDGGHKN+19]. We observe that this step consumes almost half of the execution time in the decapsulation. Therefore, we first compute $\boldsymbol{H}_{bin} = \boldsymbol{A}\tilde{\boldsymbol{H}}_{bin}$. Here we use the Karatsuba multiplication technique in the QD case to save time in our implementation.

Then, from $\boldsymbol{H}_{bin}$, we apply the inverse of the co-trace function to obtain the matrix $\tilde{\boldsymbol{H}}$ with coefficients in the extension field. We compute the matrix $\boldsymbol{C}$ from the support $\boldsymbol{u}$.

Finally, we compute $\boldsymbol{H}_{alt} = \boldsymbol{C}^{-1}\tilde{\boldsymbol{H}}$ using the same technique in Step 9 in the key Generation. The LUP decomposition factorizes the matrix $\boldsymbol{C}$ as LUP. This procedure consists in using a block decomposition, which works directly on the signatures, to exploit the simple and efficient algebra of QD matrices. Once the factorization of $\boldsymbol{C}$ is obtained, it is sufficient to perform the computation of $\boldsymbol{C}^{-1}$ in an efficient way and use the Karatsuba method to perform the product.

All these techniques could have allowed us to make the decapsulation very fast. Unfortunately, we could not generate the session key correctly for some tests with the version of the binary DAGS ($\mathsf{DAGS}_{\mathsf{bin}}\_5$) of level security 5. Therefore we cannot present the runtime for this version in the results.

In the following, we call the version of the binary DAGS with the application of the above techniques for key generation and decapsulation, $\mathsf{DAGS}_{\mathsf{bin}}$ improved.

Table 3. Timings with previous codes for security level 1

Algorithm	DAGS reloaded	$\mathsf{DAGS}_{\mathsf{bin}}\_1$	$\mathsf{DAGS}_{\mathsf{bin}}\_1$ improved
Key Generation	408 342 881	679 076 980	77 768 093
Encapsulation	5 061 697	6 564 782	4 641 252
Decapsulation	192 083 862	298 987 096	15 091 4566

Table 4. Timings with previous codes for security level 3

Algorithm	DAGS reloaded	$\mathsf{DAGS}_{\mathsf{bin}}$_3	$\mathsf{DAGS}_{\mathsf{bin}}$_3 improved
Key Generation	1 560 879 328	1 597 980 876	847 980 876
Encapsulation	14 405 500	15 200 232	8 020 732
Decapsulation	392 435 142	454 765 478	34 656 844

4.2 Results

In this section, we present the results obtained in our implementation in C. The timings were acquired by running the code 10 times and taking the average. We used CLANG compiler version 8.0.0 and the processor was an Intel(R) Core(TM) i5-5300U CPU @ 2.30 GHz.

We present below the number of cycles obtained for our binary DAGS implementation with the improvements in key generation and decapsulation compared to previous implementations (Table 3 and Table 4). Tables 5 and 6 compare our implementation with the other NIST finalists.

Table 5. Timings with NIST finalists for security level 1

Algorithm	BIKE [ABBBBDGGGM+17]	HQC [MABBBBDDGL+20]	Classic McEliece [CCUGLMMNPP+20]	$\mathsf{DAGS}_{\mathsf{bin}}$_1 improved
Key Generation	650 638	98 570	49 758 742	77 768 093
Encapsulation	247 976	356 980	56 672	4 741 252
Decapsulation	2 575 687	467 891	253 864	15 091 4566

Table 6. Timings with NIST finalists for security level 3

Algorithm	BIKE [ABBBBDGGGM+17]	HQC [MABBBBDDGL+20]	Classic McEliece [CCUGLMMNPP+20]	$\mathsf{DAGS}_{\mathsf{bin}}$_3 improved
Key Generation	3 674 894	267 983	364 756 564	847 980 876
Encapsulation	564 896	567 836	245 794	8 020 732
Decapsulation	4 298 673	947 920	387 678	34 656 844

5 Conclusion

This paper is an extension of the work of Banegas *et al.* [BBBCDGGHKN+19] on DAGS. We first established that the Barelli-Couvreur and Bardet *et al.* attacks have no effect against binary DAGS. This is a code-based KEM scheme using quasi-dyadic binary generalized Srivastava codes. We have provided parameters that are secure against folding technique and FOPT attacks *et al.* [BBBCDGGHKN+19]. Despite these parameter adjustments, binary DAGS still has a small public key size. Then, we realized an efficient software implementation of the binary DAGS using tricks to reduce the computational time mainly

in the key generation and decapsulation algorithms. We also used LUP decomposition and Karatsuba multiplication techniques in the case of quasi-dyadic matrices. This allowed us to have a competitive runtime performance compared to other code-based NIST finalists. Finally, the high execution time in binary DAGS compared to DAGS reloaded is related to the choice of parameters that are very large (e.g. $n = 6400$ security level 1). We need to reduce the parameters in binary DAGS while being aware of existing attacks. In this way, we will be able to achieve even better performance. This work is currently in progress and the results will appear in our future work.

References

[ABBBBDGGGM+17] Aragon, N., et al.: BIKE: Bit Flipping Key Encapsulation (2017). https://bikesuite.org/files/v4.2/BIKE_Spec021.09.29.1.pdf. Accessed 09 Dec 2022

[BBBCDGGHKN+18] Banegas, G., et al.: DAGS: key encapsulation using dyadic GS codes. J. Math. Cryptol. **12**(4), 221–239 (2018)

[BBBCDGGHKN+19] Banegas, G., et al.: DAGS: reloaded revisiting dyadic key encapsulation. In: Baldi, M., Persichetti, E., Santini, P. (eds.) CBC 2019. LNCS, vol. 11666, pp. 69–85. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25922-8_4

[BBCO19] Bardet, M., Bertin, M., Couvreur, A., Otmani, A.: Practical algebraic attack on DAGS. In: Baldi, M., Persichetti, E., Santini, P. (eds.) CBC 2019. LNCS, vol. 11666, pp. 86–101. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25922-8_5

[BBPS20] Banegas, G., Barreto, P.S., Persichetti, E., Santini, P.: Designing efficient dyadic operations for cryptographic applications. J. Math. Cryptol. **14**(1), 95–109 (2020)

[BC18] Barelli, É., Couvreur, A.: An efficient structural attack on NIST submission DAGS. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11272, pp. 93–118. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03326-2_4

[BCGO09] Berger, T.P., Cayrel, P.-L., Gaborit, P., Otmani, A.: Reducing key length of the McEliece cryptosystem. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 77–97. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02384-2_6

[CCUGLMMNPP+20] Chou, T., et al.: Classic McEliece: conservative code-based cryptography (2020). https://classic.mceliece.org/nist/mceliece-20201010.pdf. Accessed 09 Dec 2022

[FOPDPT15] Faugère, J.-C., Otmani, A., Perret, L., De Portzamparc, F., Tillich, J.-P.: Folding alternant and Goppa codes with non-trivial automorphism groups. IEEE Trans. Inf. Theory **62**(1), 184–198 (2015)

[FOPDPT16] Faugère, J.-C., Otmani, A., Perret, L., De Portzamparc, F., Tillich, J.-P.: Structural cryptanalysis of McEliece schemes with compact keys. Des. Codes Cryptography **79**(1), 87–112 (2016)

[FOPT10] Faugère, J.-C., Otmani, A., Perret, L., Tillich, J.-P.: Algebraic cryptanalysis of McEliece variants with compact keys. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 279–298. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13190-5_14

[FOPT13] Faugère, J.-C., Otmani, A., Perret, L., Tillich, J.-P.: Algebraic cryptanalysis of compact McEliece's variants- toward a complexity analysis. In: Conference on Symbolic Computation and Cryptography, p. 45 (2013)

[Jab01] Jabri, A.A.: A statistical decoding algorithm for general linear block codes. In: Honary, B. (ed.) Cryptography and Coding 2001. LNCS, vol. 2260, pp. 1–8. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45325-3_1

[MABBBBDDGL+20] Melchor, C.A., et al.: Hamming quasi-cyclic (HQC) (2020). https://pqc-hqc.org/doc/hqc-specification_2020-10-01.pdf. Accessed 09 Dec 2022

[MB09] Misoczki, R., Barreto, P.S.L.M.: Compact McEliece keys from Goppa codes. In: Jacobson, M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, pp. 376–392. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05445-7_24

[MS77] MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error Correcting Codes, vol. 16. Elsevier (1977)

[Per12] Persichetti, E.: Compact McEliece keys based on quasidyadic Srivastava codes. J. Math. Cryptol. **6**(2), 149–169 (2012)

[Pra62] Prange, E.: The use of information sets in decoding cyclic codes. IRE Trans. Inf. Theory **8**(5), 5–9 (1962)