DAGS: Reloaded
Revisiting Dyadic Key Encapsulation

Gustavo Banegas[1], Paulo S. L. M. Barreto[2], Brice Odilon Boidje[3], Pierre-Louis Cayrel[4], Gilbert Ndollane Dione[3], Kris Gaj[5], Cheikh Thiécoumba Gueye[3], Richard Haeussler[5], Jean Belo Klamti[3], Ousmane N'diaye[3], Duc Tri Nguyen[5], Edoardo Persichetti[6], and Jefferson E. Ricardini[7]

[1] Tecnische Universiteit Eindhoven, The Netherlands
[2] University of Washington Tacoma, USA
[3] Laboratoire d'Algebre, de Cryptographie, de Géométrie Algébrique et Applications, Université Cheikh Anta Diop, Dakar, Senegal
[4] Laboratoire Hubert Curien, Université Jean Monnet, Saint-Etienne, France
[5] George Mason University, USA
[6] Department of Mathematical Sciences, Florida Atlantic University, USA
[7] Universidade de São Paulo, Brazil.

Abstract. In this paper we revisit some of the main aspects of the DAGS Key Encapsulation Mechanism, one of the code-based candidates to NIST's standardization call for the key exchange/encryption functionalities. In particular, we modify the algorithms for key generation, encapsulation and decapsulation to fit an alternative KEM framework, and we present a new set of parameters that use binary codes. We discuss advantages and disadvantages for each of the variants proposed.

Keywords: post-quantum cryptography, code-based cryptography, key exchange.

1 Introduction

The majority of cryptographic protocols in use in the present day are based on traditional problems from number theory such as factoring or computing discrete logarithms in some group; this is the case for schemes such as RSA, DSA, ECDSA etc. This is undoubtedly about to change due to the looming threat of quantum computers. In fact, due to Shor's seminal work [21], such problems will be vulnerable to polynomial-time attacks once quantum computers with enough computational power are available, which will make current cryptographic solutions obsolete. While the resources necessary to effectively run Shor's algorithm on actual cryptographic parameters (or other cryptographically relevant

quantum algorithms such as or Grover's [12]) might be at least a decade away, post-quantum cryptography cannot wait for this to happen. In fact, today's encrypted communication could be easily stored by attackers and decrypted later with a quantum computer, compromising secrets that aim for long-term security. Therefore, it is vital that the time required to develop such resources (t_{Dev}) is not inferior to the sum of the time required to develop and deploy new cryptographic standards (t_{Dep}), and the desired lifetime of a secret (t_{Sec}), i.e. we need to ensure $t_{Dev} \geq t_{Dep} + t_{Sec}$.

With this in mind, the National Institute of Standards and Technology (NIST) has launched a Call for Proposals for Post Quantum Cryptographic Schemes [17], to select a range of post-quantum primitives to become the new public-key cryptography standard. The NIST Call is soliciting proposals in encryption, key exchange, and digital signature schemes. It is expected that the effort will require approximately 5 years, and another 5 years will likely follow for the deployment phase, which includes developing efficient implementations and updating the major cryptographic products to the new standard. Currently, there are five major families of post-quantum cryptosystems: lattice-based, code-based, hash-based, isogeny-based, and multivariate polynomial-based systems. The first two are the most investigated, comprising nearly three quarters of the total amount of submissions to the NIST competition (45 out of 69).

Our Contribution DAGS [1] was one of the candidates to NIST's Post-Quantum Standardization call [17]. The submission presents a code-based Key Encapsulation Mechanism (KEM) that uses Quasi-Dyadic (QD) Generalized Srivastava (GS) codes to achieve very small sizes for all the data (public and private key, and ciphertext); as a result, DAGS features one of the smallest data size among all the code-based submissions. Unfortunately, due to some security concerns, DAGS was not selected for the second round of the standardization call. In this paper, we present the results of our investigation of the DAGS scheme, aimed at tweaking and improving several aspects of the scheme. First, we describe a new approach to the protocol design (Section 3), which offers an important alternative and a tradeoff between security and performance. Then, in Section 4, we propose and discuss new parameters, including an all-new set based on binary codes, to protect against both known and new attacks. Finally, in Section 5, we report the numbers obtained in a new, improved implementation which uses dedicated techniques and tricks to achieve a considerable speed up. We hope this version of DAGS can provide some stability and reassurance about the security of the system.

2 Preliminaries

2.1 Notation

We will use the following conventions throughout the rest of the paper:

Table 1: Notation used in this document.

a	a constant
$\boldsymbol{a}$	a vector
A	a matrix
$\mathcal{A}$	an algorithm or (hash) function
A	a set
$(\boldsymbol{a} \parallel \boldsymbol{b})$	the concatenation of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
$\mathrm{Diag}(\boldsymbol{a})$	the diagonal matrix formed by the vector $\boldsymbol{a}$
I_n	the $n \times n$ identity matrix
$\xleftarrow{\$}$	choosing a random element from a set or distribution
ℓ	the length of a shared symmetric key

2.2 Linear Codes

We briefly recall some fundamental notions from Coding Theory. The *Hamming weight* of a vector $\boldsymbol{x} \in \mathbb{F}_q^n$ is given by the number $\mathsf{wt}(\boldsymbol{x})$ of its nonzero components. We define a linear code using the metric induced by the Hamming weight.

Definition 1. *An* $[n, k]$-linear code C *of length* n *and dimension* k *over* $\mathbb{F}_q$ *is a* k*-dimensional vector subspace of* $\mathbb{F}_q^n$*.*

A linear code can be represented by a matrix $G \in \mathbb{F}_q^{k \times n}$, called *generator matrix*, whose rows form a basis for the vector space defining the code. Alternatively, a linear code can also be represented as kernel of a matrix $H \in \mathbb{F}_q^{(n-k) \times n}$, known as *parity-check matrix*, i.e. $\mathsf{C} = \{\boldsymbol{c} : H\boldsymbol{c}^T = 0\}$. Thanks to the generator matrix, we can easily define the codeword corresponding to a vector $\boldsymbol{\mu} \in \mathbb{F}_q^k$ as $\boldsymbol{\mu}G$. Finally, we call *syndrome* of a vector $\boldsymbol{c} \in \mathbb{F}_q^n$ the vector $H\boldsymbol{c}^T$.

2.3 Structured Matrices and GS Codes

Definition 2. *Given a ring* R *(in our case the finite field* $\mathbb{F}_{q^m}$*) and a vector* $\boldsymbol{h} = (h_0, \ldots, h_{n-1}) \in \mathsf{R}^n$*, the dyadic matrix* $\Delta(\boldsymbol{h}) \in \mathsf{R}^{n \times n}$ *is the symmetric matrix with components* $\Delta_{ij} = h_{i \oplus j}$*, where* $\oplus$ *stands for bitwise exclusive-or on the binary representations of the indices. The sequence* $\boldsymbol{h}$ *is called its signature. Moreover,* $\Delta(r, \boldsymbol{h})$ *denotes the matrix* $\Delta(\boldsymbol{h})$ *truncated to its first* r *rows. Finally, we call a matrix quasi-dyadic if it is a block matrix whose component blocks are* $r \times r$ *dyadic submatrices.*

If n is a power of 2, then every $2^l \times 2^l$ dyadic matrix can be described recursively as

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

where each block is a $2^{l-1} \times 2^{l-1}$ dyadic matrix. Note that by definition any 1 $\times$ 1 matrix is trivially dyadic.

Definition 3. *For $m, n, s, t \in \mathbb{N}$ and a prime power q, let $\alpha_1, \ldots, \alpha_n$ and $w_1, \ldots, w_s$ be $n + s$ distinct elements of $\mathbb{F}_{q^m}$, and $z_1, \ldots, z_n$ be nonzero elements of $\mathbb{F}_{q^m}$. The* Generalized Srivastava (GS) code *of order st and length n is defined by a parity-check matrix of the form:*

$$H = \begin{pmatrix} H_1 \\ H_2 \\ \vdots \\ H_s \end{pmatrix}$$

where each block is given by

$$H_i = \begin{pmatrix} \frac{z_1}{\alpha_1 - w_i} & \cdots & \frac{z_n}{\alpha_n - w_i} \\ \frac{z_1}{(\alpha_1 - w_i)^2} & \cdots & \frac{z_n}{(\alpha_n - w_i)^2} \\ \vdots & \vdots & \vdots \\ \frac{z_1}{(\alpha_1 - w_i)^t} & \cdots & \frac{z_n}{(\alpha_n - w_i)^t} \end{pmatrix}.$$

The parameters for such a code are the length $n \leq q^m - s$, dimension $k \geq n - mst$ and minimum distance $d \geq st + 1$. GS codes are part of the family of alternant codes, and therefore benefit of an efficient decoding algorithm; according to Sarwate [20, Cor. 2] the complexity of decoding is $\mathcal{O}(n \log^2 n)$, which is the same as for Goppa codes. Moreover, it can be easily proved that every GS code with $t = 1$ is a Goppa code. More information about this class of codes can be found in [15, Ch. 12, §6].

3 SimpleDAGS

3.1 Construction

The core idea of DAGS is to use GS codes which are defined by matrices in quasi-dyadic form. It can be easily proved that every GS code with $t = 1$ is a Goppa code, and we know [15, Ch. 12, Pr. 5] that Goppa codes admit a parity-check matrix in Cauchy form under certain conditions (the generator polynomial has to be monic and without multiple zeros). By Cauchy we mean a matrix $C(\boldsymbol{u}, \boldsymbol{v})$ with components $C_{ij} = \frac{1}{u_i - v_j}$. Misoczki and Barreto showed in [16, Th. 2] that the intersection of the set of Cauchy matrices with the set of dyadic matrices is not empty if the code is defined over a field of characteristic 2, and the dyadic signature $\boldsymbol{h} = (h_0, \ldots, h_{n-1})$ satisfies the following "fundamental" equation

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

On the other hand, it is evident from Definition 3 that if we permute the rows of H to constitute $s \times n$ blocks of the form

$$\hat{H}_i = \begin{pmatrix} \frac{z_1}{(\alpha_1 - w_1)^i} & \cdots & \frac{z_n}{(\alpha_n - w_1)^i} \\ \frac{z_1}{(\alpha_1 - w_2)^i} & \cdots & \frac{z_n}{(\alpha_n - w_2)^i} \\ \vdots & \vdots & \vdots \\ \frac{z_1}{(\alpha_1 - w_s)^i} & \cdots & \frac{z_n}{(\alpha_n - w_s)^i} \end{pmatrix}$$

we obtain an equivalent parity-check matrix for a GS code, given by

$$\hat{H} = \begin{pmatrix} \hat{H}_1 \\ \hat{H}_2 \\ \vdots \\ \hat{H}_t \end{pmatrix}.$$

The key generation process exploits first of all the fundamental equation to build a Cauchy matrix. The matrix is then successively powered (element by element) forming several blocks which are superimposed and then multiplied by a random diagonal matrix. Thanks to the observation above, we have now formed the matrix $\hat{H}$, where for ease of notation we use $\boldsymbol{u}$ and $\boldsymbol{v}$ to denote the vectors of elements $w_1, \ldots, w_s$ and $\alpha_1, \ldots, \alpha_n$, respectively. Finally, the resulting matrix is projected onto the base field (as usual for alternant codes) and row-reduced to systematic form to form the public key. The process is essentially the same as in [19], to which we refer the readers looking for additional details about dyadic GS codes and key generation.

3.2 The New Algorithms

The DAGS algorithms, as detailed in the original proposal submitted to the first Round [8], follow the "Randomized McEliece" paradigm of Nojima et al. [18], which is built upon the McEliece cryptosystem. The fact that this version of McEliece is proved to be IND-CPA secure makes it so that the resulting KEM conversion achieves IND-CCA security tightly, as detailed in [13]. However, to apply the conversion correctly, it is necessary to use multiple random oracles. These are needed to produce the additional randomness required by the paradigm, as well as to convert McEliece into a deterministic scheme (by generating a low-weight error vector from a random seed) and to obtain an additional hash output for the purpose of plaintext confirmation. Even though, in practice, such random oracles are realized using the same hash function (the *KangarooTwelve* function [14] from the Keccak family), the protocol's description ends up being quite cumbersome and hard to follow.

A simpler protocol can be obtained, although, as we will see, not without consequences, using the Niederreiter cryptosystem. We report the new description below, following the same conventions used in the original DAGS specification [1]. Therefore, system parameters are the code length n, dimension k and co-dimension r, the values s and t which define a GS code, the cardinality of the base field q and the degree of the field extension m. Note that, unlike the original DAGS, we do not need the sub-parameters k' and k''.

Algorithm 1. Key Generation

Key generation follows closely the process described in the original DAGS Key Generation. We present here a compact version, and we refer the reader to the description in Section 3.1.1 of [8] for further details.

1. Generate dyadic signature $\boldsymbol{h}$.
2. Build the Cauchy support $(\boldsymbol{u}, \boldsymbol{v})$.
3. Form Cauchy matrix $\hat{H}_1 = C(\boldsymbol{u}, \boldsymbol{v})$.
4. Build $\hat{H}_i$, $i = 2, \dots t$, 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 vector $\boldsymbol{z}$ by sampling uniformly at random elements in $\mathbb{F}_{q^m}$ with the restriction $z_{is+j} = z_{is}$ for $i = 0, \dots, n_0 - 1$, $j = 0, \dots, s - 1$.
7. Form $H = \hat{H} \cdot \mathrm{Diag}(\boldsymbol{z})$.
8. Project H onto $\mathbb{F}_q$ using the co-trace function: call this H_{base}.
9. Write H_{base} as $(B \mid A)$, where A is $r \times r$.
10. Get systematic form[1] $(M \mid I_r) = A^{-1} H_{base}$: call this $\tilde{H}$.
11. Sample a uniform random string $\boldsymbol{r} \in \mathbb{F}_q^n$.
12. The public key is the matrix $\tilde{H}$.
13. The private key consists of $(\boldsymbol{u}, A, \boldsymbol{r})$ and $\tilde{H}$.

The main differences are as follows. First of all, the public key consists of the systematic parity-check matrix $\tilde{H} = (M \mid I_r)$, rather than the generator matrix $G = (I_k \mid M^T)$. Also, the private key only stores $\boldsymbol{u}$ instead of $\boldsymbol{v}$ and $\boldsymbol{y}$, but it includes additional elements, namely the random string $\boldsymbol{r}$, the submatrix A and $\tilde{H}$ itself[2].

[1] If A is not invertible, abort and go back to 1.

[2] This is mostly a formal difference, since $\tilde{H}$ is in fact the public key.

Algorithm 2. Encapsulation

Encapsulation uses a hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\ell$ to extract the desired symmetric key, ℓ being the desired bit length (commonly 256). The function is also used to provide plaintext confirmation by appending an additional hash value, as detailed below.

1. Sample $\boldsymbol{e} \xleftarrow{\$} \mathbb{F}_q^n$ of weight w.
2. Set $\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ where $\boldsymbol{c}_0 = \tilde{H}\boldsymbol{e}$ and $\boldsymbol{c}_1 = \mathcal{H}(2, \boldsymbol{e})$.
3. Compute $\boldsymbol{k} = \mathcal{H}(1, \boldsymbol{e}, \boldsymbol{c})$.
4. Output ciphertext $\boldsymbol{c}$; the encapsulated key is $\boldsymbol{k}$.

Algorithm 3. Decapsulation

As in every code-based scheme, the decapsulation algorithm consists mainly of decoding; in this case, like in the original DAGS version, we call upon the alternant decoding algorithm (see for example [15]).

1. Get syndrome $\boldsymbol{c}_0'$ corresponding to matrix[3] H' from private key[4].
2. Decode $\boldsymbol{c}_0$ and obtain $\boldsymbol{e}'$.
3. If decoding fails or $\mathsf{wt}(\boldsymbol{e}') \neq w$, set $b = 0$ and $\boldsymbol{\eta} = \boldsymbol{r}$.
4. Check that $\tilde{H}\boldsymbol{e}' = \boldsymbol{c}_0$ and $\mathcal{H}(2, \boldsymbol{e}') = \boldsymbol{c}_1$. If so, set $b = 1$ and $\boldsymbol{\eta} = \boldsymbol{e}'$.
5. Otherwise, set $b = 0$ and $\boldsymbol{\eta} = \boldsymbol{r}$.
6. The decapsulated key is $\boldsymbol{k} = \mathcal{H}(b, \boldsymbol{\eta}, \boldsymbol{c})$.

The description we just presented is conform to the guidelines detailed by the "SimpleKEM" construction of [5], hence our choice to call this new version "SimpleDAGS". This is one of two aspects in which this variant diverges substantially from the original; we will discuss advantages (and disadvantages) of this new paradigm in the next section. The other different aspect is that using Niederreiter requires a different strategy for decoding, which we describe below.

3.3 Decoding from a Syndrome

In the original version of DAGS, the input to the decoding algorithm is, as is commonly the case is coding theory, a noisy codeword. The alternant decoding algorithm consists of three distinct steps. First, it is necessary to compute the syndrome of the received word, with respect to the alternant parity-check matrix; this is represented as a polynomial $S(z)$. Then, the algorithm uses the syndrome to compute the *error locator polynomial* $\sigma(z)$ and the *error evaluator polynomial* $\omega(z)$, by solving the *key equation* $\omega(z)/\sigma(z) = S(z) \mod z^r$. Finally, finding the

[3] In alternant form.
[4] See next section for details.

roots of the two polynomials reveals, respectivelly, the locations and values (if the code is not binary) of the errors. Actually, as shown in Section 6.3 of [1], it is possible to speed up decapsulation by incorporating the first step of the decoding algorithm in the reconstruction of the alternant matrix, i.e. the syndrome is computed *on the fly*, while the alternant matrix is built.

We are now ready to explain how to perform alternant decoding when the input is a syndrome, rather than a noisy codeword, as is the case in Algorithm 3 above. In this case, we do not need to reconstruct the alternant matrix itself, but rather to transform the received syndrome to the syndrome corresponding to the alternant matrix. This consists of two steps. First, remember that the public key $\tilde{H}$ is the systematic form of the matrix H_{base}. This is obtained from the quasi-dyadic parity-check matrix H, whose entries are in $\mathbb{F}_{q^m}$, by projecting it onto the base field $\mathbb{F}_q$. The projection is performed using the co-trace function and a basis for the extension field, say $\{\beta_1, \ldots, \beta_m\}$. Recall that the co-trace function works similarly to the trace function, by writing each element of $\mathbb{F}_{q^m}$ as a vector whose components are the coefficients with respect to the basis $\{\beta_1, \ldots, \beta_m\}$. However, instead of writing the components on m successive rows, the co-trace function distributes them over the rows at regular intervals, r at a time. More precisely, if $\boldsymbol{a} = (a_1, a_2, \ldots, a_r)^T$ is a column of H, the corresponding column $\boldsymbol{a}' = (a'_1, a'_2, \ldots, a'_{rm})^T$ of H_{base} will be formed by writing the components of each a_i in positions $a'_i, a'_{r+i}, \ldots, a'_{r(m-1)+i}$, for all $i = 1, \ldots, m$.

The first step consists of transforming the received syndrome $\boldsymbol{c}_0 = \tilde{H}\boldsymbol{e}$ into $H\boldsymbol{e}$. For this, we need to multiply the syndrome by A to obtain $A\tilde{H}\boldsymbol{e} = AA^{-1}H_{base}\boldsymbol{e} = H_{base}\boldsymbol{e}$. Then we reverse the projection process and "bring back" the syndrome on the extension field. This is immediate when operating directly on the matrices, but a little less intuitive when starting from a syndrome. It turns out that it is still possible to do that, by using again the basis $\{\beta_1, \ldots, \beta_m\}$. Namely, it is enough to collect all the components $s_i, s_{r+i}, \ldots, s_{r(m-1)+i}$ of the syndrome $\boldsymbol{s} = H_{base}\boldsymbol{e}$ and multiply the resulting vector with the vector $(\beta_1, \ldots, \beta_m)$. This maps the vector of components back to its corresponding element in $\mathbb{F}_{q^m}$ and it is immediate to check that this process yields $H\boldsymbol{e}$.

The second step consists of relating the newly-obtained syndrome to the alternant parity-check matrix H'. Since this is just another parity-check for the same code, it is possible to obtain one from the other via an invertible matrix. In particular, for GS codes we have $H = CH'$, where the $r \times r$ matrix C can be obtained using $\boldsymbol{u}$. Namely, the r rows of C corresponds to the coefficients of the polynomials $g_1(x), \ldots, g_r(x)$, where we have

$$g_{(l-1)t+i} = \frac{\prod_{j=1}^{s}(x - u_j)^t}{(x - u_l)^i}$$

for $l = 1, \ldots, s$ and $i = 1, \ldots, t$. To complete the second step, then, it is enough to compute C and then $C^{-1}H\boldsymbol{e}$. The resulting syndrome is ready to be decoded.

3.4 Consequences

There are some notable consequence to keep in mind when switching to the SimpleDAGS variant. First of all, the change in the KEM conversion not only makes the protocol simpler, but has additional advantages. The reduction is tight in the ROM, and the introduction of the plaintext confirmation step provides an extra layer of defense, at the cost of just one additional hash value. This is similar to what done in the Classic McEliece submission [7]. Moreover, the use of *implicit rejection* and a "quiet" KEM (i.e. such that the output is always a session key) further simplifies the API, and is an incentive to design constant-time algorithms, without needing extra machinery or stronger assumptions, as explained in Sections 14 and 15 of [5].

On the other hand, using Niederreiter has a negative impact on the overall performance of the scheme. The cost of the first step of decoding, detailed above, is comparable to that of reconstructing H' (and computing the syndrome) in the original DAGS, but there is an additional cost in the multiplication by A. Moreover, inverting the matrix C in the second step is expensive, and would slow down decapsulation considerably. In alternative, one could delegate some computation time to the key generation algorithm, and store C^{-1} as private key; this would preserve the efficiency of the decapsulation but noticeably increase the size of the private key. Either way, there is a clear a tradeoff at hand, sacrificing performance and efficiency in favor of a simpler description and tighter security. It therefore falls to the user's discretion whether original DAGS or SimpleDAGS is the best variant to be employed for the purpose.

4 Improved Resistance

It is natural to think that introducing additional algebraic structure like QD in a scheme based on algebraic codes (such as Goppa or GS) can give an adversary more power to perform a structural attack. This is the case of the well-known FOPT attack [10], and successive variants [11,9], which exploit this algebraic structure to solve a multivariate system of equations and reconstruct an alternant matrix which is equivalent to the private key. A detailed analysis of such attacks, and countermeasures, is given in the original DAGS paper. Recently, Barelli and Couvreur presented a structural attack aimed precisely at DAGS [4], which is very successful against the original parameters. We discuss it here.

4.1 The Barelli-Couvreur Attack

The attack makes use of a novel construction called *Norm-Trace Codes*. As the name suggests, these codes are the result of the application of both the Trace and the Norm operation to a certain support vector, and they are alternant codes. In particular, they are subfield subcodes of Reed-Solomon codes. The construction of these codes is given explicitly only for the specific case $m = 2$ (as is the case in all DAGS parameters), i.e. the support vector has components in $\mathbb{F}_{q^2}$, in which case the norm-trace code is defined as

$$\mathcal{NT}(\mathbf{x}) = \langle \mathbf{1}, Tr(\mathbf{x}), Tr(\alpha\mathbf{x}), N(\mathbf{x}) \rangle$$

where α is an element of trace 1.

The main idea is that there exists a specific norm-trace code that is the *conductor* of the secret subcode into the public code. By "conductor" the authors refer to the largest code for which the Schur product (i.e. the component-wise product of all codewords, denoted by $\star$) is entirely contained in the target, i.e.

$$Cond(\mathcal{D}, \mathcal{C}) = \{\mathbf{u} \in \mathbb{F}_q^n : \mathbf{u} \star \mathcal{D} \subseteq \mathcal{C}\}$$

The authors present two strategies to determine the secret subcode. The first strategy is essentially an exhaustive search over all the codes of the proper co-dimension. This co-dimension is given by $2q/s$, since s is the size of the permutation group of the code, which is non-trivial in our case due to the code being quasi-dyadic. While such a brute force in principle would be too expensive, the authors present a few refinements that make it feasible, which include an observation on the code rate of the codes in use, and the use of shortened codes. The second strategy, instead, consists of solving a bilinear system, which is obtained using the parity-check matrix of the public code and treating as unknowns the elements of a generator matrix for the secret code (as well as the support vector $\mathbf{x}$). This system is solved using Gröbner bases techniques, and benefits from a reduction in the number of variables similar to the one performed in FOPT, as well as the refinements mentioned above (shortened codes).

In both cases, it is easy to deduce that the two parameters q and s are crucial in determining the cost of running this step of the attack, which dominates the overall cost. In fact, the authors are able to provide an accurate complexity analysis for the first strategy which confirms this intuition. The average number of iterations of the brute force search is given by q^{2c}, where c is exactly the co-dimension described above, i.e. $c = 2q/s$. In addition, it is shown that the cost of computing Schur products is $2n^3$ operations in the base field. Thus, the overall cost of the recovery step is $2n^3q^{4q/s}$ operations in $\mathbb{F}_q$. The authors then argue that wrapping up the attack has negligible cost, and that q-ary operations can be done in constant time (using tables) when q is not too big. All this leads to a complexity which is below the desired security level for all of the DAGS parameters that had been proposed at the time of submission. We report these numbers below.

Table 2: Early DAGS Parameters.

Security Level	q	m	n	k	s	t	w	BC 1	BC 2
1	2^5	2	832	416	2^4	13	104	2^{70}	2^{44}
3	2^6	2	1216	512	2^5	11	176	2^{80}	2^{44}
5	2^6	2	2112	704	2^6	11	352	2^{55}	2^{33}

In the last two columns we report, respectively, the complexity of the attack when running it with the first approach (exhaustive search) and the cycle count for the execution of the attack with the second approach (Gröbner bases). The latter was reported in a follow-up paper by Bardet, Bertin, Couvreur and Otmani [3]. As it is possible to observe, the attack complexity is especially low for the last set of parameters since the dyadic order s was chosen to be 2^6, and this is probably too much to provide security against this attack. Still, we point out that, at the time this parameters were proposed, there was no indication this was the case, since this attack is using an entirely new technique, and it is unrelated to the FOPT and folding attacks that we just described.

While the attack performs very well against the original DAGS parameter sets, it is overall not as critical as it appears. In fact, it is shown in Section 5.3 of [1] how this can be defeated even by modifying a single parameter, namely the size of the base field q. This is the case for DAGS_3, where changing this value from 2^6 to 2^8 is enough to push the attack complexity beyond the claimed security level. Updated parameters were introduced in [1], and we report them below.

Table 3: Updated DAGS Parameters.

Security Level	q	m	n	k	s	t	w	BC 1
1	2^6	2	832	416	2^4	13	104	$\approx 2^{128}$
3	2^8	2	1216	512	2^5	11	176	$\approx 2^{288}$
5	2^8	2	1600	896	2^5	11	176	$\approx 2^{289}$

Table 4: Memory Requirements (bytes).

Parameter Set	Public Key	Private Key	Ciphertext
DAGS_1	8112	2496	656
DAGS_3	11264	4864	1248
DAGS_5	19712	6400	1632

Note that, for DAGS_5, the dyadic order s needed to be amended too, and the rest of the code parameters modified accordingly to respect the requirements on code length, dimension etc. The case of DAGS_1 is a little peculiar. In fact, the theoretical complexity of the first attack approach can be made large enough by simply switching from $q = 2^5$ to $q = 2^6$, similarly to what was done for DAGS_3. With this in mind, and for the sake of simplicity, [1] featured this choice of parameters for DAGS_1, as reported in Table 3. However, thanks to the detailed analysis appeared in [3], it is now possible to see that these parameters are particularly vulnerable to the second attack approach. In what follows, we will briefly explain the reason for this, and present a new choice of parameters for DAGS_1.

As noted before, the success of the attack strongly depends on the dimension of the invariant code $\mathcal{D}$, which is given by $k_0 - c$, where $k_0 = k/s$ is the number of row blocks and $c = 2q/s$ was defined above. For the parameters in question, we have $k_0 = 26$ and $c = 8$ and therefore this dimension is 18. This leads to an imbalance in the ratio of number of equations to number of variables. The former are given by $(k_0-c)(n_0-k_0-1)$, where $n_0 = n/s$ is the number of column blocks, while the latter consists of the $(k_0 - c)c$ variables of the $\mathbf{U}$ type and the $n_0 - k_0 + c + \log s - 3$ variables of the $\mathbf{V}$ type that define the bilinear system. Therefore we obtain 450 equations in 179 total variables, and this ratio is about 2.5. The authors then show how the system can be solved by specializing the $\mathbf{U}$ variables to obtain linear equations, for a total cost of approximately 2^{111} operations, which is below the claimed security level. Actually, this cost can be further reduced following a hybrid approach that combines exhaustive search and Gröbner bases, to a total of 2^{83}.

The crucial point is that a ratio of 2.5 is quite high, and this is what makes the attack feasible. In contrast, the updated DAGS_5 parameters produce a ratio of 1.1 which is too low (the system has too many variables) while the situation for DAGS_3 is even more extreme, since in this case $c = k_0$ and therefore $\mathcal{D}$ does not even exist. In this case, the authors suggest to use the dual code instead, therefore replacing k_0 with $n_0 - k_0$ in all the above formulas. In principle, this makes the attack applicable, but the parameters yield a ratio of 0.7 which is again too low to be of any use. We insist on this crucial point to select our next choice of parameters for DAGS_1 (where "N.A." stands for "not applicable").

Table 5: New DAGS Parameters.

Security Level	q	m	n	k	s	t	w	BC 1	BC 2
1	2^8	2	704	352	2^4	11	88	$\approx 2^{542}$	N.A.
3	2^8	2	1216	512	2^5	11	176	$\approx 2^{288}$	N.A.
5	2^8	2	1600	896	2^5	11	176	$\approx 2^{289}$	N.A.

Table 6: New Memory Requirements (bytes).

Parameter Set	Public Key	Private Key	Ciphertext
DAGS_1	7744	2816	736
DAGS_3	11264	4864	1248
DAGS_5	19712	6400	1632

Note that we have only changed the parameters for DAGS_1, but we have chosen to report the other two sets too, in order to provide a complete view. With this new choice, we have $k_0 = 22$ and $c = 32$ and therefore $\mathcal{D}$ does not exist; in fact, not even its dual exists, since in this case $k_0 = n_0 - k_0$. This completely defeats the second attack approach, while the first approach would produce a ridiculously large complexity ($\approx 2^{542}$, see above), and we therefore feel comfortable claiming that DAGS_1 is now safe against all known attacks.

In the end, we can add the Barelli-Couvreur attack(s) to the amount of constraints on the selection of parameters, and we are very thankful to the authors of [4] and [3] for the detailed and careful analysis of the attack techniques. We will provide a complete overview of such constraints in the next section, where we will also detail an entirely new take on the subject.

4.2 Binary DAGS

Parameters in schemes based on QD-GS codes are a carefully balanced machine, needing to satisfy many constraints. First of all, we would like the code dimension $k = n - mst$ to be approximately $n/2$, since having a code rate close to $1/2$ is an optimal choice in many aspects (for instance, against ISD). In second place, in order to stay clear of the Barelli-Couvreur attack, the value q has to be sufficiently large, while the dyadic order s cannot be too big, as we just explained. On the other hand, s should still be large enough to obtain a significant reduction in key size. Balancing these two parameters can be quite hard since both q and s are required to be powers of 2. Meanwhile, the values of the extension degree m and the number of blocks t need to be, jointly, sufficiently large to reach the threshold of $mt > 21$, which is necessary to avoid the FOPT attack. Of course, m, s and t cannot all be large at the same time otherwise the code dimension k would become trivial. Moreover, it is possible to observe that the best outcome is obtained when m and t are of opposite magnitude (one big and one small) rather than both of "medium" size. Now, since s and t are also what determines the number of correctable errors (which is $st/2$), the value of t cannot be too small either, while a small m is helpful to avoid having to work on very large extension fields. Note that q^m still needs to be at least as big as the code length n (since the support elements are required to be distinct). After all these considerations, the result is that, in previous literature [19,6,1], the choice of parameters was oriented towards selecting a large base field q and the smallest possible value for the extension, i.e. $m = 2$, with s ranging from 2^4 to 2^6, and t chosen accordingly. We now investigate the consequences of choosing parameters in the opposite way.

Choosing large m and small t allows q to be reduced to the minimum, and more precisely q could be even 2 itself, meaning binary codes are obtained. Binary codes were already considered in the original QD Goppa proposal by Misoczki and Barreto [16], where they ended up being the only safe choice. The reason for this is that larger base fields mean m can be chosen smaller (and in fact, must, in order to avoid working on prohibitively large extension fields). This in turn means FOPT is very effective (remember that there is no parameter t for Goppa codes), so in order to guarantee security one had to choose m as big as possible (at least 16) and consequently $q = 2$. Now in our case, if t is small, s must be bigger (for error-correction purposes), and this pushes n and k up accordingly. We present below our binary parameters (Table 7) and corresponding memory requirements (Table 8).

Table 7: Binary DAGS Parameters.

Security Level	q	m	n	k	s	t	w	BC
1	2	13	6400	3072	2^7	2	128	N.A.
3	2	14	11520	4352	2^8	2	256	N.A.
5	2	14	14080	6912	2^8	2	256	N.A.

Table 8: Memory Requirements for Binary DAGS (bytes).

Parameter Set	Public Key	Private Key	Ciphertext
DAGS_1	9984	20800	832
DAGS_3	15232	40320	1472
DAGS_5	24192	49280	1792

The parameters are chosen to stay well clear of all the known algebraic attacks. In particular, using binary parameters should entirely prevent the latest attack by Barelli and Couvreur. In this case, in fact, we have $m >> 2$, and it is not yet clear whether the attack is applicable in the first place. However, even if this was the case, the complexity of the attack, which currently depends on the quantity q/s, would depend instead on mq^{m-1}/s. It is obvious that, with our choice of parameters, the attack would be completely infeasible in practice.
Note that, in order to be able to select binary parameters, it is necessary to choose longer codes (as explained above), which end up in slightly larger public keys: these are about 1.3 times those of the original (non-binary) DAGS. The private keys are also considerably larger. On the other hand, the binary base field should bring clear advantages in term of arithmetic, and result in a much more efficient implementation. All things considered, this variant should be seen as yet another tradeoff, in this case sacrificing key size in favor of increased security and efficient implementation.

5 Revised Implementation Results

In this section we present the results obtained in our revised implementation. Our efforts focused on several aspects of the code, with the ultimate goal of providing faster algorithms, but which are also clearer and more accessible. With this in mind, one of the main aspects that was modified is field multiplication. We removed the table-based multiplication to prevent an easy avenue for side-channel (cache) attacks: this is now vectorized by the compiler, which also allows for a considerable speedup. Moreover, we were able to obtain a considerable gain during key generation by exploiting the diagonal form of the Cauchy matrix. Finally, we "cleaned up" and polished our C code, to ensure it is easier to understand for external auditors. Below, we report timings obtained for our revised implementation (Table 10), as well as the measurements previously obtained for the reference code (Table 9), for ease of comparison. We remark that all these numbers refer to the latest DAGS parameters (i.e. those presented in Table 5);

an implementation of Binary DAGS is currently underway. The timings were acquired running the code 100 times and taking the average. We used CLANG compiler version 8.0.0 and the compilation flags -O3 -g3 -Wall -march=native -mtune=native -fomit-frame-pointer -ffast-math. The processor was an Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz.

Table 9: Timings for Reference Code.

Algorithm	Cycles		
	DAGS_1	DAGS_3	DAGS_5
Key Generation	2,540,311,986	4,320,206,006	7,371,897,084
Encapsulation	12,108,373	26,048,972	96,929,832
Decapsulation	215,710,551	463,849,016	1,150,831,538

Table 10: Timings for Revised Implementation.

Algorithm	Cycles		
	DAGS_1	DAGS_3	DAGS_5
Key Generation	408,342,881	1,560,879,328	2,061,117,168
Encapsulation	5,061,697	14,405,500	35,655,468
Decapsulation	192,083,862	392,435,142	388,316,593

As it is possible to observe, the performance is much faster than the previously reported numbers, despite the increase in parameters (and especially field size). Furthermore, we are planning to obtain even faster timings by incorporating techniques from [2]. These include a dedicated version of the Karatsuba multiplication algorithm (as detailed in [2]), as well as an application of LUP inversion to compute the systematic form of the public key in an efficient manner. Such an implementation is currently underway and results will appear in future work.

6 Conclusion

DAGS was one of the two NIST proposals based on structured algebraic codes, and the only one using Quasi-Dyadic codes (the other, BIG QUAKE, is based on Quasi-Cyclic codes). In fact, DAGS is also the only proposal exploiting the class of Generalized Srivastava codes. As such, we believe DAGS is already an

interesting candidate. At the present time, neither of these two proposals was selected to move forward to the second round of the standardization competition. Indeed, BIG QUAKE seemed to privilege security over performance, at the cost of selecting very conservative parameters, which gave way to large keys and ended up pushing the scheme into the same category of "conservative" schemes such as Classic McEliece (where a comparison favored the latter). The approach for DAGS was the opposite: we chose "aggressive" parameters, with the goal of reaching really interesting data sizes. In practice, this meant using $m = 2$ and this led to the Barelli-Couvreur attack. As a consequence, security concerns where raised and this led to the decision of not selecting DAGS for the second round.

In this paper, we investigated several aspects of DAGS, one of which is precisely its security. Exploiting the analysis given in [3], we have shown that two of the updated parameter sets, namely DAGS_3 and DAGS_5, are already beyond the scope of both attack approaches. The third set, that of DAGS_1, was unfortunate as the imbalance between number of equations and number of variables provided a way to instantiate the Gröbner basis technique effectively. Now that an analysis is available, it was easy to select a new parameter set for DAGS_1, which we trust will be the definitive one. We have then provided updated implementation figures, which take into account a variety of ideas for speeding up the code, such as vectorization, and dedicated techniques for Quasi-Dyadic codes. Moreover, we presented two variants offering some tradeoffs. The first is what we call "SimpleDAGS", and it is essentially a conversion of the original protocol to the Niederreiter framework. This allows for a cleaner protocol and a simpler security analysis (as in [5]), at the cost of increased data requirements. The second is a new set of binary parameters, which provides an advantage in terms of security against known structural attacks, again at the cost of a slight increase in data size. While an implementation is still underway for this set of parameters, we expect them to provide much faster times, in line with similar schemes such as Classic McEliece.

References

1. G. Banegas, P. S. Barreto, B. O. Boidje, P.-L. Cayrel, G. N. Dione, K. Gaj, C. T. Gueye, R. Haeussler, J. B. Klamti, O. N'diaye, , D. T. Nguyen, E. Persichetti, and J. E. Ricardini. DAGS: Key encapsulation using dyadic GS codes. *Journal of Mathematical Cryptology*, 2018.
2. G. Banegas, P. S. L. M. Barreto, E. Persichetti, and P. Santini. Designing efficient dyadic operations for cryptographic applications. *IACR Cryptology ePrint Archive*, 2018:650, 2018.
3. M. Bardet, M. Bertin, A. Couvreur, and A. Otmani. Practical algebraic attack on DAGS. *To appear*.
4. E. Barelli and A. Couvreur. An efficient structural attack on NIST submission DAGS. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 93–118. Springer, 2018.

5. D. J. Bernstein and E. Persichetti. Towards KEM unification. *IACR Cryptology ePrint Archive*, 2018:526, 2018.
6. P.-L. Cayrel, G. Hoffmann, and E. Persichetti. Efficient implementation of a CCA2-secure variant of McEliece using generalized Srivastava codes. In *Proceedings of PKC 2012, LNCS 7293, Springer-Verlag*, pages 138–155, 2012.
7. *https://classic.mceliece.org/*.
8. *http://www.dags-project.org*.
9. J.-C. Faugere, A. Otmani, L. Perret, F. De Portzamparc, and J.-P. Tillich. Structural cryptanalysis of McEliece schemes with compact keys. *DCC*, 79(1):87–112, 2016.
10. J.-C. Faugère, A. Otmani, L. Perret, and J.-P. Tillich. Algebraic cryptanalysis of McEliece variants with compact keys. In H. Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 279–298. Springer, 2010.
11. J.-C. Faugère, A. Otmani, L. Perret, and J.-P. Tillich. Algebraic Cryptanalysis of McEliece Variants with Compact Keys – Towards a Complexity Analysis. In *SCC '10: Proceedings of the 2nd International Conference on Symbolic Computation and Cryptography*, pages 45–55, RHUL, June 2010.
12. L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proc. 28th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 212–219, May 1996.
13. D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.
14. *https://keccak.team/kangarootwelve.html*.
15. F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*, volume 16. North-Holland Mathematical Library, 1977.
16. R. Misoczki and P. S. L. M. Barreto. Compact McEliece keys from Goppa codes. In *Selected Areas in Cryptography*, pages 376–392, 2009.
17. *https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization*.
18. R. Nojima, H. Imai, K. Kobara, and K. Morozov. Semantic security for the McEliece cryptosystem without random oracles. *Des. Codes Cryptography*, 49(1-3):289–305, 2008.
19. E. Persichetti. Compact McEliece keys based on quasi-dyadic Srivastava codes. *Journal of Mathematical Cryptology*, 6(2):149–169, 2012.
20. D. Sarwate. On the complexity of decoding Goppa codes. *IEEE Transactions on Information Theory*, 23(4):515 – 516, jul 1977.
21. P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.