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 [18], 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 [1], 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 [5] is one of the candidates to NIST's Post-Quantum Standardization call [1]. The submission introduces 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. 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, which offers an important alternative and a tradeoff between security and performance. Then, we discuss and propose new parameters, including an all-new set based on binary codes, to protect against both known and new attacks. Finally, we report the numbers obtained in a new, improved implementation which uses dedicated techniques and tricks to achieve a considerable speed up.

2 SimpleDAGS

The DAGS algorithms, as detailed in the original proposal submitted to the first Round [2], follow the "Randomized McEliece" paradigm of Nojima et al. [16], 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 [3] 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. In the description, we follow the same conventions used in the original DAGS specification, using variables n, k, r to denote, respectively, code length, dimension and co-dimension (as is standard in coding theory). All vectors are written in boldface, and, for ease of notation, treated as column vectors.

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 [2] 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 $(M \mid I_r) = A^{-1} H_{base}$: 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[8].

[8] 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 [14]).

1. Get syndrome $\boldsymbol{c}'_0$ corresponding to matrix[9] H' from private key[10].
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 follows the guidelines detailed by the "SimpleKEM" construction of [8], 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.

2.1 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* $\omega(z)/\sigma(z) = S(z) \pmod{z}^r$. Finally, finding the roots of the two polynomials reveals, respectivelly, the locations and values (if

[9] In alternant form.
[10] See below for details.

the code is not binary) of the errors. Actually, as shown in Section 6.3 of [5], 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 now explain how to perform alternant decoding when the input is a syndrome, rather than a noisy codeword, as in Algorithm 3 above. In this case, we don't 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.

2.2 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 [4]. 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 [8].

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.

3 Improved Resistance

It is natural to think that introducing additional algebraic structure like Quasi-Dyadicity in a scheme based on algebraic codes (such as Goppa or GS) would give an adversary more power to perform a structural attack. This is the case of the well-known FOPT attack [11], and successive variants [10], which exploit this algebraic structure to solve a multivariate system of equations and reconstruct an alternant matrix which is equivalent to the private key. Recently, Barelli and Couvreur presented a structural attack aimed precisely at DAGS [7], based on a new technique using so-called "norm-trace" codes. The attack performs very well against the original DAGS parameter sets, but it is overall not as critical as it appears. In fact, it is shown in Section 5.3 of [5] how this can be defeated, in most cases, by modifying a single parameter, namely the size of the base field q. Changing this from 2^5 to 2^6 and from 2^6 to 2^8 in, respectively, DAGS_1 and DAGS_3, is enough to push the attack complexity beyond the claimed security level. In the case of DAGS_5, the dyadic order s needs to be amended too, and the rest of the code parameters modified accordingly to respect the requirements on code length, dimension etc. This simple fix shows how the Barelli-Couvreur attack should probably be regarded as another constraint on the selection of parameters (similar to ISD, for example) rather than an attack per se. More

precisely, since the attack complexity depends strongly on the ratio q/s, parameters have to be chosen such that q is big enough and s is not too large (hence the details of the fix). For completeness, we report below the current selection of DAGS parameters (Table 1) and corresponding memory requirements (Table 2). The column labeled "BC" reports the estimated complexity of the Barelli-Couvreur attack.

Table 1: Updated DAGS Parameters.

Security Level	q	m	n	k	s	t	w	BC
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 2: 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

In what follows, we suggest another strategy for generating DAGS parameters which hasn't been explored before.

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 rate close to $1/2$ is an optimal choice in many aspect (for instance, against ISD). Secondly, the dyadic order s, which has to be a power of 2, should be as big as possible, to obtain the most reduction in key size (but not too big, for Barelli-Couvreur). On the other hand, the extension degree m and the number of blocks t need to be large enough to have $mt > 21$, in order to avoid FOPT. Of course, m, s and t can't all be large at the same time otherwise the 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 also determine the number of correctable errors, t can't 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 [17,9], the choice of parameters was oriented towards large base field q and small $m = 2$, 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 [15], 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 3) and corresponding memory requirements (Table 4).

Table 3: 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 4: 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 the algebraic attacks such as FOPT. 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, should depend instead of $mq^{m-1})/s$. 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. 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 public key size in favor of increased security and efficient implementation.

4 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: this is now vectorized by the compiler, allowing for a dramatic speedup (about 300%). Moreover, we incorporate a dedicated version of the Karatsuba multiplication algorithm (as detailed in [6]), applied to the quasi-dyadic case, which further boosts the efficiency of encapsulation (where all objects are quasi-dyadic). The integration of additional techniques from [6], such as LUP inversion, is currently in progress, and promises further speedups. 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 6), as well as the measurements previously obtained for the reference code (Table 5), for ease of comparison. We remark that all these numbers refer to the updated DAGS parameters (i.e. those presented in Table 3); an implementation of Binary DAGS is currently underway.

Table 5: 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 6: Timings for Revised Implementation.

Algorithm	Cycles		
	DAGS_1	DAGS_3	DAGS_5
Key Generation	3,840,144,290	5,397,811,954	8,512,178,387
Encapsulation	4,559,432	13,641,078	41,039,844
Decapsulation	27,304,282	387,945,056	397,973,400

References

1. *https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.*
2. *http://www.dags-project.org.*
3. *https://keccak.team/kangarootwelve.html.*
4. *https://classic.mceliece.org/.*
5. 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.
6. 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.
7. E. Barelli and A. Couvreur. An efficient structural attack on NIST submission DAGS. *arXiv preprint arXiv:1805.05429*, 2018.
8. D. J. Bernstein and E. Persichetti. Towards kem unification. *IACR Cryptology ePrint Archive*, 2018:526, 2018.
9. 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.
10. 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.
11. J.-C. Faugère, A. Otmani, L. Perret, and J.-P. Tillich. Algebraic cryptanalysis of McEliece variants with compact keys. In *EUROCRYPT*, pages 279–298, 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. F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*, volume 16. North-Holland Mathematical Library, 1977.
15. R. Misoczki and P. S. L. M. Barreto. Compact McEliece keys from Goppa codes. In *Selected Areas in Cryptography*, pages 376–392, 2009.
16. 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.
17. E. Persichetti. Compact McEliece keys based on quasi-dyadic Srivastava codes. *Journal of Mathematical Cryptology*, 6(2):149–169, 2012.
18. 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.