cryptography

MDPI

Article

On the Performance and Security of Multiplication in $GF(2^N)$

Jean-Luc Danger [1], Youssef El Housni [2,*], Adrien Facon [2,3], Cheikh T. Gueye [4], Sylvain Guilley [1,2,3], Sylvie Herbel [2], Ousmane Ndiaye [4], Edoardo Persichetti [5,*] and Alexander Schaub [1]

1 LTCI, Télécom ParisTech, Université Paris-Saclay, 75013 Paris, France; jean-luc.danger@TELECOM-ParisTech.fr (J.-L.D.); sylvain.guilley@secure-ic.com (S.G.); alexander.schaub@telecom-paristech.fr (A.S.)
2 Secure-IC S.A.S., 35510 Cesson-Sévigné, France; adrien.facon@secure-ic.com (A.F.); sylvie.herbel@secure-ic.com (S.H.)
3 Département d'Informatique, École Normale Supérieure, CNRS, PSL Research University, 75005 Paris, France
4 Département Mathématique et Informatique, Université Cheikh Anta Diop, Dakar 5005, Senegal; cheikht.gueye@ucad.edu.sn (C.T.G.); ouzdeville@gmail.com (O.N.)
5 Department of Mathematical Sciences, Florida Atlantic University, Boca Raton, FL 33431, USA
* Correspondence: youssef.housni21@gmail.com or youssef.elhousni@secure-ic.com (Y.E.H.); epersichetti@fau.edu (E.P.)

Received: 2 August 2018; Accepted: 13 September 2018; Published: 18 September 2018

Abstract: Multiplications in $GF(2^N)$ can be securely optimized for cryptographic applications when the integer N is small and does not match machine words (i.e., $N < 32$). In this paper, we present a set of optimizations applied to DAGS, a code-based post-quantum cryptographic algorithm and one of the submissions to the National Institute of Standards and Technology's (NIST) Post-Quantum Cryptography (PQC) standardization call.

Keywords: finite field arithmetic; tower fields; post-quantum cryptography; code-based cryptography; cache-timing attacks; secure implementation

1. Introduction

Arithmetic in $GF(2^N)$ is very attractive since addition is carry-less. This is why it is adopted in many cryptographic algorithms, which are thus efficient both in hardware (no carry means no long delays) and in software implementations.

In this article, we focus on software multiplication in $GF(2^N)$, and more specifically for small N. When N is smaller than a machine word size (that is, $N < 32$ or 64, on typical smartphones or desktops), all known window-based computational optimizations become irrelevant.

Our goal is to compute fast multiplications (since sums are trivially executed with the native XOR operation of computer instruction sets) that are secure with respect to cache-timing attacks. Therefore, we look for regular multiplication algorithms, that is, algorithms whose control flow does not depend on the data. Our method is not to come up with novel algorithms for multiplication, but to organize the computations in such a way that the resources of the computer are utilized optimally. Our contribution is thus to explore the way to load the machine in the most efficient way while remaining regular. We leverage the fact that regular algorithms can be executed SIMD (Single Instruction Multiple Data), hence they are natural candidates for bitslicing or similar types of parallel processing of packed operations. We compare these operations with those which are insecure and those which resort to special instructions (such as Intel Carry-Less MULtiplication, PCLMULQDQ). We conclude that the

most efficient implementations are SIMD, and hence benefit at the same time of performance and security, at no extra overhead. On top of that, we show that computations can be faster when mapping elements from tower fields $GF((2^\ell)^m)$ to isomorphic fields $GF(2^N)$, where $N = \ell m$. Previously, Paar [1] proposed an exhaustive research method to map elements of fixed isomorphic representations, and Sunar et al. [2] suggested a towering construction, from $GF(2^N)$ to $GF((2^\ell)^m)$. In this paper, we show a use-case where subsuming $GF((2^\ell)^m)$ into $GF(2^N)$ is beneficial to computation speed.

In the next section, we will show that computation in fields of characteristic two is a key building block for a wide array of cryptographic algorithms, and then cover some mathematical aspects related to computation in $GF(2^N)$ (where all computations are checked with SAGE). Our contribution is presented in Section 3. We also provide an application of our techniques to the DAGS Key Encapsulation Mechanism. DAGS was submitted to NIST [3] as one of the candidates for Post-Quantum Standardization, and so all the relative documentation (including reference code) [4]. Finally, we conclude in Section 5. The algorithms tested in this paper are given in C language in Appendix A.

2. The Field $GF(2^N)$ in Cryptography: Arithmetic and Suitability

2.1. Application to Block Ciphers

Block ciphers are the most important and most used symmetric cryptographic algorithms. Since 2001, the Advanced Encryption Standard (AES) [5] has become the most popular block cipher. AES is based on computations over the finite field $GF(2^N)$ with $N = 8$, which maps naturally to computer architectures. Representing bytes as elements of $GF(2^8)$ allows for expressing the confusion operation (named SubBytes) thanks to a multiplicative inverse in the finite field. This has the necessary properties to thwart differential and linear cryptanalyses, as well as attacks that use algebraic manipulations such as interpolation attacks.

2.2. Application to Classical Public-Key Cryptography

Elliptic curve cryptography can be executed efficiently on fields of characteristic two NIST standardizes Koblitz curves K-163, K-233, K-283, K-409, and K-571.

2.3. Application to Post-Quantum Public-Key Cryptography

As quantum systems start surfacing on the horizon, the importance of Post-Quantum Cryptography (PQC) is being recognized globally, to the point that NIST has launched a call for Post-Quantum Standardization [3]. Due to its inherent resistance to attacks by quantum computers, code-based cryptography is one of the main candidates for the task, alongside multivariate and lattice-based schemes. Although the original McEliece cryptosystem [6] is almost as old as RSA [7] and Diffie–Hellman [8], it has never been largely deployed, mainly due to large key sizes. There is thus an opportunity to revive and refine the area by developing more memory efficient primitives. However, multiple variants of the McEliece cryptosystem have been broken so far, as recalled by Bardet et al., in [9], and in practice the secure schemes are restricted to just a few families of codes, like Goppa and Generalized Srivastava codes. These schemes, by definition, need to handle elements in an extension of the base field $GF(2)$, and are thus of interest to us.

Among the candidates accepted for the first round [10] of the PQC competition, there are several schemes which perform operations in $GF(2^N)$ and fail to do so in a side-channel resistant way, at least in their reference implementation. The exhaustive list is given in Table 1.

Note that most fields $GF(2^N)$ have N smaller than the typical size of machine words.

The vulnerability to side-channel attacks mainly stems from two aspects. For small extensions (up to $N = 8$ or slightly higher), multiplication of elements in $GF(2^N)$ is implemented using log-antilog tables, using the fact that $a \times b = \log^{-1}(\log(a) + \log(b))$, for $a, b \neq 0$. The logarithm and antilogarithm of all elements in $GF(2^N)$ are tabulated, and a multiplication merely consists in three table accesses

(two in the log table, one in the antilog table, or vice versa). However, this creates a data-dependent table access, and the operands could potentially be recovered using standard side-channel attacks such as FLUSH+RELOAD [11]. For big extensions, computing these tables is too costly and the implementation of operations in $GF(2^N)$ is handled differently. However, the multiplications are executed by taking *data-dependent branches*; which branch is taken could be recovered via similar attack techniques, thereby revealing once again the operands of the multiplication.

Table 1. PQC submissions at NIST [3] using vulnerable $GF(2^N)$ operations.

Submission	Type	Finite Field	Tower Fields Used
BIG QUAKE	Code-based	$GF(2^N), N = 12, 18$	No
DAGS	Code-based	$GF((2^5)^2), GF((2^6)^2)$	Yes
EdonK	Code-based	$GF(2^N), N = 128, 192$	No
Ramstake	Code-based	$GF(2^8)$	No
RLCE	Code-based	$GF(2^N), N = 10, 11$	No
LAC	Lattice-based	$GF(2^N), N = 9, 10$	No
DME	Multivariate	$GF(2^N), N = 24, 48$	No
HIMQ-3	Multivariate	$GF(2^8)$	No
LUOV	Multivariate	$GF(2^8), GF((2^{16})^\ell), \ell = 3, 4, 5$	Yes

We checked for these kinds of potential leaks using a static analysis tool [12] specifically developed for tracking microarchitectural side-channels, including *data-dependent table accesses* and *data-dependent branches*. This tool requires the user to specify which variables are sensitive, such as the secret key or the randomness used during signature or encryption. It then performs a dependency analysis and determines whether any variable depending on sensitive values is used as an index for table access or as the condition variable of a branching operation (`If`, `While`, `Switch` or the stop condition in a `For` loop).

2.4. Arithmetic in Extensions of $GF(2)$

Let $GF(2^N)$ denote the extension field of order N defined over $GF(2)$. Let α be a primitive element of $GF(2^N)$. The set

$$\mathcal{B} = \{1, \alpha, \alpha^2, \ldots, \alpha^{N-1}\}$$

is a basis of $GF(2^N)$ over $GF(2)$, referred to as a polynomial basis. Thus, given an element $A \in GF(2^N)$, we can write

$$A = \sum_{i=0}^{N-1} a_i \alpha^i,$$

where the coefficients $a_0, a_1, \ldots, a_{N-1} \in GF(2) = \{0, 1\}$. Then, the field arithmetic can be derived using the chosen basis.

2.5. Tower Fields Representation

Depending on the choice of the basis $\mathcal{B}$, the elements of $GF(2^N)$ can be defined differently. If N is the product of two integers ℓ and m, then $GF(2^N)$ can be defined over $GF(2^\ell)$. In the rest of the paper, we call $GF((2^\ell)^m)$ a *composite field* and $GF(2^\ell)$ the *ground field*. Note that $GF((2^\ell)^m)$ and $GF(2^N)$ refer to the same field although their representation methods are different.

Given two representations of the finite field $GF(2^N)$, it is possible to map one to the other, thanks to a conversion matrix. The first representation is $GF(2^N)$ as an extension of $GF(2)$ and the second representation is $GF(2^N)$ as an extension of $GF(2^\ell)$ where $N = m\ell$ for $\ell, m \in \mathbb{N}$. Here, the elements of $GF(2^N)$ are polynomials whose coefficients are in $GF(2) = \{0, 1\}$ of degree at most $N - 1$ and the elements of $GF((2^\ell)^m)$ are polynomials whose coefficients are in $GF(2^\ell)$ of degree at most $m - 1$. Hence, we write in Kronecker style

$$GF(2^\ell) = GF(2)[\gamma]/P,$$
$$GF((2^\ell)^m) = GF(2^\ell)[\beta]/Q,$$
$$GF(2^N) = GF(2)[\alpha]/D,$$

where:

- $P(x) = x^\ell + \ldots + p_1 x + 1$ is an irreducible polynomial over $GF(2)$ of degree ℓ,
- $Q(x) = x^m + \ldots + q_1 x + 1$ is an irreducible polynomial over $GF(2^\ell)$ of degree m,
- $D(x) = x^N + \ldots + d_1 x + 1$ is an irreducible polynomial over $GF(2)$ of degree N,

and where γ, β and α are their respective roots. Thus, the elements of $GF(2^\ell)$, $GF((2^\ell)^m)$ and $GF(2^N)$ are the residue classes modulo of their respective irreducible polynomials. Such polynomials always exist [13]. In general, the number of irreducible polynomials of degree N with coefficients in $GF(q)$ is given by

$$L_q(N) = \frac{1}{N} \sum_{d|N} q^d \mu\Big(\frac{N}{d}\Big),$$

where $\mu(k)$ is the Möbius function defined by

$$\mu(k) = \begin{cases} 0, & \text{if } k \text{ has one or more repeated prime factors,} \\ 1, & \text{if } k = 1, \\ (-1)^n & \text{if } k \text{ is a product of } n \text{ distinct primes.} \end{cases}$$

For instance, the number of irreducible polynomials of degree 12 in $GF(2^{12})$ (field used in DAGS) is

$$L_2(12) = \frac{1}{12} \sum_{d|12} 2^d \mu(\frac{12}{d}) = 335$$

and thus multiple representations can be derived for the same element in the field.

2.6. Composite Fields and Fields Mapping

Since $GF((2^\ell)^m)$ and $GF(2^N)$ refer to the same field, they are isomorphic [13]. However, although two fields' representations are isomorphic, the algorithmic complexity of their field operations may differ, depending on the polynomials Q and D. A binary $N \times N$ matrix T can be derived to map elements of $GF(2^N)$ to elements of $GF((2^\ell)^m)$. The inverse of T, denoted T^{-1}, will perform the mapping the other way around. The conversion problem was addressed by Paar in [1]. In this work, conversion matrices are derived from $GF(2^N)$ and $GF((2^\ell)^m)$ that are already fixed by their generating polynomials. The construction is based on finding a relation between the primitive elements γ and α such that

- $\alpha^r = \gamma$ is known for some integer r and;
- $D(\alpha^r) \equiv 0 \pmod{P, Q}$.

Since there is no established mathematical connection between α and γ an exhaustive search is needed. In a related work [2], Sunar et al. redefined the problem. Instead of finding a conversion matrix, the paper proposes to construct the composite field given the field of characteristic two. Here, we recall the results and examine the problem in a slightly different way: our aim is to find a suitable isomorphic representation to construct the field of characteristic two given the ground and extension fields.

Theorem 1. *For $\beta \in GF(2^{m\ell})$ and $\gamma = \beta^r$ where $r = \frac{2^{m\ell}-1}{2^\ell-1}$,*

1. $\beta^r \in GF(2^\ell)$,
2. *if β is a primitive element, then γ is primitive in $GF(2^\ell)$.*

Proof. Chapter 2, [13]. □

Let $GF(2^\ell) = GF(2)[\gamma]/P$ be the ground field. The extension field $GF((2^\ell)^m)$ can be constructed using the polynomial

$$\begin{aligned} S_\beta(x) &= (x+\beta)(x+\beta^{2^\ell})(x+\beta^{2^{2\ell}})\dots(x+\beta^{2^{\ell^{m-1}}}) \\ &= s_m x^m + \dots + s_1 x + s_0. \end{aligned}$$

Noting that $s_m = 1$ and using Vieta's formulas, we obtain

$$\begin{cases} s_{m-1} & = -\beta - \beta^{2^\ell} - \beta^{2^{2\ell}} - \dots - \beta^{2^{\ell^{m-1}}} \\ s_{m-2} & = (\beta\beta^{2^\ell} + \beta\beta^{2^{2\ell}} + \dots + \beta\beta^{2^{(m-1)\ell}}) + \\ & \quad + (\beta^{2^\ell}\beta^{2^{2\ell}} + \beta^{2^\ell}\beta^{2^{3\ell}} + \dots + \beta^{2^\ell}\beta^{2^{(m-1)\ell}}) + \dots + \beta^{2^{(m-2)\ell}}\beta^{2^{(m-1)\ell}} \\ & \quad \vdots \\ s_0 & = (-1)^m \beta\beta^{2^\ell}\beta^{2^{2\ell}}\dots\beta^{2^{\ell^{m-1}}} \end{cases}$$

or, equivalently, the $(m-k)$-th coefficient s_{m-k} is related to a signed sum of all possible subproducts of roots, taken k-at-a-time:

$$s_{m-k} = (-1)^k \sum_{1 \le i_1 \le i_2 \le \dots \le i_k \le m} \beta_{i_1}\beta_{i_2}^{2^\ell}\beta_{i_3}^{2^{2\ell}}\dots\beta_{i_m}^{2^{\ell^{m-1}}}.$$

Thus, given a ground field $GF(2^\ell)$ and its extension field $GF((2^\ell)^m)$, we are looking for a field $GF(2^N)$ with primitive element α such that:

- $P(\alpha^r) = 0$ where $r = (2^{m\ell}-1)/(2^\ell - 1)$, hence $\alpha^r = \gamma$,
- $Q(x) = S_\alpha(x)$ hence, $q_{m-k} = (-1)^k \sum_{1 \le i_1 \le i_2 \le \dots \le i_k \le m} \alpha_{i_1}\alpha_{i_2}^{2^\ell}\alpha_{i_3}^{2^{2\ell}}\dots\alpha_{i_m}^{2^{\ell^{m-1}}}$.

Once we find the suitable representation of $GF(2^N)$, we derive the conversion matrix as follows. Any element A in $GF(2^N)$ has two representations:

$$A = \sum_{i=0}^{N-1} a_i\alpha^i = \sum_{j=0}^{m-1} a'_j\beta^j, \quad a_i \in GF(2),\ a'_j \in GF(2^\ell).$$

We showed that our construction allows β^r to be primitive in $GF(2^\ell)$. Thus, $\{1, \beta^r, \beta^{2r}, \dots, \beta^{(m-1)r}\}$ is a basis in $GF(2^\ell)$ and we can write

$$a'_j = \sum_{i=0}^{\ell-1} a_{ji}\beta^{ri}.$$

Thus, $A = \sum_{j=0}^{m-1} a'_j\beta^j = \sum_{j=0}^{m-1}\sum_{i=0}^{\ell-1} a_{ji}\beta^{ri}\beta^j = \sum_{j=0}^{m-1}\sum_{i=0}^{\ell-1} a_{ji}\beta^{ri+j}$.

Then, the terms β^{ri+j} are reduced using the generating polynomial $P(x)$

$$\beta^{ri+j} = \sum_{f=0}^{\ell-1} t_f\beta^f,\ t_f \in GF(2).$$

These are the coefficients of the conversion matrix. In the end, we will have

$$A = \sum_{j=0}^{m-1}\sum_{i=0}^{\ell-1}\sum_{f=0}^{N-1} a_{ji}t_f\beta^f. \tag{1}$$

The $N \times N$ conversion matrix T with coefficients in $GF(2)$ is obtained from Equation. (1):

$$\begin{bmatrix} a_0 \\ \vdots \\ a_{m\ell-1} \end{bmatrix} = T \begin{bmatrix} a_{00} \\ \vdots \\ a_{(m-1)(\ell-1)\cdot} \end{bmatrix}$$

The conversion matrix from the field to the composite field is then T^{-1}.

Example 1. *Let the ground and extension fields be*

$$GF(2^6) = GF(2)[\gamma]/\langle \gamma^6 + \gamma + 1 \rangle,$$
$$GF((2^6)^2) = GF(2^6)[\beta]/\langle \beta^2 + \gamma^{34}\beta + \gamma \rangle.$$

The field $GF(2^{12}) = GF(2)[\alpha]/P$ *with* $(\alpha^{65})^6 + \alpha^{65} + 1 = 0$ *and* $\alpha^{65} = \gamma$ *and* $\alpha^{64} + \alpha = \gamma^{34}$ *is* $GF(2)[\alpha]/\langle \alpha^{12} + \alpha^{11} + \alpha^8 + \alpha^6 + 1 \rangle$. *Now, we show the construction of the conversion matrices. An element* A *in* $GF((2^6)^2)$ *is expressed as*

$$A = a_0 + a_1\alpha,$$

where $a_j \in GF(2^6)$. *We can express* a_j *in* $GF(2^6)$ *using* $\gamma = \alpha^{65}$ *as the basis element*

$$\begin{aligned} a_j &= a_{j0} + a_{j1}\gamma + a_{j2}\gamma^2 + a_{j3}\gamma^3 + a_{j4}\gamma^4 + a_{j5}\gamma^5 \\ &= a_{j0} + a_{j1}\alpha^{65} + a_{j2}\alpha^{130} + a_{j3}\alpha^{195} + a_{j4}\alpha^{260} + a_{j5}\alpha^{325}, \end{aligned}$$

where $a_{ji} \in GF(2)$ *for* $j \in \{0,1\}$ *and* $i \in \{0,1,2,3,4,5\}$. *Thus, the representation of* A *in the composite field is*

$$\begin{aligned} A = a_{00} + a_{01}\alpha^{65} + a_{02}\alpha^{130} + a_{03}\alpha^{195} + a_{04}\alpha^{260} + a_{05}\alpha^{325} + \\ + a_{10}\alpha + a_{11}\alpha^{66} + a_{12}\alpha^{131} + a_{13}\alpha^{196} + a_{14}\alpha^{261} + a_{15}\alpha^{326}. \end{aligned} \quad (2)$$

The next step is to reduce the terms α^{65i+j} *for* $j = \{0,1\}$ *and* $i = \{0,1,2,3,4,5\}$ *using the generating polynomial* $p(x)$. *This will give terms of Equation* (2) *with* α *exponent between* 0 *and* 11. *The reduction modulo* $p(x)$ *is done by using successively the relation* $\alpha^{12} = \alpha^{11} + \alpha^8 + \alpha^6 + 1$. *We then obtain the representation of* A *in the field* $GF(2^{12})$ *using the basis* $\{1, \alpha, \alpha^2, \ldots, \alpha^{11}\}$ *as*

$$A = b_0 + b_1\alpha + b_2\alpha^2 + \ldots + b_{11}\alpha^{11}.$$

The entries of the 12×12 *matrix* T *are determined by the relationship between the term* b_h *for* $h = 1, 2, \ldots, 11$ *and* a_{ji} *for* $j = \{0,1\}$ *and* $i = \{0,1,2,3,4,5\}$. *Gathering all the terms of* $\{1, \alpha, \alpha^2, \ldots, \alpha^{11}\}$, *we obtain*

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad (3)$$

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}. \quad (4)$$

The matrix T gives the representation of an element in the field $GF(2^{12})$ given its representation in the composite field $GF((2^6)^2)$. The inverse transformation, i.e., the conversion from $GF(2^{12})$ to $GF((2^6)^2)$, requires the computation of T^{-1}.

3. Results and Discussion

We start by presenting state-of-the-art multiplication algorithms in $GF(2^N)$ for small values of N, i.e., when N is smaller than the machine word. Those algorithms are insecure. Then, we present secure variants with respect to cache timing attacks.

3.1. Multiplication in $GF(2^N)$

Fast implementation techniques for $GF(2^N)$ arithmetic have been studied intensively by researchers. Among various arithmetic operations, $GF(2^N)$ multiplications gained the most attention.

For small N, the multiplication is usually carried out using look-up tables, called log-antilog tables. Algorithms of tables initialization and the derived tabulated multiplication are given in Algorithm 1 and Algorithm 2 and in Algorithms A1 and A2 in Appendix A as C code. This method (Algorithm A3 in Appendix A) presents a testing vulnerability. Indeed, 0 is always mapped to -1 and never used; furthermore, this method is vulnerable anyways to cache-timing attacks such as PRIME+PROBE [14], and FLUSH+RELOAD [11] that targets the Last-Level-Cache. In a cryptographic scheme where critical operations such as key generation or encryption use log-antilog multiplication over $GF(2^N)$, the difference in memory access time to lookup tables caused by the cache may leak information about the secret key. These attacks first completely fill the cache with the attacker's data. The critical operation is run, and, as it is running, the parts of tables that it uses are loaded from main memory into the cache. Since the cache is full of attacker's data, some of it will have to be evicted to make place. Once the operation is done, the attacker analyses which parts of his data have been evicted and this tells him which table indexes were used, leaking information about the secret key.

Algorithm 1 Initialization of the antilog table

Require: The finite field $GF(2^n)$ and its generator polynomial P.
Ensure: The antilog table.

```
antilog[0]= 0
for i in range(1, 2^n) do
    antilog[i]=antilog[i − 1]<< 1                ▷ Shift to the left
    if antilog[i − 1]= 2^(n−1) then
        antilog[i]=antilog[i]⊕P                  ▷ XOR with the generator polynomial
antilog[2^n − 1]= 1
```

Algorithm 2 Initialization of the log table

Require: the antilog table.
Ensure: the log table.
1: log[0]$= -1$
2: log[1]$= 0$
3: **for** i in range$(1, 2^n)$ **do**
4: log[antilog[i]]$= i$

The implementation of the tabulated method may be sometimes too costly for large N and the multiplication is then handled differently. One may use tower field arithmetic and store lookup tables for $GF(2^\ell)$ where ℓ is taken small such as $\ell \mid N$ (Algorithm A4 in Appendix A). Tower field arithmetic is slow, but we can perform conversions both ways with respect to the theory presented in Section 2 (Algorithm 3).

Algorithm 3 Multiplication in the tower field in $GF((2^6)^2)$

Require: two polynomials $x = \{x_i\}, y = \{y_i\}$ and an extension polynomial p of order 2
Ensure: polynomial $r = x.y = \{r_i\}$
1: $a_1 = x >> 6$
2: $a_2 = y >> 6$
3: $b_1 = x\&63$
4: $b_2 = y\&63$
5: $tmp_1 = \text{mult}(a_1, a_2)$
6: $a_3 = \text{mult}(tmp_1, p_1) \oplus \text{mult}(a_1, b_2) \oplus \text{mult}(b_1, a_2)$ ▷ p_1 the coefficient of x in p
7: $b_3 = \text{mult}(tmp_1, p_0) \oplus \text{mult}(b_1, b_2)$ ▷ p_0 the constant term in p
8: $r = (a_3 << 6) \oplus b_3$

A straightforward alternative method is iterative multiplication. This is done performing polynomial multiplication and conditional reduction modulo the generator polynomial (Algorithm 4 and Algorithm A5 in Appendix A as C code). Iterative methods cannot be executed without taking data-dependent branches and thus are vulnerable to branch prediction analysis [15,16]. In fact, the information leakage is based on the timing differences produced by the branch prediction unit (BPU).

Algorithm 4 Iterative multiplication with conditional reduction

Require: Two polys $X = \{x_i\}, Y = \{y_i\}$ of orders at most n and a reduction polynomial P of order n
Ensure: Polynomial $R = X.Y = \{r_i\}$ of order n
1: $R \leftarrow 0$
2: **for** i in range(n) **do**
3: **if** $y_i = 1$ **then**
4: $r_i = r_i + x_i$
5: **if** order$(R) > n$ **then**
6: reduce R by P ▷ polynomial division

3.2. Secure Computation in $GF(2^N)$

There are many countermeasures to prevent cache-timing attacks, but they may affect the computation performance. Here, we propose a trade-off between secure and fast $GF(2^N)$ computation. One countermeasure is the constant time implementation where the execution time does not depend

on the secret key or input data. In case of tabulated multiplication, this cannot be achieved, but, in case of iterative methods, we replace conditional reduction by unconditional reduction. This means that the reduction modulo the generator polynomial is performed at each iteration and therefore no timing information is leaked (Algorithm 5 and Algorithm A6 in Appendix A as C code).

Algorithm 5 Iterative multiplication with unconditional reduction

Require: Two polynomials $X = \{x_i\}$, $Y = \{y_i\}$ of orders at most n and a reduction polynomial P of order n

Ensure: Polynomial $R = X.Y = \{r_i\}$ of order n

1: $R \leftarrow 0$
2: **for** i in range(n) **do**
3: $m \leftarrow -y_i$ ▷ the mask
4: $r_i = r_i + x_m$
5: $m \leftarrow -x_n$ ▷ the mask
6: reduce R by P ▷ polynomial division

Basically, this code adds the term $x_i X^i$ if y_i is one (where x_i and y_i are the polynomials to multiply). Note that, because of the two's complement representation, -0 is 0000000000000000 over 16 bits and -1 is 1111111111111111. We can, thus, use $-y_i$ as a mask in the first branch. We follow the same idea in the second branch shifting the condition by ℓ. Another constant time countermeasure is the bitsliced implementation [17,18] that uses SIMD architecture to perform the same operation on multiple data points simultaneously. In fact, we convert 64 N-bit words into N 64-bit words and multiply the 64-bit words in a single generation (Algorithm 6 and Algorithm A7 in Appendix A as C code).

Algorithm 6 Bitsliced multiplication

Require: 2×64 n-bit words X_i and Y_i where $i \in [1, 64]$

Ensure: 64 n-bit words $R_i = X_i.Y_i$

1: Transpose $X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{64} \end{bmatrix}$ to $X' = \begin{bmatrix} X'_1 \\ X'_2 \\ \vdots \\ X'_n \end{bmatrix}$ where X'_j are 64-bit words for $j \in [1, n]$

2: Transpose $Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_{64} \end{bmatrix}$ to $Y' = \begin{bmatrix} Y'_1 \\ Y'_2 \\ \vdots \\ Y'_n \end{bmatrix}$ where Y'_j are 64-bit words for $j \in [1, n]$.

3: Get R' where $R'_j = X'_j.Y'_j$ for $j \in [1, n]$

4: Transpose $R' = \begin{bmatrix} R'_1 \\ R'_2 \\ \vdots \\ R'_n \end{bmatrix}$ to $R = \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_n \end{bmatrix}$ where R_i are n-bit words for $i \in [1, 64]$

This countermeasure prevents information leakage and speeds up the implementation. To compensate for the loss of performance, one can use Intel's CLMUL (Carry-Less Multiplication) assembly instruction set to improve the speed of multiplication over $GF(2^N)$. PCLMULQDQ, available with the new 2010 Intel Core processor family based on the 32 nm Intel microarchitecture codename Westmere, performs carry-less multiplication of two 64-bit operands over a finite field (without reduction). The optimizations take advantage of the processor datapath (64 bit) and of available instructions. It is thus artificial and probably not very informative to write multiplications as algorithms. Instead, we provide the extensive C code for the case $N = 12$ ($\ell = 6$ and $m = 2$), written in a portable way (ANSI POSIX). The code is abundantly commented on, hence the operations carried out should not leave place to ambiguity. Regarding performance evaluation of the different functionally equivalent codes, again, a pure algorithmic description would be misleading. For instance, 64 XOR operations can be conducted in one clock cycle provided the operands are laid out as the 64 bits of a quad word, or otherwise in 64 clock cycles if the operands are located at different addresses. Therefore, performance reads better from the C code. We estimated the performance by averaging the execution time of each multiplication placed in a loop. This method allows to filter out abnormal durations caused by improper pipeline or cache initializations.

In the next section, we show a case study where we compare different implementations of $GF(2^N)$ multiplication on the basis of security and performance. The comparison further points out computation performance over the tower field $GF((2^\ell)^m)$ and the field $GF(2^N)$, where $N = m\ell$, using the derived conversion matrices in Section 2.

4. Case Study: Optimization of DAGS

DAGS is a code-based Key Encapsulation Mechanism (KEM). As all code-based primitives, it relies on the Syndrome Decoding Problem [19], and shows no vulnerabilities against quantum attacks. In particular, DAGS is based on the McEliece cryptosystem [6] and uses Quasi-Dyadic Generalized Srivastava (GS) codes to address the issue of the large public key size, which is inherent to code-based cryptography. We start by recalling some important definitions.

Definition 1. *For $m, n, s, t \in \mathbb{N}$ and a prime power q, let $\alpha_1, \ldots, \alpha_n, w_1, \ldots, w_s$ be $n + s$ distinct elements of $GF(q^m)$ and $z_1, \ldots, z_n$ be nonzero elements of $GF(q^m)$. 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 defined as

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

Parameters are the length $n \leq q^m - s$, dimension $k \geq n - mst$ and minimum distance $d \geq st + 1$.

Definition 2. *Given a ring $\mathcal{R}$ (in our case $\mathbb{F}_{q^m}$) and a vector $\bar{h} = (h_0, \ldots, h_{n-1}) \in \mathcal{R}^n$, the dyadic matrix $\Delta(\bar{h}) \in \mathcal{R}^{n \times n}$ is the symmetric matrix with components $\Delta_{ij} = h_{i \oplus j}$, where $\oplus$ stands for bitwise exclusive-or on the binary representations of the indices. The sequence $\bar{h}$ is called its signature. 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 (and where any* 1×1 *matrix is dyadic).*

A linear code is *quasi-dyadic* if it admits a parity-check in quasi-dyadic form, i.e., a block matrix whose component blocks are dyadic matrices.

It has been shown by Misoczki and Barreto [20] that it is possible to build Goppa codes in quasi-dyadic form if the code is defined over a field of characteristic 2, and the dyadic signature satisfies the fundamental equation

$$\frac{1}{h_{i \oplus j}} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0}. \tag{5}$$

Persichetti in [21] then showed how to adapt the Misoczki–Barreto algorithm to generate quasi-dyadic GS codes. Intuitively, using this larger class of codes (of which Goppa codes are a subclass) provides more flexibility in the design of cryptographic schemes. More importantly, thanks to their "layered" structure, GS codes make it easier to resist structural attacks aimed at recovering the private key. In particular, the parameter t plays a crucial role in defining the complexity of one such attack (and successive variants), due to Faugère, Otmani, Perret and Tillich, and simply known as FOPT [22].

The attack, succinctly, consists of generating a system of equations from the fundamental relationship between generator and parity-check matrices $G \cdot H^T = 0$. The system of equations is heavily simplified thanks to the particular relations stemming from the quasi-dyadic form (and the limitations inherent to alternant codes) and successively solved using Gröbner bases. This allows for recovering an equivalent matrix for decoding, i.e., a private key. Despite the lack of a definitive complexity analysis, it is possible to observe that the attack scales somewhat proportionally to the value that defines the solution space (i.e., number of free variables) of the system of equations. In the case of quasi-dyadic Goppa codes, this is given simply by $m - 1$; thus, the key factor is that the value m be large enough to make the attack unfeasible. In the original proposal by Misoczki and Barreto, this value varies from 2 to 16, where the large extension field $GF(2^N)$ is kept constant (i.e., $N = 16$) and the base field varies, (i.e., $\ell = 1, 2, 4, 8$). As a consequence, the dimension of the solution space is in most cases trivial or so, and the only parameters that weren't broken in practice were those corresponding to $m = N = 16$, although the attack authors recommend $m - 1$ to be at least 20.

In the case of GS codes, it is possible to apply the attack, but the dimension of the solution space is given by $mt - 1$ instead. It is thus a lot easier to achieve larger values for this, while at the same time keeping the extension field small (but still large enough to define long codes) and consequently having more efficient arithmetic. It follows that schemes based on GS codes (and DAGS is no exception) are usually defined over a relatively large base field, with the goal of minimizing the value m; the "burden" of thwarting FOPT falls then on t, which is chosen as relatively large. This has the additional advantage of a better error-correction capacity, since the number of correctable errors depends on t (it is in fact $st/2$). Better error-correction means that generic decoding attacks like Information-Set Decoding (ISD) [23,24] are harder, and thus implies the possibility of better parameters.

4.1. Initial Choice of Parameters

We report DAGS parameters below Table 2. Note that in all cases the condition $mt \geq 21$ is satisfied to prevent the FOPT attack, as suggested by the authors themselves.

Table 2. DAGS [4] parameters.

Name	Security Level	q	m	n	k	s	t	Public Key Size
DAGS_1	128	2^5	2	832	416	2^4	13	6760
DAGS_3	256	2^6	2	1216	512	2^5	11	8448
DAGS_5	512	2^6	2	2112	704	2^6	11	11,616

For DAGS 3 and DAGS 5, the finite field $GF(2^6)$ is built using the polynomial $x^6 + x + 1$ and then extended to $GF(2^{12})$ using the quadratic irreducible polynomial $x^2 + \alpha^{34}x + \alpha$, where α is a primitive element of $GF(2^6)$.

4.2. Improved Field Selection

The protocol specification for DAGS 3 and DAGS 5 is the same; hence, we choose to focus on DAGS 5 optimization in this section. The overall process is

- Key Generation,
- Encapsulation,
- Decapsulation.

Multiplications over the finite fields $GF(2^6)$ and $GF((2^6)^2)$ are carried all along the process and must be protected especially in critical operations such as key generation and encapsulation. The key generation is performed over the tower field $GF((2^6)^2)$ using the log-antilog tables of $GF(2^6)$. Thus, it must be protected against cache-timing attack on one hand and optimized for fast implementation on the other. Encapsulation is a critical operation performed over $GF(2^6)$ using the tabulated method and hence is vulnerable to cache-leakage. In the following, we propose comparing the performance of seven implementations of multiplication algorithms over $GF(2^6)$, $GF((2^6)^2)$ and $GF(2^{12})$. In fact, according to Section 2, we can convert elements from $GF((2^6)^2)$ to $GF(2^{12})$ to perform multiplication in the key generation process faster. In the example of Section 2, we used DAGS polynomials $x^6 + x + 1$ for $GF(2^6)$ and $x^2 + \alpha^{34}x + \alpha$ for $GF((2^6)^2)$ with α a primitive in $GF(2^6)$ and hence we can use the derived matrix T for the isomorphic mapping. Note that we can change the tower field polynomial, yet still be consistent with DAGS design, in order to construct a field $GF(2^{12})$ with a sparse generator polynomial. That is to say, using the polynomial $x^2 + \alpha^{27}x + \alpha$ for the extension yields a mapping to $GF(2^{12})$ with the generator trinomial $x^{12} + x^7 + 1$. However, the overall gain when compared to the performances using the pentanomial from the example is negligible, not to mention the cost of changing the initial polynomial in the reference code. Thus, we chose to keep the initial parameters and compare the following seven implementations on the basis of security and performance:

- Tabulated log/antilog (Algorithms A1–A3),
- Iterative, conditional reduction (Algorithm A5),
- Iterative, ASM with PCLMUL, conditional reduction (Algorithm A5),
- Iterative, unconditional reduction (Algorithm A6),
- Iterative, ASM with PCLMUL, unconditional reduction (Algorithm A6),
- Iterative, unconditional reduction, 1-bit-sliced, 64 comput. in parallel (Algorithm A7),
- Iterative, ASM with PCLMUL, unconditional reduction, bit-sliced 2 computs. In parallel (Algorithm A8).

4.3. Implementation Performances

We will present the results of our experiments below.

- (*): Conversion from $GF((2^6)^2)$ to $GF(2^{12})$ using T in Example (1) is 112 cycles, using POPCNT ASM instruction is 38 cycles (Algorithm A9).
- (**): Time to initialize the tables: (Algorithm A1 and A2);
  - 2360 cycles on $GF(2^6)$,
  - 267,086 cycles on $GF((2^6)^2)$ and,
  - 7884 cycles on $GF(2^{12})$,

  (can be precomputed, hence cycles=0)
- (***): Transposition (Algorithm A7.1) time is;

  - 780 cycles on $GF(2^6)$ and,
  - 1613 cycles on $GF(2^{12})$,
- (****): Transposition $(X, X') \rightarrow X'2^{2N} + X$ is 2 cycles on $GF(2^N)$.

In Table 3, we have presented the performances of different implementations of multiplication over the finite field $GF(2^6)$. We further compared these implementations over the tower field $GF((2^6)^2)$ and the isomorphic field $GF(2^{12})$. Note that the costs of isomorphic mapping, bitslice transposition and log-antilog tables initialization are excluded from Table 3 since they can be carried only a few times during the process and not at each multiplication. Accordingly, we conclude the following:

- The tabulated log-antilog version is the fastest amongst non-parallel algorithms.
- It is faster to implement tower field computation directly in an isomorphic field of characteristic two.
- The modular multiplication with Carry-less MULtiplication (PCLMUL) dedicated Assembly (ASM) instruction does not improve the speed since the overhead in the function call is dominating the computation for those small values of N. However, in case of only one serial operation, PCLMUL should be used because it has the lowest latency.
- Constant-time cache secure implementations take more time than those that are not secure. Moreover, we noticed that "conditional reduction" in C code is actually constant-time once compiled in assembly code (when optimization flag is set) owing to the use by the compiler of the CMOV (conditional move) assembly instruction, which executes in one single clock cycle.
- The bitsliced single multiplication takes only $55/64 = 0.86$ cycle over $GF(2^6)$ and $335/64 = 5.23$ cycles for $GF(2^{12})$ and is invulnerable to cache-timing attacks. Thus, it is our champion implementation to be chosen for fast and secure arithmetic over $GF(2^N)$.
- For the second version of bitsliced implementation, we pack two words $X, X' \in GF(2^N)$ as $X'2^{2N} + X$. Then, the products XY and $X'Y'$ can be computed in one go by noticing that PCLMULQDQ($X'2^{2N} + X, Y'2^{2N} + Y$) = PCLMULQDQ($X'$, Y') 2^{4N} + (PCLMULQDQ(X, Y')$\oplus$ PCLMULQDQ(X', Y)) 2^{2N} + PCLMULQDQ(X, Y); hence, the results are obtained at bit indices $[6N, 4N]$ and $[2N, 0]$.

Table 3. Performances (in clock cycles) of several multiplication algorithms on an Intel(R) Core(TM) i7-4790 CPU @3.60GHz with one processor.

Multiplication Algorithm	**Algorithm**	$GF(2^6)$	$GF(2^{12})$ (*)	$GF((2^6)^2)$	**Constant-Time**
Tabulated log/antilog (**)	3, 4	8	11	20	No
Iterative, conditional reduction	5	27	51	133	No
Iterative, ASM with PCLMUL, conditional reduction	5	29	41	146	No
Iterative, unconditional reduction	6	30	58	155	Yes
Iterative, ASM with PCLMUL, unconditional reduction	6	35	65	225	Yes
Iterative, unconditional reduction, 1-bit-sliced (***) 64 computations in parallel	7	55/64	335/64	-	Yes
Iterative, ASM with PCLMUL, unconditional reduction, bit-sliced (****) 2 computations in parallel	8	55/2	95/2	-	Yes

For DAGS key generation, we chose to map elements from $GF((2^6)^2)$ to $GF(2^{12})$ and perform a bitsliced multiplication for secure and fast computation. The encapsulation process is carried over $GF(2^6)$ and we chose the bitsliced multiplication as well. Concerning the decapsulation, we mapped elements to $GF(2^{12})$ and kept the tabulated method because it is unimportant to secure this public-key process against cache-leakage.

5. Conclusions

In this paper, we compared several implementations of multiplication over the finite field $GF(2^N)$, for $N < 32$, on the basis of security and performance. Our analysis showed that log-antilog tabulated method and conditional iterative methods are vulnerable to cache-timing attacks. Moreover, for big values of N, the tabulated method becomes costly and tower fields are used to perform the arithmetic. We showed that this towering technique is slow and we proposed to map the elements to the isomorphic field for better performances.

To counter the cache-attacks, we presented two constant-time implementations: the iterative method with unconditional reduction which removes branches and thus is longer and the bitsliced implementation which is executed SIMD. We used DAGS, a code-based KEM submission to NIST's PQC call, as a case study to examine the different multiplications over $GF((2^6)^2)$. The conclusions are that the bitsliced implementation is faster than the tower multiplication and secure with respect to cache-attacks. It should be pointed out that our results also apply to secure and accelerated implementations of the other PQC algorithms listed, as well as AES and symmetric ciphers that run over finite fields $GF(2^N)$. Finally, note that all the algorithms tested in the paper are provided in C language in Appendix A.

Author Contributions: Conceptualization, J.-L.D.; investigation and writing: Y.E.H.; review and editing, A.F.; formal analysis, C.T.G.; project administration, S.G.; data curation, S.H.; visualization, O.N.; investigation, E.P.; investigation, visualization, A.S.

Funding: We acknowledge the support of the French *Programme d'Investissements d'Avenir* under the national project RISQ.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. C Code for Various Algorithms

```
#include <stdlib.h>
#include <stdint.h>

typedef uint16_t gf_t; /* Galois field elements */

#define gf_extd 6
#define gf_card (1 << gf_extd)
#define gf_ord ((gf_card)-1)
#define poly_primitive_subfield 67 // 0x43 (0b01000011; the bits are defined
//following the polynomial: X^6 + x + 1

/* Algorithm A1: Precomputation of the antilog table for F(2)[x]/x^6+x+1 */
static gf_t *gf_antilog;
void gf_init_antilog()
{
int i = 1;
int temp = 1 << (gf_extd - 1);
gf_antilog = (gf_t *)malloc((gf_card * sizeof(gf_t)));
gf_antilog[0] = 1; // Dummy value (not used)
for (i = 1; i < gf_ord; ++i)
{
gf_antilog[i] = gf_antilog[i - 1] << 1;
if ((gf_antilog[i - 1]) & temp)
{
// XOR with 67: X^6 + x + 1
gf_antilog[i] ^= poly_primitive_subfield;
}
```

```
}
gf_antilog[gf_ord] = 1;
}

/* Algorithm A2: Precomputation of the log table for F(2)[x]/x^6+x+1 */
static gf_t* gf_log;
void gf_init_log()
{
int i = 1;
gf_log = (gf_t *)malloc((gf_card * sizeof(gf_t)));
gf_log[0] = -1; // Dummy value (not used)
gf_log[1] = 0;
for (i = 1; i < gf_ord; ++i)
{
gf_log[gf_antilog[i]] = i;
}
}

/* Algorithm A3: Tabulated multiplication over GF(2^6)
/* Use precomputed tables to accelerate the multiplication: it uses the  */
/* algorithm 1 and 2 which are done just once to the DAGS initialization*/
/* This algorithm is not constant-time, so it is not protected.          */
#define gf_mult_tabulated(x,y)((y) ? gf_antilog[(gf_log[x]+gf_log[y])
% gf_ord] : 0)
//not constant-time

/* Algorithm A4: Tabulated multiplication over GF((2^6)^2)      */
/* Uses the algorithm 3, so it is not protected                 */
gf_t gf_mult_extension_tabulated(gf_t x, gf_t y)
{
gf_t a1, b1, a2, b2, a3, b3;
a1 = x >> 6;
b1 = x & 63;
a2 = y >> 6;
b2 = y & 63;
// not constant-time
a3 = gf_mult_tabulated(gf_mult_tabulated(a1, a2), 36) ^
gf_mult_tabulated(a1, b2)^ gf_mult_tabulated(b1, a2);
//36 is p_1 in the extension polynomial
b3 = gf_mult_tabulated(gf_mult_tabulated(a1, a2), 2)
^ gf_mult_tabulated(b1, b2); //2 is p_0 in the extension polynomial

return (a3 << 6) ^ b3;
}

/* Algorithm A5: Iterative multiplication over GF(2^6) with conditional reduction
/* The multiplication does not use the precomputed tables and the
ASM PCLMUL instruction  */
/* can be used. It is not constant-time.
gf_t gf_mult_iterative_conditional(gf_t x, gf_t y)
{
#ifndef PCLMUL
gf_t res,m;
res = 0; // this variable will contain the result
```

```
for(int i=0;i<6;++i) // For each coefficient of the polynomial
{
if(y&1==1) // Check the coefficient, it is not constant-time
{
res = res ^ x; // addition
}
y=y>>1;
//this shift permits to have the next coefficient b_i for the next iteration
x = x << 1;
if((x & 64) != 0)
// x must be reduced modulo X^6+X+1, 64 for 0x40, 0b01000000
{
x ^= 67; // 0x43 : X^6 + x + 1
}
}
return res;
#else
//using ASM PCLMUL instruction
uint32_t a, m;

// Multiplication
asm volatile ("movdqa␣␣␣␣␣%1,␣%%xmm0;\n\t"
"movdqa␣␣␣␣␣%2,␣%%xmm1;\n\t"
"pclmulqdq␣$0x00,␣%%xmm0,␣%%xmm1;\n\t"
"movdqa␣%%xmm1,␣%0;\n\t"
: "=x"(a)
: "x"((uint32_t)y), "x"((uint32_t)x)
: "%xmm0","%xmm1"
);

// reduction polynomial (conditional reduction)
for (int k=0; k<6; k++) { // For each coefficient of the polynomial
if (a >> (11-k)) //not constant-time
{
a ^= (67 << (5-k)); // 0x43 : X^6 + x + 1
}
}
return a&0xFFFF;
#endif
}

/* Algorithm A6: Iterative multiplication
over GF(2^6) with unconditional reduction                    */
/* The multiplication does not use the
precomputed tables and the ASM PCLMUL instruction       */
/* can be used. It is constant-time.
gf_t gf_mult_iterative_unconditional(gf_t x, gf_t y)
{
#ifndef PCLMUL
gf_t res,m;
res = 0;
for(int i=0;i<6;++i)
// For each coefficient of the polynomial, constant-time
{
m = -(y&1); //m is either 0xffff or 0x0000
```

```
res = res ^ (x&m); // addition
y=y>>1;
x = x << 1;
// x must be reduced modulo X^6+X+1
m=-(((x)>>6)&1);
x ^= m & 67;  // 0x43 : X^6 + x + 1
}
return res;
#else
//using ASM PCLMUL instruction
uint32_t a, m;

// multiplication
asm volatile ("movdqa␣␣␣␣%1,␣%%xmm0;\n\t"
"movdqa␣␣␣␣%2,␣%%xmm1;\n\t"
"pclmulqdq␣$0x00,␣%%xmm0,␣%%xmm1;\n\t"
"movdqa␣%%xmm1,␣%0;\n\t"
: "=x"(a)
: "x"((uint32_t)y), "x"((uint32_t)x)
:"%xmm0","%xmm1"
);

// reduction polynomial
for (int k=0; k<6; k++) {
m = -((a >> (11-k))&1);
a ^= ((67 << (5-k))&m); // 0x43 : X^6 + x + 1
}
return a&0xFFFF;
#endif
}

/* Algorithm A7: 1-bitsliced multiplication
over GF(2^6) (64 computations in parallel) */
/* A7.1: Transpositions */
void to_bitslice(gf_t *x, uint64_t *res) {
int i = 0;
for (i = 0; i<64; i++) {
res[0] |= (((((uint64_t)x[i])) & 1) << i);
res[1] |= (((((uint64_t)x[i]) >> 1) & 1) << i);
res[2] |= (((((uint64_t)x[i]) >> 2) & 1) << i);
res[3] |= (((((uint64_t)x[i]) >> 3) & 1) << i);
res[4] |= (((((uint64_t)x[i]) >> 4) & 1) << i);
res[5] |= (((((uint64_t)x[i]) >> 5) & 1) << i);
}
}

void from_bitslice(uint64_t *res, gf_t *x) {
int i = 0;
for (i = 0;i<64; i++) {
x[i] |= (((res[0] >> i) & 1));
x[i] |= (((res[1] >> i) & 1) << 1);
x[i] |= (((res[2] >> i) & 1) << 2);
x[i] |= (((res[3] >> i) & 1) << 3);
x[i] |= (((res[4] >> i) & 1) << 4);
x[i] |= (((res[5] >> i) & 1) << 5);
```

```
}
}

/* A7.2: 1-bit-sliced multiplication (SIMD code) */
void gf_multsubTab(gf_t *x, gf_t *y, gf_t *z)
{
uint64_t xbin[6];
uint64_t ybin[6];
uint64_t res[6];

xbin[0]=xbin[1]=xbin[2]=xbin[3]=xbin[4]=xbin[5] = 0;
ybin[0]=ybin[1]=ybin[2]=ybin[3]=ybin[4]=ybin[5] = 0;

// Transpose x and y
to_bitslice(x, xbin);
to_bitslice(y, ybin);

// Multiplication and reduction polynomial
//with 64 computations in parallel for a each coefficient of the polynomial
// constant-time
uint64_t const xbin05 = xbin[0] ^ xbin[5];
uint64_t const xbin54 = xbin[5] ^ xbin[4];
uint64_t const xbin43 = xbin[4] ^ xbin[3];
uint64_t const xbin32 = xbin[3] ^ xbin[2];
uint64_t const xbin21 = xbin[2] ^ xbin[1];

res[0] = (xbin[0] & ybin[0]);
res[1] = (xbin[1] & ybin[0]);
res[2] = (xbin[2] & ybin[0]);
res[3] = (xbin[3] & ybin[0]);
res[4] = (xbin[4] & ybin[0]);
res[5] = (xbin[5] & ybin[0]);

res[0] ^= (xbin[5] & ybin[1]);
res[1] ^= (xbin05  & ybin[1]);
res[2] ^= (xbin[1] & ybin[1]);
res[3] ^= (xbin[2] & ybin[1]);
res[4] ^= (xbin[3] & ybin[1]);
res[5] ^= (xbin[4] & ybin[1]);

res[0] ^= (xbin[4] & ybin[2]);
res[1] ^= (xbin54  & ybin[2]);
res[2] ^= (xbin05  & ybin[2]);
res[3] ^= (xbin[1] & ybin[2]);
res[4] ^= (xbin[2] & ybin[2]);
res[5] ^= (xbin[3] & ybin[2]);

res[0] ^= (xbin[3] & ybin[3]);
res[1] ^= (xbin43  & ybin[3]);
res[2] ^= (xbin54  & ybin[3]);
res[3] ^= (xbin05  & ybin[3]);
res[4] ^= (xbin[1] & ybin[3]);
res[5] ^= (xbin[2] & ybin[3]);

res[0] ^= (xbin[2] & ybin[4]);
```

```
res[1] ^= (xbin32  & ybin[4]);
res[2] ^= (xbin43  & ybin[4]);
res[3] ^= (xbin54  & ybin[4]);
res[4] ^= (xbin05  & ybin[4]);
res[5] ^= (xbin[1] & ybin[4]);

res[0] ^= (xbin[1] & ybin[5]);
res[1] ^= (xbin21 & ybin[5]);
res[2] ^= (xbin32 & ybin[5]);
res[3] ^= (xbin43 & ybin[5]);
res[4] ^= (xbin54 & ybin[5]);
res[5] ^= (xbin05 & ybin[5]);

// Transpose
from_bitslice(res, z);
}

/* Algorithm A8: Iterative, ASM with PCLMUL, unconditional reduction, bit-sliced
(2 computations in parallel) */
/* with PCMUL, 2 computations maximum are possible. It is constant-time.
void gf_mult_bitslice_2computations(gf_t *x, gf_t *y, gf_t *tab) {
// Transposition for computation in parallel
uint64_t x2 = x[1] << 12 | x[0] , y2 = y[1] << 12 | y[0] ;
uint64_t a, m, m1, s, m0;

// Multiplication
// As the output is on 64 bits max
asm volatile ("movdqa␣␣␣␣%1,␣%%xmm0;\n\t"
"movdqa␣␣␣␣%2,␣%%xmm1;\n\t"
"pclmulqdq␣$0x00,␣%%xmm0,␣%%xmm1;\n\t"
"movq␣%%xmm1,␣%0;\n\t"
: "=x"(a)
: "x"(y2), "x"(x2)
:"%xmm0","%xmm1"
);

// Polynomial reduction
for (int k=0; k<6; k++) {
m0 = a >> (11-k);
m = -(m0&1);
m1 = -((m0>>24)&1);
s = (67 << (5-k)); // 0x43 : X^6 + x + 1
a ^= (( s & m ) | ((s << 24) & m1 ));
}

// Transposition
tab[0] = a&0x3F;
tab[1] = (a>>24)&0x3F;
}

/* Algorithm A9: Mapping between GF((2^6)^2) and GF(2^12) */
//Conversion Matrix from GF((2^6)^2) to GF(2^12)
static const gf_t T[12] = {3857, 1140, 3330, 132, 286,
1954, 1938, 1208, 314, 3754, 2750, 188};
```

```
//Conversion Matrix from GF(2^12) to GF((2^6)^2)
static const gf_t Ti[12] = {3321, 3388, 4080, 2152,
3712, 3808, 2274, 4088, 1076, 3904, 1904, 3708};

//Hamming weight computation
static inline gf_t hamming_weight(gf_t n) {
#ifndef ASM_POPCNT
n = ((n & 0x0AAA) >> 1) + (n & 0x0555);
n = ((n & 0x0CCC) >> 2) + (n & 0x0333);
n = ((n & 0x00F0) >> 4) + (n & 0x0F0F);
n = ((n & 0x0F00) >> 8) + (n & 0x00FF);
#else
//using ASM
asm (
"POPCNT %1, %0\n" // Count of Number of Bits Set to 1
: "=r" (n)
: "mr" (n)
: "cc"
);
#endif
return n;
}

/* A9.1: Conversion from GF(2^12) to GF((2^6)^2)                */
/* with the convertion Matrix Ti from GF(2^12) to GF((2^6)^2)   */
gf_t iconv_bit(gf_t x)
{
gf_t res = 0;
for (int i=0; i<12; i++) {
res |= (hamming_weight(x & Ti[i])&1) << i; // Ti defined in (3.5)
}
return res;
}

/* A9.2: Conversion from GF((2^6)^2) to GF(2^12)                */
/* with the convertion Matrix T from GF((2^6)^2) to GF(2^12)    */
gf_t conv_bit(gf_t x)
{
gf_t res = 0;
for (int i=0; i<12; i++) {
res |= (hamming_weight(x & T[i])&1) << i; // T defined in (3.5)
}
return res;
}
```

References

1. Paar, C. Efficient VLSI architectures for Bit-Parallel Computation in Galois Fields. Ph.D. Thesis, Institute for Experimental Mathematics, University of Essen, Duisburg, Germany, 1994. Available online: https://tinyurl.com/yc7hmfmo (accessed on 18 September 2018).
2. Sunar, B.; Savas, E.; Koç, Ç.K. Constructing composite field representations for efficient conversion. *IEEE Trans. Comput.* **2003**, *52*, 1391–1398. [CrossRef]
3. Round 1 Submissions (30/11/2017)—Post-Quantum Cryptography. Available online: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions (accessed on 18 September 2018).

4. DAGS project. Available online: http://www.dags-project.org (accessed on 18 September 2018).
5. NIST/ITL/CSD. Advanced Encryption Standard (AES). FIPS PUB 197, 11/26/2001. (Also ISO/IEC 18033-3:2010). Available online: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (accessed on 18 September 2018).
6. McEliece, R.J. A public-key cryptosystem based on algebraic coding theory. *JPL DSN Prog. Rep.* **1978**, *42–44*, 114–116.
7. Rivest, R.L.; Shamir, A.; Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [CrossRef]
8. Diffie, W.; Hellman, M. New directions in cryptography. *IEEE Trans. Inf. Theory* **1976**, *22*, 644–654. [CrossRef]
9. Bardet, M.; Chaulet, J.; Dragoi, V.; Otmani, A.; Tillich, J.P. Cryptanalysis of the McEliece public key cryptosystem based on polar codes. In Proceedings of the 7th International Conference on Post-Quantum Cryptography (PQCrypto 2016), Fukuoka, Japan, 24–26 February 2016; Springer: Berlin, Germany, 2016; pp. 118–143.
10. Post-Quantum Cryptography Challenge (ongoing). Available online: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions (accessed on 18 September 2018).
11. Yarom, Y.; Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 719–732.
12. Facon, A.; Guilley, S.; Lec'hvien, M.; Schaub, A.; Souissi, Y. Detecting cache-timing vulnerabilities in post-quantum cryptography algorithms. In Proceedings of the 3rd IEEE International Verification and Security Workshop, Hotel Cap Roig, Platja d'Aro, Costa Brava, Spain, 2–4 July 2018.
13. Lidl, R.; Niederreiter, H. *Finite Fields*; Cambridge University Press: Cambridge, UK, 1997.
14. Tromer, E.; Osvik, D.A.; Shamir, A. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* **2010**, *23*, 37–71. [CrossRef]
15. Aciiçmez, O.; Koç, Ç.K.; Seifert, J.P. On the power of simple branch prediction analysis. In Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, Singapore, 20–22 March, 2007; pp. 312-320.
16. Aciiçmez, O.; Koç, Ç.K.; Seifert, J. Predicting Secret Keys Via Branch Prediction. In Proceedings of the Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, 5–9 February 2007; pp. 225–242.
17. Biham, E. A Fast New DES Implementation in Software. In Proceedings of the the Fourth International Workshop on Fast Software Encryption, Haifa, Israel, 20–22 January 1997; pp. 260–272.
18. Matsui, M.; Nakajima, J. On the Power of Bitslice Implementation on Intel Core2 Processor. In Proceedings of the Cryptographic Hardware and Embedded Systems, Vienna, Austria, 10–13 September 2007; pp. 121–134. [CrossRef]
19. Berlekamp, E.; McEliece, R.; van Tilborg, H. On the Inherent Intractability of Certain Coding Problems. *IEEE Trans. Inform. Theory* **1978**, *24*, 384–386.
20. Misoczki, R.; Barreto, P.S.L.M.B. Compact McEliece Keys from Goppa Codes. In Proceedings of the 16th Workshop on Selected Areas in Cryptography (SAC 2009), Calgary, AB, Canada, 13–14 August 2009; pp. 376–392. [CrossRef]
21. Persichetti, E. Compact McEliece keys based on quasi-dyadic Srivastava codes. *J. Math. Cryptol.* **2012**, *6*, 149–169.
22. Faugère, J.C.; Otmani, A.; Perret, L.; Tillich, J.P. Algebraic Cryptanalysis of McEliece Variants with Compact Keys. In Proceedings of the 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, France, 30 May–3 June 2010; pp. 279–298. [CrossRef]
23. Prange, E. The use of information sets in decoding cyclic codes. *IRE Trans. Inf. Theory* **1962**, *8*, 5–9.
24. Peters, C. Information-Set Decoding for Linear Codes over F_q. In Proceedings of the The Third International Workshop on Post-Quantum Cryptography, Darmstadt, Germany, 25–28 May 2010; pp. 81–94.