Diffusion Matrices from Algebraic-Geometry Codes with Efficient SIMD Implementation*

Daniel Augot[1,2], Pierre-Alain Fouque[3,4], and Pierre Karpman[1,5,2]

[1] Inria, France
[2] LIX — École Polytechnique, France
[3] Université de Rennes 1, France
[4] Institut universitaire de France, France
[5] Nanyang Technological University, Singapore
{daniel.augot,pierre.karpman}@inria.fr, pierre-alain.fouque@irisa.fr

Abstract. This paper investigates large linear mappings with very good diffusion and efficient software implementations, that can be used as part of a block cipher design. The mappings are derived from linear codes over a small field (typically $\mathbf{F}_{2^4}$) with a high dimension (typically 16) and a high minimum distance. This results in diffusion matrices with equally high dimension and a large branch number. Because we aim for parameters for which no MDS code is known to exist, we propose to use more flexible *algebraic-geometry* codes.
We present two simple yet efficient algorithms for the software implementation of matrix-vector multiplication in this context, and derive conditions on the generator matrices of the codes to yield efficient encoders. We then specify an appropriate code and use its automorphisms as well as random sampling to find good such matrices.
We provide concrete examples of parameters and implementations, and the corresponding assembly code. We also give performance figures in an example of application which show the interest of our approach.

Keywords: Diffusion matrix, algebraic-geometry codes, algebraic curves, SIMD, vector implementation, SHARK.

1 Introduction

The use of *MDS* matrices over finite fields as a linear mapping in block cipher design is an old trend, followed by many prominent algorithms such as the AES/Rijndael family [6]. These matrices are called MDS as they are derived from *maximum distance separable* linear error-correcting codes, which achieve the highest minimum distance possible for a given length and dimension. This notion of minimum distance coincides with the one of *branch number* of a mapping [6], which is a measure of the effectiveness of a diffusion layer. MDS matrices thus have an optimal diffusion, in a cryptographic sense, which makes them attractive for cipher designs.

The good security properties that can be derived from MDS matrices are often counter-balanced by the cost of their computation. The standard matrix-vector product is quadratic in the dimension of the vector, and finite field operations are not always efficient. For that reason, there is often a focus on finding matrices allowing efficient implementations. For instance, the AES matrix is circulant and has small coefficients. More recently, the PHOTON hash function [8] introduced the use of matrices that can be obtained as the power of a companion matrix, which sparsity may be useful in lightweight hardware implementations. The topic of finding such so-called recursive diffusion layers has been quite active in the past years, and led to a series of papers investigating some of their various aspects [17,22,2]. One of the most recent developments shows how to systematically construct some of these matrices from BCH codes [1]. This allows in particular to construct very large recursive MDS matrices, for instance of dimension 16 over $\mathbf{F}_{2^8}$. This defines a linear mapping over a full 128-bit block with excellent diffusion properties, at a moderate hardware implementation cost.

As interesting as it may be in hardware, the cost in software of a large linear mapping tends to make these designs rather less attractive than more balanced solutions. An early attempt to use a large matrix was the block cipher SHARK, a Rijndael predecessor [15]. It is a 64-bit cipher which

* A shorter version of this paper is to appear in the proceedings of SAC 2014.

uses an MDS matrix of dimension 8 over $\mathbf{F}_{2^8}$ for its linear diffusion. The usual technique for implementing such a mapping in software is to rely on a table of precomputed multiples of the matrix rows. However, table-based implementations now tend to be frown upon as they may lead to *timing attacks* [20], and this could leave ciphers with a structure similar to SHARK's without reasonable software implementations when resistance to these attacks is required. Yet, such designs also have advantages of their own; their diffusion acts on the whole state at every round, and therefore makes structural attacks harder, while also ensuring that many S-Boxes are kept active. Additionally, the simplicity of the structure makes it arguably easier to analyze than in the case of most ciphers.

Our contributions. In this work, we revisit the use of a *SHARK structure* for block cipher design and endeavour to find good matrices and appropriate algorithms to achieve both a linear mapping with very good diffusion and efficient software implementations that are not prone to timing attacks. To be more specific on this latter point, we target software running on 32 or 64-bit CPUs featuring an SIMD vector unit.

An interesting way of trying to meet both of these goals is to decrease the size of the field from $\mathbf{F}_{2^8}$ to $\mathbf{F}_{2^4}$. However, according to the *MDS conjecture*, there is no MDS code over $\mathbf{F}_{2^4}$ of length greater than 17, and no such code is known [14]. Because a diffusion matrix of dimension n is typically obtained from a code of length $2n$, MDS matrices over $\mathbf{F}_{2^4}$ are therefore restricted to dimensions less than 8. Hence, the prospect of finding an MDS matrix over $\mathbf{F}_{2^4}$ diffusing on more than $8 \times 4 = 32$ bits is hopeless. Obviously, 32 bits is not enough for a large mapping *à la* SHARK. We must therefore search for codes with a slightly smaller minimum distance in the hope that they can be made longer.

Our proposed solution to this problem is to use *algebraic-geometry codes* [21], as they precisely offer this tradeoff. One way of defining these codes is as evaluation codes on algebraic curves; thus our proposal brings a nice connection between these objects and symmetric cryptography. Although elliptic and hyperelliptic curves are now commonplace in public-key cryptography, we show a rare application of an hyperelliptic curve to the design of block ciphers. We present a specific code of length 32 and dimension 16 over $\mathbf{F}_{2^4}$ with minimum distance 15, which is only 2 less than what an MDS code would achieve. This lets us deriving a very good diffusion matrix on $16 \times 4 = 64$ bits in a straightforward way. Interestingly, this matrix can also be applied to vectors over an extension of $\mathbf{F}_{2^4}$ such as $\mathbf{F}_{2^8}$, while keeping the same good diffusion properties. This allows for instance to increase the diffusion to $16 \times 8 = 128$ bits.

We also study two simple yet efficient algorithms for implementing the matrix-vector multiplication needed in a SHARK structure, when a *vector permute* instruction is available. From one of these, we derive conditions on the matrix to make the product faster to compute, in the form of a cost function; we then search for matrices with a low cost, both randomly, and by using automorphisms of the code and of the hyperelliptic curve on which it is based. The use of codes automorphisms to derive efficient encoders is not new [10,5], but it is not generally applied to the architecture and dimensions that we consider in our case.

We conclude this paper by presenting examples of performance figures of assembly implementations of our algorithms when used as the linear mapping of a block cipher.

Structure of the paper. We start with a few background notions in §2. We then present our algorithms for matrix-vector multiplication and their context in §3, and derive a cost function for the implementation of matrices. This is followed by the definition of the algebraic-geometry code used in our proposed linear mapping, and a discussion of how to derive efficient encoders in §4. We conclude with insights into the performance of the mapping when used over both $\mathbf{F}_{2^4}$ and $\mathbf{F}_{2^8}$ in §5.

2 Preliminaries

We note $\mathbf{F}_{2^m}$ the finite field with 2^m elements. We often consider $\mathbf{F}_{2^4}$, and implicitly use this specific field if not mentioned otherwise. W.l.o.g. we use the representation $\mathbf{F}_{2^4} \cong \mathbf{F}_2[\alpha]/(\alpha^4+\alpha+1)$. We freely use "integer representation" for elements of $\mathbf{F}_{2^4}$ by writing $n \in \{0\dots15\} = \sum_{i=0}^{3} a_i 2^i$ to represent the element $x \in \mathbf{F}_{2^4} = \sum_{i=0}^{3} a_i \alpha^i$.

Bold variables denote vectors (in the sense of elements of a vector space), and subscripts are used to denote their i^{th} coordinate, starting from zero. For instance, $\mathbf{x} = (1,2,7)$ and $\mathbf{x}_2 = 7$. If M is a matrix of n columns, we call $\mathbf{m}^i = (M_{i,j},\ j = 0\dots n-1)$ the row vector formed from the coefficients of its i^{th} row. We use angle brackets "$\langle$" and "$\rangle$" to write ordered sets.

Arrays, or tables, (in the sense of software data structures) are denoted by regular variables such as x or T, and their elements are accessed by using square brackets. For instance, $T[i]$ is the i^{th} element of the table T, starting from zero.

We conclude with two definitions.

Definition 1 (Systematic form and dual of a code) *Let $\mathscr{C}$ be an $[n,k,d]_{\mathbf{F}_{2^m}}$ code of length n, dimension k and minimum distance d with symbols in $\mathbf{F}_{2^m}$. A generator matrix for $\mathscr{C}$ is in* systematic form *if it is of the form $(I_k\ A)$, with I_k the identity matrix of dimension k and A a matrix of k rows and $n-k$ columns. A systematic generator matrix for the* dual *of $\mathscr{C}$ is given by $(I_{n-k}\ A^t)$.*

Definition 2 (Branch number [6]) *Let A be the matrix of a linear mapping over $\mathbf{F}_{2^m}$, and $\mathrm{w}_m(\mathbf{x})$ be the number of non-zero positions of the vector $\mathbf{x}$ over $\mathbf{F}_{2^m}$. Then the* differential branch number *of A is equal to $\min_{\mathbf{x}\neq 0}(\mathrm{w}_m(x)+\mathrm{w}_m(A(x)))$, and the* linear branch number *of A is equal to $\min_{\mathbf{x}\neq 0}(\mathrm{w}_m(x)+\mathrm{w}_m(A^t(x)))$.*

Note that if A is such that $(I_k\ A)$ is a generator matrix of a code of minimum distance d which dual code has minimum distance d', then A has a differential (resp. linear) branch number of d (resp. d').

3 Efficient algorithms for matrix-vector multiplication

This section presents software algorithms for matrix-vector multiplication over $\mathbf{F}_{2^4}$. We focus on square matrices of dimension 16. This naturally defines linear operations on 64 bits, which can also be extended to 128 bits, as it will be made clear in §5. Both cases are a common block size for block ciphers.

Targeted architecture. The algorithms in this section target CPUs featuring vector instructions, including in particular a *vector shuffle* instruction such as Intel's `pshufb` from the SSSE3 instruction set extension [11]. These instructions are now widespread and have already been used successfully in fast cryptographic implementations, see *e.g.* [9,19,3]. We mostly considered SSSE3 when designing the algorithms, but other processor architectures do feature vector instructions. This is for instance the case of ARM's NEON extensions, which may also yield efficient implementations, see *e.g.* [4]. We do not consider these explicitly in this paper, however.

Because it plays an important role in our algorithms, we briefly recall the semantics of `pshufb`. The `pshufb` instruction takes two 128-bit inputs[1]. The first (the destination operand) is an `xmm` SSE vector register which logically represents a vector of 16 bytes. The second (the source operand) is either a similar `xmm` register, or a 128-bit memory location. The result of calling `pshufb` x y is to overwrite the input x with the vector x' defined by:

$$x'[i] = \begin{cases} x[\lfloor y[i] \rfloor_4] & \text{if the most significant bit of } y[i] \text{ is not set} \\ 0 & \text{otherwise} \end{cases}$$

[1] The instruction can actually also be used on 64-bit operands, but we do not consider this possibility here.

where $\lfloor\cdot\rfloor_4$ denotes truncation to the 4 least significant bits. This instruction allows to arbitrarily *shuffle* a vector according to a mask, with possible repetition and omission of some of the vector values[2]. Notice that this instruction can also be used to perform 16 parallel 4-to-8-bit table lookups: let us call T this table; take as first operand to `pshufb` the vector $x = (T[i], i = 0\ldots15)$, as second operand the vector $y = (a, b, c, d, \ldots)$ on which to perform the lookup; then we see that the first byte of the result is $x[y[0]] = T[a]$, the second is $x[y[1]] = T[b]$, etc.

Finally, there is a three-operand variant of this instruction in the more recent AVX instruction set and onward [11], which allows not to overwrite the first operand.

Targeted properties. In this paper we focus solely on algorithms that can easily be implemented in a way that makes them immune to timing-attacks [20]. Specifically, we consider the matrix as a known constant but the vector as a secret, and we wish to perform the multiplication without secret-dependent branches or memory accesses. It might not always be important to be immune (or even partially resistant) to this type of attacks, but we consider that it should be important for any cryptographic primitive or structure to possibly be implemented in such a way. Hence we try to find efficient such implementations for the `SHARK` structure and therefore for dense matrix-vector multiplications.

We now go on to describe the algorithms. In all of the remainder of this section, $\mathbf{x}$ and $\mathbf{y}$ are two (column) vectors of $\mathbf{F}_{2^4}^{16}$, and M a matrix of $\mathscr{M}_{16}(\mathbf{F}_{2^4})$. We first briefly recall the principle of table implementations, which are unsatisfactory when timing attacks are taken into account.

3.1 Table implementation

We wish to compute $\mathbf{y} = M \cdot \mathbf{x}$. The idea behind this algorithm is to use table lookups to perform the equivalent multiplication $\mathbf{y}^t = \mathbf{x}^t \cdot M^t$, *i.e.* $\mathbf{y}^t = \sum_{i=0}^{15} \mathbf{x}_i \cdot (\mathbf{m}^t)^i$ (where $(\mathbf{m}^t)^i$ is the i^{th} row of M^t). This can be computed efficiently by tabulating beforehand the products $\lambda \cdot (\mathbf{m}^t)^i, \lambda \in \mathbf{F}_{2^4}$ (resulting in 16 tables, each of 16 entries of 64 bits), and then for each multiplication by accessing the table for $(\mathbf{m}^t)^i$ at the index $\mathbf{x}_i$ and summing all the retrieved table entries together. This only requires 16 table lookups per multiplication. However, the memory accesses depend on the value of $\mathbf{x}$, which makes this algorithm inherently vulnerable to timing attacks.

Note that there is a more memory-efficient alternative implementation of this algorithm which consists in computing each term $\lambda \cdot (\mathbf{m}^t)^i$ with a single `pshufb` instruction instead of using a table-lookup. In that case, only the 16 multiplication tables need to be stored, but their accesses still depend on the secret value $\mathbf{x}$.

3.2 A generic constant-time algorithm

We now describe our first algorithm, which can be seen as a variant of table multiplication that is immune to timing attacks. The idea consists again in computing the right multiplication $\mathbf{y}^t = \mathbf{x}^t \cdot M^t$, *i.e.* $\mathbf{y}^t = \sum_{i=0}^{15} \mathbf{x}_i \cdot (\mathbf{m}^t)^i$. However, instead of tabulating the results of the scalar multiplication of the matrix rows $(\mathbf{m}^t)^i$, those are always recomputed, in a way that does not explicitly depend on the value of the scalar.

Description of algorithm 1. We give the full description of Alg. 1 in appendix A.1, and focus here on the intuition. We want to perform the scalar multiplication $\lambda \cdot \mathbf{z}$ for an unknown scalar λ and a known, constant vector $\mathbf{z}$, over $\mathbf{F}_{2^4}$. Let us write λ as the polynomial $\lambda_3 \cdot \alpha^3 + \lambda_2 \cdot \alpha^2 + \lambda_1 \cdot \alpha + \lambda_0$ with coefficients in $\mathbf{F}_2$. Then, the result of $\lambda \cdot \mathbf{z}$ is simply $\lambda_3 \cdot (\alpha^3 \cdot \mathbf{z}) + \lambda_2 \cdot (\alpha^2 \cdot \mathbf{z}) + \lambda_1 \cdot (\alpha \cdot \mathbf{z}) + \lambda_0 \cdot \mathbf{z}$. Thus we just need to precompute the products $\alpha^i \cdot \mathbf{z}$, select the right ones with respect to

[2] We will use the word *shuffle* with this precise meaning in the remainder of this paper.

the binary representation of λ, and add these together. This can easily be achieved thanks to a *broadcast* function defined as:

$$broadcast(x,i)_n = \begin{cases} \mathbf{1}_n \text{ if the } i^{\text{th}} \text{ bit of } x \text{ is set} \\ \mathbf{0}_n \text{ otherwise} \end{cases}$$

where $\mathbf{1}_n$ and $\mathbf{0}_n$ denote the n-bit binary string made all of one and all of zero respectively. The full algorithm then just consists in using this scalar-vector multiplication 16 times, one for each row of the matrix.

Implementation of algorithm 1 with SSSE3 instructions. We now consider how to efficiently implement algorithm 1 in practice. The only non-trivial operation is the *broadcast* function, and we show that this can be performed with only one or two `pshufb` instructions.

To compute $broadcast(\lambda,i)_{64}$, with λ a 4-bit value, we can use a single `pshufb` with first operand x, such that $x[j] = 11111111_2$ if the i^{th} bit of j is set and 0 otherwise, and with second operand $y = (\lambda,\lambda,\lambda,\ldots)$. The result of `pshufb` x y is indeed $(x[\lambda],x[\lambda],\ldots)$ which is $\mathbf{1}_{64}$ if the i^{th} bit of λ is set, and $\mathbf{0}_{64}$ otherwise, that is $broadcast(\lambda,i)_{64}$.

In practice, the vector x can conveniently be constructed offline and stored in memory, but the vector y might not be readily available before performing this computation[3]. However, it can easily be computed thanks to an additional `pshufb`. Alternatively, if the above computation is done with a vector $y = (\lambda,?,?,\ldots)$ instead (with ? denoting unknown values) and call z its result $(x[\lambda],?,?,\ldots)$, then we have $broadcast(\lambda,i)_n =$ `pshufb` z $(0,0,\ldots)$.

In the specific case of matrices of dimension 16 over $\mathbf{F}_{2^4}$, one can take advantage of the 128-bit wide `xmm` registers by interleaving, say, $8\cdot\mathbf{x}$ with $4\cdot\mathbf{x}$, and $2\cdot\mathbf{x}$ with $\mathbf{x}$, and by computing a slightly more complex version of the *broadcast* function $broadcast(x,i,j)_{2n}$ which interleaves $broadcast(x,i)_n$ with $broadcast(x,j)_n$. In that case, an implementation of one step of algorithm 1 only requires two *broadcast* calls, two logical *and*, folding back the interleaved vectors (which only needs a couple of logical shift and exclusive or), and adding the folded vectors together. We give a snippet of such an implementation in appendix B.1.

3.3 A faster algorithm exploiting matrix structure

The above algorithm is already reasonably efficient, and has the advantage of being completely generic w.r.t. the matrix. Yet, better solutions may exist in more specific cases. We present here an alternative that can be much faster when the matrix possesses a particular structure.

The idea behind this second algorithm is to take advantage of the fact that in a matrix-vector product, the same constant values may be used many times in finite-field multiplications. Hence, we try to take advantage of this fact by performing those in parallel. The fact that we now focus on multiplications by constants (*i.e.* matrix coefficients) allows us to compute these multiplications with a single `pshufb` instead of using the process from algorithm 1.

Description of algorithm 2. We give the full description of Alg. 2 in appendix A.2, and focus here on the intuition. Let us first consider a small example, and compute $M\cdot\mathbf{x}$ defined as:

$$\begin{pmatrix} 1 & 0 & 2 & 2 \\ 3 & 1 & 2 & 3 \\ 2 & 3 & 3 & 2 \\ 0 & 2 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}. \tag{1}$$

It is obvious that this is equal to:

$$\begin{pmatrix} x_0 \\ x_1 \\ 0 \\ x_3 \end{pmatrix} + 2\cdot\begin{pmatrix} x_2 \\ x_2 \\ x_0 \\ x_1 \end{pmatrix} + 2\cdot\begin{pmatrix} x_3 \\ 0 \\ x_3 \\ 0 \end{pmatrix} + 3\cdot\begin{pmatrix} 0 \\ x_0 \\ x_1 \\ x_2 \end{pmatrix} + 3\cdot\begin{pmatrix} 0 \\ x_3 \\ x_2 \\ 0 \end{pmatrix},$$

[3] And because it depends on what we assume to be a secret value, it cannot either be fetched from memory.

where both the constant multiplications of the vector $(x_0\ x_1\ x_2\ x_3)^t$ and the shuffles of its coefficients can be computed with a single `pshufb` instruction each, while none of these operations directly depends on the value of the vector. This type of decomposition can be done for any matrix, but the number of operations depends on the value of its coefficients.

We now sketch one way of obtaining an optimal decomposition as above. We consider a matrix product $M \cdot \mathbf{x}$ with M constant and $\mathbf{x}$ unknown, where $\mathbf{x}$ is seen as the formal arrangement of variables $\mathbf{x}_i$. Let us define $\mathscr{S}(M,\gamma)$ as one of the minimal sets of shuffles of coefficients of $\mathbf{x}$, such that there exists a unique vector $\mathbf{z} \in \mathscr{S}(M,\gamma)$ with $\mathbf{z}_i = \mathbf{x}_j$ iff $M_{i,j} = \gamma$. For instance, in the above example, we have $\mathscr{S}(M,2) = \{(x_2\ x_2\ x_0\ x_1)^t, (x_3\ 0\ x_3\ 0)^t\}$. Equivalently, we could have taken $\mathscr{S}(M,2) = \{(x_3\ x_2\ x_3\ 0)^t, (x_2\ 0\ x_0\ x_1)^t\}$. These sets are straightforward to compute from this particular matrix, and so are they in the general case.

From the definition of $\mathscr{S}$, it is clear that we have $M \cdot \mathbf{x} = \sum_{\gamma \in \mathbf{F}_{2^4}^*} \sum_{\mathbf{s} \in \mathscr{S}(M,\gamma)} \gamma \cdot \mathbf{s}$. Once the values of the sets $\mathscr{S}$ have been determined, it is clear that we only need to compute this sum to get our result, and this is precisely what this second algorithm does.

Cost of algorithm 2. The cost of computing a matrix-vector product with algorithm 2 depends on the coefficients of the matrix, since the size of the sets $\mathscr{S}(M,\gamma)$ depends both on the density of the matrix and of how its coefficients are arranged.

If we assume that a vector implementation of this algorithm is used, and if the dimension and the field of the matrix are well chosen, we can assume that both the scalar multiplication of $\mathbf{x}$ by a constant and its shuffles can be computed with a single `pshufb` and a few ancillary instructions. Hence, we can define a cost function for a matrix with respect to its implementation with algorithm 2 to be $\mathit{cost2}(M) = \left(\sum_{\gamma \in \mathbf{F}_{2^4}*} \mathbb{1}(\mathscr{S}(M,\gamma)) + \#\mathscr{S}(M,\gamma)\right) - \mathbb{1}(\mathscr{S}(M,1))$, where $\mathbb{1}(\mathscr{E})$ with $\mathscr{E}$ a set is one if $\mathscr{E} \neq \emptyset$, and zero otherwise. We may notice that $\#\mathscr{S}(M,\gamma)$ is equal to the maximum number of occurrence of γ in a single row of M, and the *cost2* function is therefore easy to compute. As an example the cost of the matrix M from equation 1 is 7.

In order to find matrices that minimize the *cost2* function, we would like to minimize the sum of the maximum number of occurrence of γ for every $\gamma \in \mathbf{F}_{2^4}^*$. A simple observation is that for matrices with the same number of non-zero coefficients, this amount is minimal when every row can be deduced by permutation of a single one; an important particular case being the one of *circulant* matrices. More generally, we can heuristically hope that the cost of a matrix will be low if all of its rows can be deduced by permutation of a small subset thereof.

We can try to estimate the minimum cost for an arbitrary dense circulant matrix of dimension 16 over $\mathbf{F}_{2^4}$. It is fair to assume that nearly all of the values of $\mathbf{F}_{2^4}$ should appear as coefficients of such a matrix, 14 of them needing a multiplication. Additionally, 15–16 permutations are needed if all the rows are to be different. Hence we can assume that the *cost2* function of such a matrix is about 30.

Finally, let us notice that special cases of this algorithm have already been used for circulant matrices, namely in the case of the AES `MixColumn` matrix [9,3].

Implementation of algorithm 2 with SSSE3 instructions. The implementation of algorithm 2 is quite straightforward. We give nonetheless a small code snippet in appendix B.2.

3.4 Performance

In Table 2 of §5, we give a few performance figures for ciphers with a `SHARK` structure using assembly implementations of algorithms 1 and 2 for their linear mapping. From there it can be seen without surprise that algorithm 2 is more efficient if the matrix is well chosen. However, algorithm 1 still performs reasonably well, without imposing any condition on the matrix.

4 Diffusion matrices from algebraic-geometry codes

In this section, we present so-called algebraic-geometry codes and show how they can give rise to diffusion matrices with interesting parameters. We also focus on implementation aspects, and investigate how to find matrices with efficient implementations with respect to the algorithms of §3, and in particular algorithm 2.

4.1 A short introduction to algebraic-geometry codes

We first briefly present the concept of algebraic-geometry codes (or AG codes for short), which are linear codes, and how to compute their generator matrices. Because the codes are linear, these encoders are matrices. We do not give a complete description of AG codes, and refer to *e.g.* [21] for a more thorough treatment. We present a class of AG codes as a generalization of Reed-Solomon (RS) codes. We give a quick presentation of RS codes in appendix C for the reader not familiar with them.

We see AG codes as *evaluation codes*: to build the codeword for a message w, we consider w as a function, and the codeword as a vector of values of this function evaluated on some "elements". In our case, the elements are points of the two-dimensional affine space $\mathbf{A}^2(\mathbf{F}_{2^m})$, and the functions are polynomials in two variables, that is elements of $\mathbf{F}_{2^m}[x,y]$. The core idea of AG codes is to consider points of a (smooth) projective curve of the projective space $\mathbf{P}^2(\mathbf{F}_{2^m})$ and functions from the curve's function space. Points at infinity are never included in the (ordered) set of points. However, points of the curve at infinity *are* useful in defining the curve's function (sub)-space, which is why we do consider the curve in the projective space instead of the affine one.

We first give the definition of the Riemann-Roch space in the special case where it is defined from a divisor made of a single point at infinity. We refer to *e.g.* [18] or [21] for a more complete and rigorous definition.

Definition 3 (Riemann-Roch space) *Let $\mathscr{X}$ be a smooth projective curve of $\mathbf{P}^2(\mathbf{F}_{2^m})$ defined by the homogeneous polynomial $p(x,y,z)$, and let $p'(x,y)$ be the dehomogenized of p. We define $\mathbf{F}_{2^m}[\mathscr{X}] = \mathbf{F}_{2^m}[x,y]/p'$ as the* coordinate ring *of $\mathscr{X}$, and its corresponding quotient field $\mathbf{F}_{2^m}(\mathscr{X})$ as the* function field *of $\mathscr{X}$. Assume Q is the only point of $\mathscr{X}$ at infinity, and let r be a positive integer. The Riemann-Roch space $\mathscr{L}(rQ)$ is the set of all functions of $\mathbf{F}_{2^m}(\mathscr{X})$ with poles only at Q of order less than r. This is a finite-dimensional $\mathbf{F}_{2^m}$-vector-space. Furthermore, let $\mathrm{o}_Q(x)$ and $\mathrm{o}_Q(y)$ be the order of the poles of x and y in Q[4], then a basis of $\mathscr{L}(rQ)$ is formed by all the monomial functions $x^i y^j$ that are such that $i\cdot\mathrm{o}_Q(x) + j\cdot\mathrm{o}_Q(y) \leq r$.*

This space is particularly important because of the following theorem, which links its dimension with the genus of $\mathscr{X}$ [18].

Theorem 1 (Riemann and Roch) *Let $\mathscr{L}(rQ)$ be a Riemann-Roch space defined on $\mathscr{X}$, and g be the genus of $\mathscr{X}$. We have* $\dim(\mathscr{L}(rQ)) \geq r+1-g$, *with equality when $r > 2g-2$.*

We have also mentioned earlier that a basis for a space $\mathscr{L}(rQ)$ can be computed as soon as the order of the poles of x and y in Q are known, and the dimension of the space can obviously be computed from the basis. In practice, computing $\mathrm{o}_Q(x)$ and $\mathrm{o}_Q(y)$ can be done from a local parameterization of x and y in Q. Both this parameterization and the values $\mathrm{o}_Q(x)$ and $\mathrm{o}_Q(y)$ can easily be obtained from a computational algebra software such as Magma. Again, we refer to [21] for more details.

We are now ready to define a simple class of AG codes.

[4] We slightly abuse the notations here and actually mean x/z and y/z. But we prefer manipulating their dehomogenized equivalents x and y. It is obvious that x/z and y/z indeed have poles in Q, which is at infinity and hence has a zero z coordinate.

Definition 4 (Algebraic-Geometry codes) *Let $\mathscr{X}$ be a smooth projective curve of $\mathbf{P}^2(\mathbf{F}_{2^m})$ with a unique point Q at infinity, and call $\#\mathscr{X}$ its number of affine points (that is not counting Q). Assume that $\#\mathscr{X} \geq n$ and let r be s.t.* $\dim(\mathscr{L}(rQ)) = k$, *and call $(f_0, \ldots, f_{k-1})$ one basis of this space. We define the codeword of the $[n,k,d]_{\mathbf{F}_{2^m}}$ algebraic-geometry code $\mathscr{C}_{AG}$ associated with the message $\mathbf{m}$ as the vector $\sum_{i=0\ldots k-1}(\mathbf{m}_i \cdot f_i(p_j), j = 0\ldots n-1)$, where $P = \langle p_0, \ldots, p_{n-1}\rangle$ is an ordered set of points of $\mathscr{X}/\{Q\}$. The code $\mathscr{C}_{AG}$ is the set of all such codewords.*

These codes have the following properties: for fixed parameters n and k and a curve $\mathscr{X}$, there are $\binom{\#\mathscr{X}}{n} \cdot n!$ *equivalent* codes, which corresponds to the number of possible ordered sets P; it is also obvious that the maximal length of a code over $\mathscr{X}$ is $\#\mathscr{X}$. We also have the following proposition [21]:

Proposition 1 *Let $\mathscr{C}_{AG}$ be a code of length n and dimension k, and let r be an integer such that* $\dim(\mathscr{L}(rQ)) = k$. *Then the minimum distance of $\mathscr{C}_{AG}$ is at least $n-r$. If $\mathscr{X}$ is of genus g and $r > 2g-2$, this is equal to $n-((k-1)+g) = n-k-g+1$. Therefore, the "gap" between this code and an MDS code of the same length is g. The same holds for the dual code.*

The minimum distance of AG codes thus depends on the genus of the curves used to define them. Because the maximal number of points on a curve increases with its genus, there is a tradeoff between the length of a code and its minimum distance.

Construction of a generator matrix of an AG code. Once the parameters of a code have been fixed, including the ordered set P, one just has to specify a basis of $\mathscr{L}(rQ)$, and to form the encoding matrix $M \in \mathscr{M}_{k,n}(\mathbf{F}_{2^m})$ obtained by evaluating this basis on P. A useful basis is one such that the encoding matrix is in systematic form, but it does not necessarily exist for any P. Note however that in the case of MDS codes (such as RS codes) this basis always exists whatever the parameters and the choice of P: this is because in this case every minor of M is of full rank [14]. When such a basis exists, it is easy to find as one just has to start from an arbitrary basis and to compute the reduced row echelon form of the matrix thus obtained.

Example 1: An AG code from an elliptic curve. We give parameters for a code built from the curve defined on $\mathbf{P}^2(\mathbf{F}_{2^4})$ by the homogeneous polynomial $x^2z + xz^2 = y^3 + yz^2$, or equivalently defined on $\mathbf{A}^2(\mathbf{F}_{2^4})$ by $x^2 + x = y^3 + y$. It is of genus 1, and hence it is an elliptic curve. It has 25 points, including one point at infinity, the point $Q = [1:0:0]$; the order of the poles of x and y in Q are respectively 3 and 2. From this, a basis for the space $\mathscr{L}(12Q)$ can easily be obtained. This space has dimension $12+1-g = 12$, and can be used to define a $[24,12,12]_{\mathbf{F}_{2^4}}$ code by evaluation over the affine points of the curve. This allows to define a matrix of dimension 12 over $\mathbf{F}_{2^4}$, which diffuses over $12 \times 4 = 48$ bits and has a differential and linear branch number of 12.

Example 2: An AG code from an hyperelliptic curve. We increase the length of the code by using a curve with a larger genus. We give parameters for a rather well-known code, built from the curve defined on $\mathbf{P}^2(\mathbf{F}_{2^4})$ by the homogeneous polynomial $x^5 = y^2z + yz^4$. This curve has 33 points, including one point at infinity, the point $Q = [0:1:0]$; the order of the poles of x and y in Q are respectively 2 and 5. From this, a basis for the space $\mathscr{L}(17Q)$ can easily be defined. This space has dimension $17+1-g = 16$, and can be used to define a $[32,16,15]_{\mathbf{F}_{2^4}}$ code by evaluation over the affine points of the curve. This code has convenient parameters for defining diffusion matrices: from a generator matrix in systematic form $(I_{16}\ A)$, we can extract the matrix A, which naturally diffuses over 64 bits and has a differential and linear branch number of 15. Furthermore, the code is *self-dual*, which means that A is orthogonal: $A \cdot A^t = I_{16}$. The inverse of A is therefore easy to compute.

We give the right matrix of two matrices of this code in systematic form in appendices D.1 and D.2, the latter further including an example of a basis of $\mathscr{L}(17Q)$ and the order of the points used to construct the matrix.

The problem for the rest of the section is now to find good point orders P for the hyperelliptic code of Ex. 2 such that efficient encoders can be constructed thanks to algorithm 2 of §3.3. For convenience, we name $\mathscr{C}_{\mathscr{HE}}$ any of the codes equivalent to the one of Ex. 2.

4.2 Compact encoders using code automorphisms

We consider matrices in systematic form $(I_{16}\ A)$. For dense matrices, Alg. 2 tends to be most efficient when all the rows of a matrix can be deduced by permutation of one of them, or more generally of a small subset of them. Our objective is thus to find matrices of this form.

The main tool we use to achieve this goal are *automorphisms* of $\mathscr{C}_{\mathscr{HE}}$. Let us first give a definition. (In the following, $\mathfrak{S}_n$ denotes the group of permutations of n elements.)

Definition 5 (Automorphisms of a code) *The automorphism group $Aut(\mathscr{C})$ of a code $\mathscr{C}$ of length n is a subgroup of $\mathfrak{S}_n$ such that $\pi \in Aut(\mathscr{C}) \Rightarrow (c \in \mathscr{C} \Rightarrow \pi(c) \in \mathscr{C})$.*

Because we consider here the code $\mathscr{C}_{\mathscr{HE}}$ which is an evaluation code, we can equivalently define its automorphisms as being permutations of the points on which the evaluation is performed. If π is an automorphism of $\mathscr{C}_{\mathscr{HE}}$, if $\{O_0,\dots,O_l\}$ are its orbits, and if the code is defined with a point order P such that for each orbit all of its points are neighbours in the order P, then the effect of π on a codeword of $\mathscr{C}_{\mathscr{HE}}$ is to cyclically permute its coordinates along each orbit.

To see that this is useful, assume that there is an automorphism π with two orbits O_0 and O_1 of size $n/2$ each. Then, if $M = (I_{n/2}\ A)$ is obtained with point order $P = \langle O_0, O_1\rangle$, each row of M can be obtained by the repeated action of π on, say, $\mathbf{m}^0$, and it follows that A is circulant (and therefore has a low cost w.r.t. algorithm 2). More generally, if an automorphism can be found such that it has orbits of size summing up to $n/2$, the corresponding matrix M can be deduced from a small set of rows. We give two toy examples with Reed-Solomon codes, which can easily be verified.

$\pi:\ \mathbf{F}_{2^4} \to \mathbf{F}_{2^4},\ x \mapsto 8x$. This automorphism has $O_0 = \langle 1,8,12,10,15\rangle$ and $O_1 = \langle 2,3,11,7,13\rangle$ for orbits, among others. The systematic matrix for the $[10,5,6]_{\mathbf{F}_{2^4}}$ code obtained with the points in that order is then such that A is circulant and obtained from the cyclic permutation of the row $(12,10,2,6,3)$.

$\pi:\ \mathbf{F}_{2^4} \to \mathbf{F}_{2^4},\ x \mapsto 7x$. This automorphism has $O_0 = \langle 1,7,6\rangle$, $O_1 = \langle 2,14,12\rangle$, $O_2 = \langle 4,15,11\rangle$, and $O_3 = \langle 8,13,5\rangle$ for orbits, among others. The systematic matrix for the $[12,6,7]_{\mathbf{F}_{2^4}}$ code obtained with the points in that order is then of the form $\begin{pmatrix} I_3 & 0_3 & A & B \\ 0_3 & I_3 & C & D \end{pmatrix}$ with A, B, C and D circulant matrices. It can thus be obtained by cyclic permutation of only two rows.

Application to $\mathscr{C}_{\mathscr{HE}}$. Automorphisms of $\mathscr{C}_{\mathscr{HE}}$ are quite harder to find than ones of RS codes. They can however be found within automorphisms of the curve $\mathscr{X}$ on which it is based [18]. This is quite intuitive, as these will precisely permute points on the curve, which are the points on which the code is defined. We mostly need to be careful to ensure that the point at infinity is fixed by these automorphisms. We considered the degree-one automorphisms of $\mathscr{X}$ described by Duursma [7]. They have two generators: $\pi_0 : \mathbf{F}_{2^4}^2 \to \mathbf{F}_{2^4}^2,\ (x,y) \mapsto (\zeta x, y)$ with $\zeta^5 = 1$, and $\pi_{1_{(a,b)}} : \mathbf{F}_{2^4}^2 \to \mathbf{F}_{2^4}^2,\ (x,y) \mapsto (x+a, y+a^8x^2+a^4x+b^4)$, with (a,b) an affine point of $\mathscr{X}$. These generators span a group of order 160. When considering their orbit decomposition, the break-up of the size of the orbits can only be of one of five types, given in Table 1.
From these automorphisms, it is possible to define a partitions of P in two sets of size 16, which are union of orbits. We may therefore hope to obtain systematic matrices of the type we are looking for. Unfortunately, after an extensive search[5], it appears that ordering P in this fashion *never* results in obtaining a systematic matrix. We recall that indeed, because AG codes are not MDS, it is not always the case that computing the reduced row echelon form of an arbitrary encoding matrix yields a systematic matrix.

[5] Both on $\mathscr{C}_{\mathscr{HE}}$ and on the smaller elliptic code of Ex. 1. However, we are not as yet able to explain this fact.

Table 1. Possible combination of orbit sizes of automorphisms of $\mathscr{C}_{\mathscr{H}\mathscr{E}}$ spanned by π_0 and π_1. A number n in col. c means that an automorphism of this type has n orbits of size c.

Orbit size	1	2	4	5	10
Type 1	32	0	0	0	0
Type 2	0	16	0	0	0
Type 3	0	0	8	0	0
Type 4	2	0	0	6	0
Type 5	0	1	0	0	3

Extending the automorphisms with the Frobenius mapping. We extend the previous automorphisms with the Frobenius mapping $\theta : \mathbf{F}_{2^4}^2 \to \mathbf{F}_{2^4}^2,\ (x,y) \mapsto (x^2, y^2)$; this adds another 160 automorphisms for $\mathscr{X}$. However, these will not anymore be automorphisms for the *code* $\mathscr{C}_{\mathscr{H}\mathscr{E}}$ in general, and we will therefore obtain matrices of a form slightly different from what we first hoped to achieve.

The global strategy is still the same, however, and consists in ordering the points along orbits of the curve automorphisms. By using the Frobenius, new combinations of orbits are possible, notably 4 of size 8. We study below the result of ordering P along the orbits of one such automorphism. We take the example of $\sigma = \theta \circ \sigma_2 \circ \sigma_1$, with $\sigma_1 : (x,y) \mapsto (x+1, y+x^2+x+7)$, $\sigma_2 : (x,y) \mapsto (12x, y)$, and θ the Frobenius mapping. The key observation is that in this case, only σ^0 and σ^4 are automorphisms of $\mathscr{C}_{\mathscr{H}\mathscr{E}}$. Note that not all orbits orderings of σ for P yield a systematic matrix. However, unlike as above, we were able to find some orders that do. In these cases, the right matrix "A" of the full generator matrix $(I_{16}\ A)$ is of the form:

$$(\mathbf{a}^0, \mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3, \sigma^4(\mathbf{a}^0), \sigma^4(\mathbf{a}^1), \sigma^4(\mathbf{a}^2), \sigma^4(\mathbf{a}^3), \mathbf{a}^8, \mathbf{a}^9, \mathbf{a}^{10}, \mathbf{a}^{11}, \sigma^4(\mathbf{a}^8), \sigma^4(\mathbf{a}^9), \sigma^4(\mathbf{a}^{10}), \sigma^4(\mathbf{a}^{11}))^t,$$

with $\mathbf{a}^0, \ldots, \mathbf{a}^3$, $\mathbf{a}^8, \ldots, \mathbf{a}^{11}$ row vectors of dimension 16. For instance, the first and fifth row of one such matrix are:

$$\mathbf{a}^0 = (5,2,1,3,\textit{8, 5, 1, 5},12,10,14,6,\textit{7, 11, 4, 11})$$
$$\mathbf{a}^4 = \sigma^4(\mathbf{a}^0) = (\textit{8, 5, 1, 5},5,2,1,3,\textit{7, 11, 4, 11},12,10,14,6).$$

We give the full matrix in appendix D.1. We have therefore partially reached our goal of being able to describe A from a permutation of a subset of its rows. However this subset $\mathbf{a}^0, \ldots, \mathbf{a}^3$, $\mathbf{a}^8, \ldots, \mathbf{a}^{11}$ is not small, as it is of size 8 —half of the matrix dimension. Consequently, these matrices have a moderate cost according to the *cost2* function, when implemented with algorithm 2, but it is not minimal. Interestingly, all the matrices of this form that we found have the same cost of 52.

4.3 Fast random encoders

We conclude this section by presenting the results of a very simple random search for efficient encoders of $\mathscr{C}_{\mathscr{H}\mathscr{E}}$ with respect to algorithm 2. Unlike the above study, this one does not exploit any kind of algebraic structure. Indeed, the search only consists in repeatedly generating a random permutation of the affine points of the curve, building a matrix for the code with the corresponding point order, tentatively putting it in systematic form $(I_{16}\ A)$, and if successful evaluating the *cost2* function from §3.3 on A. We then collect matrices with a minimum cost.

Because there are $32! \approx 2^{117,7}$ possible point orders, we can only explore a very small part of the search space. However, matrices of low cost can be found even after a moderate amount of computation, and we found many matrices of cost 43, though none of a lower cost. We present in Table 3 from appendix E the number of matrices of cost strictly less than 60 that we found during a search of 2^{38} encoders. We give an example of a matrix of cost 43 in appendix D.2, which is only about a factor 1.5 away from the estimate of the minimum cost of a circulant matrix given in §3.3. We observe that the transpose of this matrix also has a cost of 43.

5 Applications and performance

This last section presents the performance of straightforward assembly implementations of both of our algorithms when applied to a fast encoder of the code $\mathscr{C}_{\mathscr{HE}}$ from §4. More precisely, we consider the diffusion matrix "$M_{\mathscr{H}16}$" of appendix D.2; it is of dimension 16 over $\mathbf{F}_{2^4}$ and has a differential and linear branch number of 15. We do this study in the context of block ciphers, by assuming that $M_{\mathscr{H}16}$ is used as the linear mapping of two ciphers with a SHARK structure: one with 4-bit S-Boxes and a 64-bit block, and one with 8-bit S-Boxes and a 128-bit block. What we wish to measure in both cases is the speed in cycles per byte of such hypothetical ciphers, so as to be able to gauge the efficiency of this linear mapping and of the resulting ciphers. In order to do this, we need to estimate how many rounds would be needed for the ciphers to be secure.

Basic statistical properties of a 64-bit block cipher with 4-bit S-Boxes and $M_{\mathscr{H}16}$ as a linear mapping. We use standard *wide-trail* considerations to study differential and linear properties of this cipher [6]. This is very easy to do thanks to the simple structure of the cipher. The branch number of $M_{\mathscr{H}16}$ is 15, which means that at least 15 S-boxes are active in any two rounds of a differential path or linear characteristic. The best 4-bit S-boxes have a maximum differential probability and a maximal linear bias of 2^{-2} (see *e.g.* [16,13]). By using such S-boxes, one can upper-bound the probability of a single differential path or linear characteristic for $2n$ rounds by $2^{-2\cdot 15n}$. This is smaller than 2^{-64} as soon as $n > 2$. Hence we conjecture that 6 to 8 rounds are enough to make a cipher resistant to standard statistical attacks.

If one were to propose a concrete cipher, a more detailed analysis would of course be needed, especially w.r.t. more dedicated structural attacks. However, it seems reasonable to consider at a first glance that 8 rounds would indeed be enough to bring adequate security. It is only 2 rounds less than AES-128, which uses a round with comparatively weaker diffusion. Also, 6 rounds might be enough. Consequently, we present software performance figures for 6 and 8 rounds of such an hypothetical 64-bit cipher with 4-bit S-Boxes in Table 2, on the left. We include data both for a strict SSSE3 implementation and for one using AVX extensions, which can be seen to bring a considerable benefit. Note that the last round is complete and includes the linear mapping, unlike *e.g.* AES. Also, note that the parallel application of the S-Boxes can be implemented very efficiently with a single `pshufb`, and thus has virtually no impact on the speed.

Basic statistical properties of a 128-bit block cipher with 8-bit S-Boxes and $M_{\mathscr{H}16}$ as a linear mapping. Although the code $\mathscr{C}_{\mathscr{HE}}$ from which the matrix $M_{\mathscr{H}16}$ is built was initially defined with $\mathbf{F}_{2^4}$ as an alphabet, this latter can be replaced by an algebraic extension of $\mathbf{F}_{2^4}$ such as $\mathbf{F}_{2^8}$, to yield a code $\mathscr{C}_{\mathscr{HE}}{}'$ with the same parameters, namely a $[32,16,15]_{\mathbf{F}_{2^8}}$ code. Indeed, by using a suitable representation such as $\mathbf{F}_{2^4} \cong \mathbf{F}_2[\alpha]/(\alpha^4+\alpha^3+\alpha^2+\alpha+1)$; $\mathbf{F}_{2^8} \cong \mathbf{F}_{2^4}[t]/(t^2+t+\alpha)$[6], an element of $\mathbf{F}_{2^8}$ is represented as a degree-one polynomial $at+b$ over $\mathbf{F}_{2^4}$. It follows that the minimum weight of a codeword $w = (a_i t + b_i)$, $i \in \{0\ldots31\}$ of $\mathscr{C}_{\mathscr{HE}}{}'$ is at least equal to the minimum weight of words (a_i), $i = 0\ldots31$ and (b_i), $i \in \{0\ldots31\}$. If those are taken among codewords of $\mathscr{C}_{\mathscr{HE}}$, their minimum weight is 15, and thus so is the one of w. It is also possible to efficiently compute the multiplication by $M_{\mathscr{H}16}$ over $\mathbf{F}_{2^8}$ from two computations over $\mathbf{F}_{2^4}$: because the coefficients of $M_{\mathscr{H}16}$ are in $\mathbf{F}_{2^4}$, we have $M_{\mathscr{H}16}\cdot(a_i t + b_i) = (M_{\mathscr{H}16}\cdot(a_i)\cdot t) + (M_{\mathscr{H}16}\cdot(b_i))$. As a result, applying $M_{\mathscr{H}16}$ to $\mathbf{F}_{2^8}$ has only twice the cost of applying it to $\mathbf{F}_{2^4}$, while effectively doubling the size of the block.

From wide-trail considerations, the resistance of such a cipher to statistical attacks is comparatively even better than when using 4-bit S-Boxes, when an appropriate 8-bit S-Box is used. For instance, the AES S-box has a maximal differential probability and linear bias of 2^{-6} [6]. This implies that the probability of a single differential path or linear characteristic for $2n$ rounds is upper-bounded by $2^{-6\cdot 15n}$, which is already much smaller than 2^{-128} as soon as $n > 1$. Again, 8 rounds of such a cipher should bring adequate security, and 6 rounds might be enough. We provide performance figures for both an SSSE3 and an AVX implementation in Table 2, on the right. However, in

[6] This representation is used by Hamburg in [9].

the case of 8-bit S-Boxes, the S-Box application is rather more complex and expensive a step than with 4-bit S-Boxes. In these test programs, we decided to use the efficient vector implementation of the AES S-Box from Hamburg [9].

Table 2. Performance of software implementations of the hypothetical 64 and 128-bit cipher, in cycles per byte (cpb). Figures in parentheses are for an AVX implementation (when applicable).

		64-bit Block		128-bit Block	
Processor type	# rounds	cpb (Alg. 1)	cpb (Alg. 2)	cpb (Alg. 1)	cpb (Alg. 2)
Intel Xeon E5-2650 @ 2.00GHz	6	50 (45.5)	33 (24.2)	58 (52.3)	32.7 (26.5)
	8	66.5 (60.2)	44.5 (31.9)	76.8 (69.6)	43.8 (35.7)
Intel Xeon E5-2609 @ 2.40GHz	6	72.3 (63.7)	45.3 (33.2)	79.8 (75.6)	47.1 (36.8)
	8	95.3 (84.7)	63.3 (45.6)	106.6 (97.1)	62.1 (50.3)
Intel Xeon E5649 @ 2.53GHz	6	84.7	46	84.5	47
	8	111.3	59.8	111	61.9

Discussion. The performance figures given in Table 2 are average for a block cipher. For instance, it compares favourably with the optimized vector implementations of 64-bit ciphers LED and Piccolo in sequential mode from [3], which run at speeds between 70 and 90 cpb., depending on the CPU. It is however slower than Hamburg's vector implementation of AES, with reported speeds of 6 to 22 cpb. (9 to 25 for the inverse cipher) [9,19].

6 Conclusion

We revisited the SHARK structure by replacing the MDS matrix of its linear diffusion layer by a matrix built from an algebraic-geometry code. Although this code is not MDS, it has a very high minimum distance, while being defined over $\mathbf{F}_{2^4}$ instead of $\mathbf{F}_{2^8}$. This allows to reduce the size of the coefficients of the matrix from 8 to 4 bits, and has important consequences for efficient implementations of this linear mapping. We studied algorithms suitable for a vector implementation of the multiplication by this matrix, and how to find matrices that are most efficiently implemented with those algorithms. Finally, we gave performance figures for assembly implementations of hypothetical SHARK-like ciphers using this matrix as a linear layer.

This work provided the first generalizations of SHARK that are not vulnerable to timing attacks as is the original cipher, and also the first generalization to 128-bit blocks. It also showed that even if not the fastest, such potential design could be implemented efficiently in software.

As a future work, it would be interesting to investigate how to use the full automorphism group of the code to design matrices with a lower cost. In that case, we would not restrict ourselves to derive the rows from a single row and the powers of a *single* automorphism, but could use several independent automorphisms instead.

Acknowledgments

Pierre Karpman is partially supported by the Direction générale de l'armement and by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Augot, D., Finiasz, M.: Direct Construction of Recursive MDS Diffusion Layers using Shortened BCH Codes. In: FSE. Lecture Notes in Computer Science (to appear), Springer. Available at `https://www.rocq.inria.fr/secret/Matthieu.Finiasz/research/2014/augot-finiasz-fse14.pdf`.
2. Augot, D., Finiasz, M.: Exhaustive Search for Small Dimension Recursive MDS Diffusion Layers for Block Ciphers and Hash Functions. In: ISIT, IEEE (2013) 1551–1555
3. Benadjila, R., Guo, J., Lomné, V., Peyrin, T.: Implementing lightweight block ciphers on x86 architectures. In Lange, T., Lauter, K., Lisonek, P., eds.: Selected Areas in Cryptography. Volume 8282 of Lecture Notes in Computer Science., Springer (2013) 324–351
4. Bernstein, D.J., Schwabe, P.: NEON Crypto. In Prouff, E., Schaumont, P., eds.: CHES. Volume 7428 of Lecture Notes in Computer Science., Springer (2012) 320–339
5. Chen, J.P., Lu, C.C.: A Serial-In-Serial-Out Hardware Architecture for Systematic Encoding of Hermitian Codes via Gröbner Bases. IEEE Transactions on Communications **52**(8) (2004) 1322–1332
6. Daemen, J., Rijmen, V.: The Design of Rijndael: AES — The Advanced Encryption Standard. Information Security and Cryptography. Springer (2002)
7. Duursma, I.: Weight distributions of geometric Goppa codes. Transactions of the American Mathematical Society **351**(9) (1999) 3609–3639
8. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON Family of Lightweight Hash Functions. In Rogaway, P., ed.: CRYPTO. Volume 6841 of Lecture Notes in Computer Science., Springer (2011) 222–239
9. Hamburg, M.: Accelerating AES with Vector Permute Instructions. In Clavier, C., Gaj, K., eds.: CHES. Volume 5747 of Lecture Notes in Computer Science., Springer (2009) 18–32
10. Heegard, C., Little, J., Saints, K.: Systematic Encoding via Gröbner Bases for a Class of Algebraic-Geometric Goppa Codes. IEEE Transactions on Information Theory **41**(6) (1995) 1752–1761
11. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual. (March 2012)
12. Knudsen, L.R., Wu, H., eds.: Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers. In Knudsen, L.R., Wu, H., eds.: Selected Areas in Cryptography. Volume 7707 of Lecture Notes in Computer Science., Springer (2013)
13. Leander, G., Poschmann, A.: On the Classification of 4 Bit S-Boxes. In Carlet, C., Sunar, B., eds.: WAIFI. Volume 4547 of Lecture Notes in Computer Science., Springer (2007) 159–176
14. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland Mathematical Library. North-Holland (1978)
15. Rijmen, V., Daemen, J., Preneel, B., Bosselaers, A., Win, E.D.: The Cipher SHARK. In Gollmann, D., ed.: FSE. Volume 1039 of Lecture Notes in Computer Science., Springer (1996) 99–111
16. Saarinen, M.J.O.: Cryptographic Analysis of All 4×4-Bit S-Boxes. In Miri, A., Vaudenay, S., eds.: Selected Areas in Cryptography. Volume 7118 of Lecture Notes in Computer Science., Springer (2011) 118–133
17. Sajadieh, M., Dakhilalian, M., Mala, H., Sepehrdad, P.: Recursive Diffusion Layers for Block Ciphers and Hash Functions. In Canteaut, A., ed.: FSE. Volume 7549 of Lecture Notes in Computer Science., Springer (2012) 385–401
18. Stichtenoth, H.: Algebraic Function Fields and Codes. 2 edn. Volume 254 of Graduate Texts in Mathematics. Springer (2009)
19. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE: A Lightweight Block Cipher for Multiple Platforms. [12] 339–354
20. Tromer, E., Osvik, D.A., Shamir, A.: Efficient Cache Attacks on AES, and Countermeasures. J. Cryptology **23**(1) (2010) 37–71
21. Van Lint, J.H.: Introduction to Coding Theory. 3 edn. Volume 86 of Graduate Texts in Mathematics. Springer (1999)
22. Wu, S., Wang, M., Wu, W.: Recursive Diffusion Layers for (Lightweight) Block Ciphers and Hash Functions. [12] 355–371

A Algorithms for matrix-vector multiplication

A.1 Broadcast-based algorithm

The broadcast-based algorithm for matrix-vector multiplication of §3.2 is given in Algorithm 1, where"$\wedge_{64}$" denotes the bitwise *logical and* on 64-bit values.

Algorithm 1 Broadcast-based matrix-vector multiplication

Input: $\mathbf{x} \in \mathbf{F}_{2^4}^{16}$, $M \in \mathscr{M}_{16}(\mathbf{F}_{2^4})$
Output: $\mathbf{y} = \mathbf{x}^t \cdot M^t$

Offline phase
for $i \in \{0 \dots 15\}$ **do**
  $8\mathbf{m}^i \leftarrow \alpha^3 \cdot \mathbf{m}^i$
  $4\mathbf{m}^i \leftarrow \alpha^2 \cdot \mathbf{m}^i$
  $2\mathbf{m}^i \leftarrow \alpha \cdot \mathbf{m}^i$
end for
Online phase
$\mathbf{y} \leftarrow 0_{64}$
for $i \in \{0 \dots 15\}$ **do**
  $\gamma_i^8 \leftarrow 8\mathbf{m}^i \wedge_{64} broadcast(\mathbf{x}_i, 3)_{64}$
  $\gamma_i^4 \leftarrow 4\mathbf{m}^i \wedge_{64} broadcast(\mathbf{x}_i, 2)_{64}$
  $\gamma_i^2 \leftarrow 2\mathbf{m}^i \wedge_{64} broadcast(\mathbf{x}_i, 1)_{64}$
  $\gamma_i^1 \leftarrow \mathbf{m}^i \wedge_{64} broadcast(\mathbf{x}_i, 0)_{64}$
  $\gamma_i \leftarrow \gamma_i^8 + \gamma_i^4 + \gamma_i^2 + \gamma_i^1$
  $\mathbf{y} \leftarrow \mathbf{y} + \gamma_i$
end for
return $\mathbf{y}$

A.2 Shuffle-based algorithm

The shuffle-based algorithm for matrix-vector multiplication of §3.3 is given in Algorithm 2.

Algorithm 2 Shuffle-based matrix-vector multiplication

Input: $\mathbf{x} \in \mathbf{F}_{2^4}^{16}$, $M \in \mathscr{M}_{16}(\mathbf{F}_{2^4})$
Output: $\mathbf{y} = M \cdot \mathbf{x}$

Offline phase
for $i \in \{1 \dots 15\}$ **do**
  $s_i[\,] \leftarrow \mathscr{S}(M, i)$ ▷ *Initialize the array s_i with one of the possible sets of shuffles $\mathscr{S}(M, i)$*
end for
Online phase
$\mathbf{y} \leftarrow 0_{64}$
for $i \in \{1 \dots 15\}$ **do**
  for $j < \#s_i$ **do**
    $\mathbf{y} \leftarrow \mathbf{y} + i \cdot s_i[j]$
  end for
end for
return $\mathbf{y}$

B Excerpts of assembly implementations of matrix-vector multiplication

B.1 Excerpt of an implementation of algorithm 1

```
; cleaning mask
cle: dq 0x0f0f0f0f0f0f0f0f, 0x0f0f0f0f0f0f0f0f
; mask for the selection of v, 2v
oe1: dq 0xff0ff000ff0ff000, 0xff0ff000ff0ff000
; mask for the selection of 4v, 8v
oe2: dq 0xf0f0f0f000000000, 0xffffffff0f0f0f0f
; m0 and 2m0 interleaved
c01: dq 0x91a7efa76c126cb5, 0x9124fd91cb3636d9
; 4m0 and 8m0 interleaved
c02: dq 0x24efd9efb548b5a7, 0x248391245acbcb12
; etc.
c11: dq 0x249183244800cb6c, 0x6cfd12485a91b55a
```

```
c12: dq 0x83246c8336005ab5, 0xb59148367e24a77e
c21: dq 0xfdd9b5122424b591, 0xa73612ef122436d9
c22: dq 0x9112a7488383a724, 0xefcb48d94883cb12
; [...]

; macro for one matrix row multiplication and accumulation
; (nasm syntax)

; 1 is input
; 2, 3, 4, 5 are storage
; 6 is constant zero
; 7 is accumulator
; 8 is index
%macro m_1_row 8
  ; selects the right double-masks
  movdqa %2, [oe1]
  movdqa %3, [oe2]
  pshufb %2, %1
  pshufb %3, %1
  ; shift the input for the next round
  psrldq %1, 1
  ; expand
  pshufb %2, %6
  pshufb %3, %6
  ; select the rows with the double-masks
  pand   %2, [c01 + %8*16*2]
  pand   %3, [c02 + %8*16*2]
  ; shift and xor the rows together
  movdqa %4, %2
  movdqa %5, %3
  psrlq  %2, 4
  psrlq  %3, 4
  pxor   %4, %5
  pxor   %2, %3
  ; accumulate everything
  pxor   %2, %4
  pxor   %7, %2
%endmacro

; the input is in xmm0, with only the four lsb of each byte set
_m64:
  ; constant zero
  pxor xmm5, xmm5
  ; accumulator
  pxor xmm6, xmm6
.mainstuff:
  m_1_row xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, 0
  m_1_row xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, 1
  m_1_row xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, 2
  ; [...]
.fin:
  ; the result is in xmm0, and still of the same form
  pand xmm6, [cle]
  movdqa xmm0, xmm6
  ret
```

B.2 Excerpt of an implementation of algorithm 2

```
; multiplication tables (in the order where we need them)
ttim2 : dq 0x0e0c0a0806040200, 0x0d0f090b05070103
ttim8 : dq 0x0d050e060b030800, 0x0109020a070f040c
; [...]
```

```
; shuffles
; 0xff for not selected nibbles (it will zero them)
pp10: dq 0x0005000507040d02, 0x0f01040a03060809
pp11: dq 0xffff0e0c0e0bffff, 0xff0b08ff040cffff
pp12: dq 0xffffff0eff0dffff, 0xffffffffffffffff

pp20: dq 0x050306070602040e, 0x0408010c01000b04
pp21: dq 0x0c0e0909ff0307ff, 0x0bff020efffff0dff
pp22: dq 0xffffff0fff0affff, 0xffffffffffffffff

pp30: dq 0xffffff02ff09ff09, 0x0afff04ffffff02
pp31: dq 0xffffff04ff0eff0a, 0x0efff0fffffff0f
; [...]

; 2 useful macros (nasm syntax)

; 1 is the location of the vector,
; 2 is storage for the multiplied thingy,
; 3 is the constant index
%macro vec_mul 3
  movdqa %2, [ttim2 + %3*16]
  pshufb %2, %1
  movdqa %1, %2
%endmacro

; 1 is the accumulator,
; 2 is the multiplied vector,
; 3 is storage for the shuffled thingy,
; 4 is the index for the shuffle
%macro pp_accu 4
  movdqa %3, %2
  pshufb %3, [pp10 + %4*16]
  pxor   %1, %3
%endmacro

; the input is in xmm0, with only the four lsb of each byte set
_m64:
  pxor xmm1, xmm1 ; init the accumulator
.mainstuff:
  ; 1.x
  pp_accu xmm1, xmm0, xmm2, 0
  pp_accu xmm1, xmm0, xmm2, 1
  pp_accu xmm1, xmm0, xmm2, 2
  ; 2.x
  vec_mul xmm0, xmm2, 0
  pp_accu xmm1, xmm0, xmm2, 3
  pp_accu xmm1, xmm0, xmm2, 4
  pp_accu xmm1, xmm0, xmm2, 5
  ; 3.x
  vec_mul xmm0, xmm2, 1
  pp_accu xmm1, xmm0, xmm2, 6
  pp_accu xmm1, xmm0, xmm2, 7
; [...]
.fin:
  ; the result is in xmm0, and still of the same form
  movdqa xmm0, xmm1
  ret
```

C A short introduction to Reed-Solomon codes

Definition 6 (Reed-Solomon codes) *Let $\mathscr{C}_{RS}$ be an $[n,k,d]_{\mathbf{F}_{2^m}}$ Reed-Solomon code of length n, dimension k, and minimal distance d with symbols in $\mathbf{F}_{2^m}$. The codeword $\mathscr{C}_{RS}(\mathbf{m})$ corresponding to the*

message $\mathbf{m}$ *of dimension* k *is defined as the vector* $(\mathbf{m}(p_i), i = 0 \dots n-1)$*, where* $\mathbf{m}$ *is seen as the polynomial* $\sum_{i=0\dots k-1} \mathbf{m}_i \cdot x^i$ *of* $\mathbf{F}_{2^m}[x]$ *of degree at most* $k-1$*, and where* $P = \langle p_0, \dots, p_{n-1} \rangle$ *is an ordered set of* n *elements of* $\mathbf{F}_{2^m}$*,* $n \leq 2^m$*. The code is the set of all such codewords.*

These codes have the following properties: for fixed parameters n and k, there are $\binom{2^m}{n} \cdot n!$ *equivalent* codes, which corresponds to the number of possible ordered sets P; the minimal distance d of a code of parameters n and k is at least $n-(k-1) = n-k+1$. This is because a polynomial of degree $k-1$ has at most $k-1$ zeros, and hence any codeword except the all-zero word is non-zero on at least $n-(k-1)$ positions. Because this bound reaches the Singleton bound, RS codes are MDS codes; finally, it is obvious that the maximal length of an RS code over $\mathbf{F}_{2^m}$ is 2^m.

D Examples of diffusion matrices of dimension 16 over $\mathbf{F}_{2^4}$

D.1 A matrix of cost 52

The following matrix for the diffusion part of $\mathscr{C}_{\mathscr{HE}}$ was found thanks to the automorphisms from §4.2, based on the Frobenius mapping. It has cost 52, and both a differential and a linear branch number of 15.

$$\begin{pmatrix}
5 & 2 & 1 & 3 & 8 & 5 & 1 & 5 & 12 & 10 & 14 & 6 & 7 & 11 & 4 & 11 \\
2 & 2 & 4 & 1 & 5 & 12 & 2 & 1 & 9 & 15 & 8 & 11 & 7 & 6 & 9 & 3 \\
1 & 4 & 4 & 3 & 1 & 2 & 15 & 4 & 5 & 13 & 10 & 12 & 9 & 6 & 7 & 13 \\
3 & 1 & 3 & 3 & 5 & 1 & 4 & 10 & 14 & 2 & 14 & 8 & 15 & 13 & 7 & 6 \\
8 & 5 & 1 & 5 & 5 & 2 & 1 & 3 & 7 & 11 & 4 & 11 & 12 & 10 & 14 & 6 \\
5 & 12 & 2 & 1 & 2 & 2 & 4 & 1 & 7 & 6 & 9 & 3 & 9 & 15 & 8 & 11 \\
1 & 2 & 15 & 4 & 1 & 4 & 4 & 3 & 9 & 6 & 7 & 13 & 5 & 13 & 10 & 12 \\
5 & 1 & 4 & 10 & 3 & 1 & 3 & 3 & 15 & 13 & 7 & 6 & 14 & 2 & 14 & 8 \\
12 & 9 & 5 & 14 & 7 & 7 & 9 & 15 & 7 & 6 & 11 & 3 & 15 & 5 & 13 & 7 \\
10 & 15 & 13 & 2 & 11 & 6 & 6 & 13 & 6 & 6 & 7 & 9 & 5 & 10 & 2 & 14 \\
14 & 8 & 10 & 14 & 4 & 9 & 7 & 7 & 11 & 7 & 7 & 6 & 13 & 2 & 8 & 4 \\
6 & 11 & 12 & 8 & 11 & 3 & 13 & 6 & 3 & 9 & 6 & 6 & 7 & 14 & 4 & 12 \\
7 & 7 & 9 & 15 & 12 & 9 & 5 & 14 & 15 & 5 & 13 & 7 & 7 & 6 & 11 & 3 \\
11 & 6 & 6 & 13 & 10 & 15 & 13 & 2 & 5 & 10 & 2 & 14 & 6 & 6 & 7 & 9 \\
4 & 9 & 7 & 7 & 14 & 8 & 10 & 14 & 13 & 2 & 8 & 4 & 11 & 7 & 7 & 6 \\
11 & 3 & 13 & 6 & 6 & 11 & 12 & 8 & 7 & 14 & 4 & 12 & 3 & 9 & 6 & 6
\end{pmatrix}$$

D.2 A matrix of cost 43

The following matrix, for the same code, was found by randomly testing permutations of the points of the curve. It has cost 43, and so does its transpose.

$$\begin{pmatrix}
11 & 6 & 1 & 6 & 10 & 14 & 10 & 9 & 13 & 3 & 3 & 12 & 9 & 15 & 2 & 9 \\
6 & 12 & 0 & 4 & 2 & 8 & 9 & 2 & 5 & 11 & 9 & 5 & 4 & 1 & 15 & 6 \\
9 & 11 & 2 & 2 & 1 & 11 & 13 & 15 & 13 & 3 & 2 & 1 & 14 & 1 & 3 & 10 \\
0 & 0 & 9 & 8 & 11 & 6 & 2 & 1 & 11 & 10 & 15 & 10 & 10 & 15 & 1 & 14 \\
13 & 13 & 3 & 15 & 3 & 1 & 11 & 2 & 9 & 2 & 10 & 14 & 1 & 11 & 1 & 2 \\
1 & 9 & 8 & 4 & 14 & 10 & 2 & 5 & 15 & 2 & 12 & 12 & 9 & 10 & 1 & 9 \\
5 & 9 & 11 & 2 & 15 & 1 & 12 & 4 & 6 & 0 & 6 & 4 & 5 & 8 & 2 & 9 \\
1 & 4 & 14 & 9 & 13 & 2 & 10 & 12 & 0 & 6 & 6 & 9 & 2 & 0 & 11 & 10 \\
13 & 10 & 3 & 9 & 2 & 15 & 6 & 6 & 11 & 1 & 9 & 9 & 12 & 14 & 10 & 3 \\
0 & 10 & 6 & 12 & 11 & 0 & 4 & 9 & 1 & 14 & 10 & 2 & 9 & 2 & 13 & 6 \\
2 & 0 & 5 & 6 & 9 & 0 & 1 & 5 & 15 & 12 & 13 & 15 & 1 & 11 & 13 & 11 \\
11 & 2 & 10 & 1 & 1 & 15 & 0 & 8 & 0 & 9 & 14 & 10 & 10 & 6 & 11 & 15 \\
12 & 14 & 10 & 11 & 3 & 10 & 6 & 0 & 5 & 11 & 1 & 8 & 2 & 9 & 2 & 3 \\
15 & 2 & 2 & 5 & 1 & 10 & 9 & 4 & 1 & 8 & 9 & 9 & 12 & 10 & 14 & 12 \\
15 & 1 & 12 & 5 & 13 & 11 & 0 & 6 & 2 & 5 & 11 & 1 & 15 & 0 & 9 & 13 \\
5 & 6 & 11 & 0 & 2 & 9 & 14 & 11 & 12 & 10 & 3 & 2 & 8 & 10 & 3 & 1
\end{pmatrix}$$

It can easily be obtained as follows. First, define a basis of $\mathscr{L}(17Q)$, for instance: $(1, x, x^2, y, x^3, xy, x^4, x^2y, x^5, x^3y, x^6, x^4y, x^7, x^5y, x^8, x^6y)$. Then, arrange the affine points of the curve on which $\mathscr{C}_{\mathscr{HE}}$ is defined in the order[7]: ⟨ (8, 7), (13, 2), (4, 5), (0, 0), (14, 5), (15, 6), (7, 3), (1, 7), (2, 2), (11, 3), (3, 3), (10, 7), (6, 4), (5, 5), (9, 5), (12, 6), (3, 2), (11, 2), (12, 7), (13, 3), (4, 4), (0, 1), (1, 6), (7, 2), (9, 4), (6, 5), (2, 3), (5, 4), (15, 7), (10, 6), (14, 4), (8, 6) ⟩. Finally, evaluate the basis of $\mathscr{L}(17Q)$ on these points (this defines a matrix of 16 rows and 32 columns), and compute the reduced row echelon form of this matrix; the right square matrix of dimension 16 of the result is the matrix above.

E Statistical distribution of the cost of matrices of $\mathscr{C}_{\mathscr{HE}}$

Table 3. Statistical distribution of the cost of 2^{38} randomly-generated generator matrices of $\mathscr{C}_{\mathscr{HE}}$.

cost	#matrices	cumulative #matrices	cumulative proportion of the search space
43	146 482	146 482	0.00000053
44	73 220	219 702	0.00000080
45	218 542	438 244	0.0000016
46	879 557	1 317 801	0.0000048
47	1 978 159	3 295 960	0.000012
48	5 559 814	8 855 774	0.000032
49	21 512 707	30 368 481	0.00011
50	93 289 020	123 657 501	0.00045
51	356 848 829	480 506 330	0.0017
52	1 282 233 658	1 762 739 988	0.0064
53	3 534 412 567	5 297 152 555	0.019
54	8 141 274 412	13 438 426 967	0.049
55	15 433 896 914	28 872 323 881	0.11
56	24 837 735 898	53 710 059 779	0.20
57	33 794 051 687	87 504 111 466	0.32
58	38 971 338 149	126 475 449 615	0.46
59	38 629 339 524	165 104 789 139	0.60

[7] We omit the z coordinate for conciseness, as it is always 1.