Universiteit
Leiden
The Netherlands

FFCSA - Finite Field Constructions, Search, and Algorithms

Zidaric, N.; Gong, G.; Aagaard, M.; Jurisic, A.; Konovalov, O.

Citation
Zidaric, N., Gong, G., Aagaard, M., Jurisic, A., & Konovalov, O. (2023). FFCSA - Finite Field Constructions, Search, and Algorithms. *Acm Communications In Computer Algebra*, *57*(2), 57-64. doi:10.1145/3614408.3614416

Version:	Publisher's Version
License:	Licensed under Article 25fa Copyright Act/Law (Amendment Taverne)
Downloaded from:	https://hdl.handle.net/1887/3718467

Note: To cite this publication please use the final published version (if applicable).

FFCSA – Finite Field Constructions, Search, and Algorithms

Nuša Zidarič[1]*, Guang Gong[2], Mark Aagaard[2], Aleksandar Jurišić[3], Olexandr Konovalov[4]

[1] Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, The Netherlands
[2] Electrical and Computer Engineering, University of Waterloo Waterloo, Canada
[3] Faculty of Computer and Information Science, University of Ljubljana Ljubljana, Slovenija
[4] School of Computer Science, University of St Andrews, St Andrews, Scotland

Abstract

In this work we present the new GAP package FFCSA – Finite Field Constructions, Search, and Algorithms. It was designed to enable Design Space Exploration for hardware implementations of cryptographic algorithms defined over finite fields. FFCSA constructions and searches are used to produce the design space, and FFCSA algorithms, parameterized for the current candidate field, are used to generate expressions needed for implementation in hardware.

CCS Concepts: Computing methodologies → Symbolic and algebraic manipulation.
Keywords: FFCSA, GAP, FSR, finite field arithmetic, cryptographic hardware

1 Introduction

In this work we present the new GAP package FFCSA – Finite Field Constructions, Search, and Algorithms. It was designed to enable Design Space Exploration (DSE) for hardware implementations of cryptographic algorithms defined over finite fields. With the rise of new technologies, such as Internet of Things, Cyber-Physical Systems, automotive, to name just a few, microchips with communication capabilities are extremely common. Furthermore, they have diverse hardware implementation requirements, and DSE is mandatory to select suitable parameters and trade-offs. FFCSA is a part of a larger framework [15, 19], which includes packages for automated generation of hardware modules; the entire framework is beyond the scope of this paper. To build such a framework we need a Computer Algebra System for symbolic computation that supports finite fields; we chose the open-source system GAP [5]. Our approach opens numerous possibilities for automated generation and optimization of hardware guided by the mathematical properties, and to change mathematical properties of cryptographic algorithms based on their hardware cost and performance.

2 Background

2.1 GAP

GAP [5] is an open source system for discrete computational algebra. It was selected as a platform for the implementation due to multiple reasons, in particular, for its excellent support for finite fields,

*Corresponding author: Nuša Zidarič n.zidaric@liacs.leidenuniv.nl . The majority of this work was done when Nuša Zidarič was affiliated with the University of Waterloo. She is currently affiliated with LIACS, Leiden University.

which includes their constructions, arithmetic operations, factorization of polynomials, irreducibility check, conjugates, decomposition of elements w.r.t. a given basis, etc.. GAP also has a well-defined mechanism of extending it with packages, which are managed by their authors and maintainers, and may be submitted for the redistribution with GAP. One of such packages is the JupyterKernel package [9] which adds Jupyter support to GAP and allows to make interactive and engaging demonstrations. Finally, GAP and a number of its packages are included in the SageMath [14].

FFCSA package [16] requires FSR package [18], designed for cryptographic modules with filtering structures [17], such as the WG cipher [8]. FFCSA package adds support for tower fields to the hardware design flow in [17]. FSR and FFCSA were used during design stage of authenticated encryption scheme WAGE [1, 2, 15].

2.2 Preliminaries

Let $\mathbb{F}_q$ be a binary extension field with q elements, where $q = 2^m$ and $m \geq 1$. We consider multivariate polynomial functions $f : \mathbb{F}_q^t \to \mathbb{F}_q$ in $t \geq 1$ variables $x_0, \ldots, x_{t-1}$ with their corresponding integer exponents i_j $(0 \leq j \leq t-1)$ and coefficients $\gamma_{i_0,\ldots,i_{t-1}} \in \mathbb{F}_q$:

$$f(x_0, \ldots, x_{t-1}) = \sum_{(i_0,\ldots,i_{t-1}) \in \mathbb{Z}_{q-1}^t} \gamma_{i_0,\ldots,i_{t-1}} x_0^{i_0} \ldots x_{t-1}^{i_{t-1}}. \tag{1}$$

The exponents i_j are reduced modulo $q-1$ by the generalization of the Fermats little theorem [7] (i.e., $x^q = x \ \forall x \in \mathbb{F}_q$) and hence the t-tuple $(i_0, \ldots, i_{t-1}) \in \mathbb{Z}_{q-1}^t$ uniquely describes each monomial. For the remainder of this text, the term expression is of the form given in Equation (1).

Let $\mathcal{K}$ be a finite field and $\mathcal{F}$ its extension of degree m, i.e., $\mathcal{F}/\mathcal{K}$ and $m = [\mathcal{F} : \mathcal{K}]$. Let $B_{\mathcal{F}/\mathcal{K}} = \{\rho^{(0)}, \rho^{(1)}, \ldots, \rho^{(m-1)}\}$, where $\rho \in \mathcal{F}$, be an arbitrary basis of $\mathcal{F}/\mathcal{K}$. The following are the representation of $A \in \mathcal{F}$ w.r.t. $B_{\mathcal{F}/\mathcal{K}}$, its vector form and the notation for the i-th coordinate of A, where $a_i \in \mathcal{K}$ $(0 \leq i \leq m-1)$:

$$A = \sum_{i=0}^{m-1} a_i \rho^{(i)}, \qquad [A]_{B_{\mathcal{F}/\mathcal{K}}} = [a_0, a_1, \ldots, a_{m-1}], \qquad [A]_{(i)} = a_i.$$

We use the term "field instance" to refer to a particular finite field and its basis, i.e., one of the many candidates for the DSE. We will use ffe to abbreviate a finite field element.

3 Basic structure of FFCSA

Different algorithms for finite field arithmetic that have been developed over the years will be simply referred to as "algorithms" (the letter "A" in "FFCSA", Section 3.1). To list a few examples: classic two-step multiplication, Massey-Omura multiplication, Itoh-Tsuji inversion, ... Some algorithms were optimized for software applications, and will not necessarily perform well in hardware. An example of an algorithm optimized for hardware implementations is the reduced redundancy Massey-Omura parallel multiplier [11].

For DSE we need to generate many field instances, and for hardware design automation we need to extract submodules and generate expressions for their implementation. From the perspective of DSE, a candidate field instance is the finite field and its basis; FFCSA construction and search methods were designed to find the candidates (letters "CS" in "FFCSA", Section 3.2).

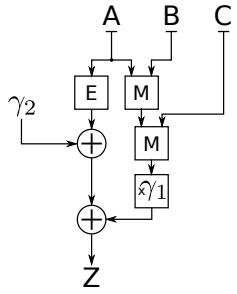
To define $\mathbb{F}_{q^m}$ it is enough to construct the basis $B_{\mathbb{F}_{q^m}/\mathbb{F}_q}$ [7]. We introduce a notion of *direction* for the basis as `To` and `Downto`. This is adopted from a hardware description language called VHDL: its signals are specified by their *domain*, *range* and *direction*. In order to merge the worlds of digital hardware design and finite fields, we map the domain to the subfield and the range to the degree of extension $\mathcal{F}/\mathcal{K}$, and generate our bases according to the desired direction: `To` for `(0 to m-1)` or `Downto` for `(m-1 downto 0)`.

3.1 Finite field arithmetic algorithms

In this section we explain how we use symbolic computation to generate datapaths. We use a simple expression over $\mathbb{F}_q$, where $q = 2^m$, $m > 1$, as an example:

$$f(x_0, x_1, x_2) = \gamma_1 x_0 \cdot x_1 \cdot x_2 + x_0^2 + \gamma_2. \tag{2}$$

Expression (2) has variables $x_0, x_1, x_2 \in \mathbb{F}_q$ and two arbitrary, not necessary distinct, constants $\gamma_1, \gamma_2 \in \mathbb{F}_q^*$.



map GAP variables to input ports:
$x_0 \to$A, $x_1 \to$B, $x_2 \to$C
map function f to the output port:
$f \to$Z
Submodules: squarer E, two multipliers M,
and constant multiplier $\times\gamma_1$.
Addition modulo 2: XOR gates,
also used for the additive constant $+\gamma_1$

Figure 1: Circuit schematics for the example in Equation (2).

Figure 1 shows submodules for arithmetic operations needed to implement expression (2) as top-level hardware module. To generate the datapath, first all field parameters and constants are set, then:

1. **Extract submodule.** To extract the submodules, we parse the top-level expression. GAP uses algebraic normal form for expressions such as the right hand side in Equation (1). We ensure all exponents are reduced modulo $q - 1$, then split the expression into two vectors: monomials and coefficients. We obtain (i.) the multiplicative and additive constants from the coefficients, and (ii.) finite field multiplications and exponentiations from the monomials.

2. **Generate expressions.** To implement the submodules, we need an expression for each output of the submodule. Figure 2 shows an example of extracted finite field multiplication on top and generated multiplication expressions, i.e., the compoenent functions, on the bottom. The basis B must be known and the designer must select one of the FFCSA algorithms for the arithmetic operation in basis B.

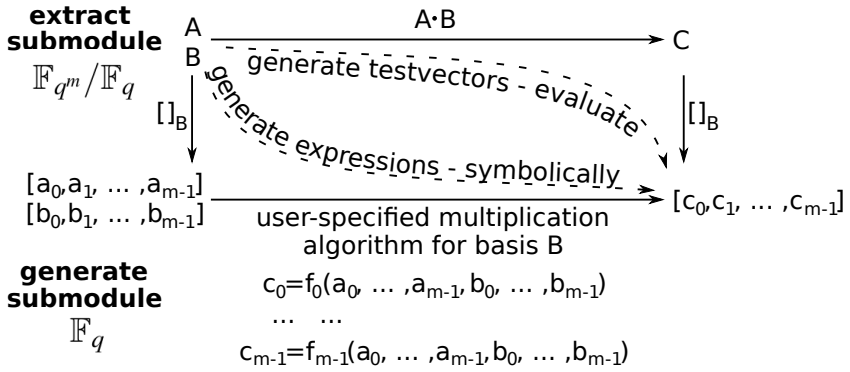


Figure 2: Extracted and generated multiplication

FFCSA implements a collection of methods to generate the required expressions according to a specified algorithm, parameterized for the current field instance. The package currently supports basic functionality (Section 3.1.2).

3.1.1 Symbolic computation

The `ChooseFieldElms(`F`)` method prepares vectors $avec = [a_0, \ldots, a_{m-1}]$ and $bvec = [b_0, \ldots, b_{m-1}]$ with default direction `To`, where $m = [\mathcal{F} : \mathcal{K}]$ is the degree of extension. Method `ChooseFieldElmsDownto` creates vectors $avec = [a_{m-1}, \ldots, a_0]$ and $bvec = [b_{m-1}, \ldots, b_0]$. Note that a_i, b_j are not coefficients, but GAP variables [5] to allow symbolic computation.

3.1.2 Generalized algorithm for multiplication

To compute the product $C = A \cdot B$, where $A, B \in \mathcal{F}/\mathcal{K}$ and $m = [\mathcal{F} : \mathcal{K}]$, we begin by forming the matrix U for a given basis $B_{\mathcal{F}/\mathcal{K}}$. Expressions obtained with matrix U follow the *Generalized algorithm for multiplication* [6]. This method produces a matrix-vector multiplier, where one of the factors is merged into the matrix U and then multiplied by the other factor. We chose this algorithm for multiplication as it is universal in the sense that it works for an arbitrary basis.

The matrix U is an $m \times m$ matrix with components $u_{i,j}$ $(0 \leq i, j \leq m-1)$ obtained by multiplying an element A with the j-th basis element $\rho^{(j)}$ and then taking the i-th coordinate of $\rho^{(j)} \cdot A$:

$$u_{i,j} = [\rho^{(j)} \cdot A]_{(i)}. \tag{3}$$

The columns of matrix U are exactly the vectors $[\rho^{(j)}A]$. The product $C = A \cdot B$ can be written in matrix form as

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} u_{0,0} & u_{0,1} & \cdots & u_{0,m-1} \\ u_{1,0} & u_{1,1} & \cdots & u_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m-1,0} & u_{m-1,1} & \cdots & u_{m-1,m-1} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix}. \tag{4}$$

The expressions for the product C can then be obtained by multiplying the right hand side of equation (4):

$$c_i = \sum_{j=0}^{m-1} u_{i,j} b_j \qquad (0 \leq i \leq m-1). \tag{5}$$

This generates the expressions $c_i = f_i(a_0, \ldots, a_{m-1}, b_0, \ldots, b_{m-1})$, which are used for the implementation of a circuit, where a combinational datapath is implemented for each output, e.g., each c_i for the multiplier in Figure 2. Similarly, we can generate expressions for arbitrary exponents of A, including the inverse. Since these methods use the *avec* variables a_i more than once, we must ensure their exponents are reduced modulo $|\mathcal{K}| - 1$ on each step.

FFCSA contains the following methods:

- `MatrixUExpression(`$B, avec$`)` computes the matrix U with elements obtained based on Equation (3).
- `FFA_mult_matrixU(`$B, avec, bvec$`)` first computes U with `MatrixUExpression` and *avec*, and then returns $U * bvec$.
- `FFA_sq_matrixU(`$B, avec$`)` first computes U with the method `MatrixUExpression` and *avec*, and then returns $U * avec$.
- `FFA_exp_matrixU(`$B, avec, e$`)` computes the expressions for exponentiation A^e, using a classic square and multiply with methods `FFA_mult_matrixU` and `FFA_sq_matrixU`.
- `FFA_inv_matrixU(`$B, avec$`)` computes the inverse expressions as exponentiation `FFA_exp_matrixU` with $e = |\mathcal{F}| - 2$.

Example 3.1 (Multiplication expressions for $\mathbb{F}_{2^4}$) *We use irreducible polynomial $f(x) = x^4 + x + 1$ with root $\rho \in \mathbb{F}_{2^4}$ and the polynomial basis $PB = \{1, \rho, \rho^2, \rho^3\}$. The GAP code below shows the setup and the outputs of* `FFA_mult_matrixU`*: the expressions used for the hardware implementation. For example, to drive the multiplier output c_0, the expression $a_0b_0 + a_1b_3 + a_2b_2 + a_3b_1$ must be implemented in hardware. No submodules are needed to compute a_ib_j terms, since multiplication in $\mathbb{F}_2$ is implemented with an* `AND` *gate.*

GAP code 3.1

```
gap> K := GF(2);; x := X(K, "x");;
gap> f := x^4+x+1;; F := FieldExtension(K,  f);; ChooseFieldElms(F);
variables
[ "a_0", "a_1", "a_2", "a_3" ]
... OMITTED ...
gap> PB := GeneratePB(F, RootOfDefiningPolynomial(F));;
gap> mult := FFA_mult_matrixU(PB, avec, bvec);;
gap> for i in mult do Display(i); od;
a_0*b_0+a_1*b_3+a_2*b_2+a_3*b_1
a_0*b_1+a_1*b_0+a_1*b_3+a_2*b_2+a_2*b_3+a_3*b_1+a_3*b_2
a_0*b_2+a_1*b_1+a_2*b_0+a_2*b_3+a_3*b_2+a_3*b_3
a_0*b_3+a_1*b_2+a_2*b_1+a_3*b_0+a_3*b_3
```

FFCSA package can also generate other expressions commonly needed for implementations. `MatrixMultByConstExpression`(B, `ffe`, $avec$) returns the expressions for implementing a constant multiplier submodule ${}_{\times}\gamma$, where ffe $= \gamma \in \mathcal{F}$ and B the basis. The method `TransitionMatrixExpression`($B1$, $B2$, $avec$) returns the expressions needed to implement the basis transition $B1 \to B2$.

3.2 Finite field constructions and searches

To produce a candidate list for DSE, we first search for defining polynomials or normal elements, and then generate the polynomial or normal bases, respectively. Current version of FFCSA can generate polynomial bases, normal bases, and their dual bases, and finally, different tower-field bases (Section 3.2.1). We implemented a set of methods to find normal elements. `FindNormalFFEs`$^{\spadesuit}$`(`F`)` checks the elements of $\mathcal{F}$ one by one using the method `IsNormalFFE(`F`,ffe)`. The symbol $^{\spadesuit}$ stands for optional `IgnoreConjugates` and reduces the search space. Many FFCSA methods use cyclotomic coset leaders to reduce search space [4]. `IsNormalFFE` check for ffe $= \rho$ computes the polynomial $T_\rho(x) = \sum_{i=0}^{m-1} \sigma^i(\rho)x^i$, where σ is the Frobenius map of $\mathcal{F}$, and returns `true` iff $\gcd(T_\rho(x), x^m - 1) = 1$ (by Theorem 5.2.11(1.) in [7]).

3.2.1 Generating tower field bases

For a composite integer $m = n_1 \cdot \ldots \cdot n_k$, where n_i $(1 \le i \le k)$ is a positive integer (not necessarily prime), it is possible to build $\mathbb{F}_{p^m}$ as a tower of extensions $\mathbb{F}_{(\ldots((p^{n_1})^{n_2})\ldots)^{n_k}}$ over its prime subfield $\mathbb{F}_p$:

$$\mathbb{F}_p = \mathcal{K}_0 \subset \mathcal{K}_1 \subset \cdots \subset \mathcal{K}_{k-1} \subset \mathcal{K}_k = \mathbb{F}_{(\ldots((p^{n_1})^{n_2})\ldots)^{n_k}} \cong \mathbb{F}_{p^m}.$$

We allow different options using either the same type of basis on each level, or mixed bases, e.g., polynomial basis on one level and normal basis on the next. In FFCSA package we make a distinction between reference field defining polynomials (RDP) and extension field defining polynomial (EDP). For example, the reference field for $\mathbb{F}_{((2^2)^2)^2}$ is the isomorphic $\mathbb{F}_{2^8}$ with a RDP of degree 8. The tower field is obtained with EDPs f_1, f_2, f_3 of degree 2, see Table 1.

With each new extension $\mathcal{K}_j/\mathcal{K}_{j-1}$ $(1 \le j \le k)$ we find the "per-level" basis (PLB). For an arbitrary expression over $\mathcal{K}_j/\mathcal{K}_{j-1}$ we extract and generate all its submodules, as was described in Section 3.1. We repeat the submodule extraction and generation for each level of the tower field until $\mathcal{K}_0 = \mathbb{F}_p$ is reached. For each lower level, we have to call `ChooseFieldElms` anew. In example 3.2 we need $\mathbb{F}_{(2^2)^2}$ multiplier submodules to compute $a_i b_j$ terms appearing in the generated expressions.

Table 1: Tower construction of $\mathbb{F}_{((2^2)^2)^2}$

$$\mathbb{F}_2 \xrightarrow{f_1(x)} \mathbb{F}_{2^2} \xrightarrow{f_2(x)} \mathbb{F}_{(2^2)^2} \xrightarrow{f_3(x)} \mathbb{F}_{((2^2)^2)^2}$$

Finite field $\mathbb{F}_{q^2}$	**Extension defining polynomial (EDP)** $f_i(x)$	**"per-level" PB** $B_{\mathbb{F}_{q^2}/\mathbb{F}_q} = \{1, \rho\}$	**Comments** $f_i(\rho) = 0$
$\mathbb{F}_{((2^2)^2)^2}$	$f_3(x) = x^2 + \lambda x + \lambda^2\mu$	$\{1, \nu\}$	$f_3(\nu) = 0$
$\mathbb{F}_{(2^2)^2}$	$f_2(x) = x^2 + \lambda x + 1$	$\{1, \mu\}$	$f_2(\mu) = 0$
$\mathbb{F}_{2^2}$	$f_1(x) = x^2 + x + 1$	$\{1, \lambda\}$	$f_1(\lambda) = 0$

The PLBs lead to the construction of a tower field basis (TFB) of the isomorphic field $\mathbb{F}_{2^8}/\mathbb{F}_2$ obtained as products of PLB elements as $\mathrm{TFB}_{\mathbb{F}_{2^8}/\mathbb{F}_2} = \{t_0, t_1, \ldots, t_7\} = \{1, \lambda, \mu, \mu\lambda, \nu, \nu\lambda, \nu\mu, \nu\mu\lambda\}$. When needed, the TFB is used for transition matrices between the tower field construction and the isomorphic field, constructed with a single extension, e.g., between $\mathbb{F}_{((2^2)^2)^2}$ and $\mathbb{F}_{2^8}$. TFB is also used to generate the testvectors for the top-level hardware modules.

Example 3.2 shows the interplay of different parts of the FFCSA package: search for irreducible polynomials, extension fields, bases, expressions for a multiplier, and finally the TFB for testvectors.

Example 3.2 *(Multiplication expressions for $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$.) In this example we show the construction $\mathbb{F}_{((2^2)^2)^2}$ with the EDPs listed in Table 1. The initial setup requires the list of EDPs selected from the output generated by* `FindEDPLAllfromEDL`(n_1, n_2, n_3)*. The long outputs were manually shortened for this example. The input to method* `FFA_mult_matrixU` *is the "per-level" polynomial basis B3, obtained for $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$. It produces the expressions for the multiplication on the top level of the tower field $\mathbb{F}_{((2^2)^2)^2}/\mathbb{F}_{(2^2)^2}$. Note that* `ChooseFieldElms` *in the GAP code* `Example 3.2` *returns vectors of length 2, not 8. The multiplications in expressions for the product need a multiplier from the lower level $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$ (submodule). We also need two subfield constant multipliers for $\gamma_1 = \lambda^2\mu \in \mathbb{F}_{(2^2)^2}$ and $\gamma_2 = \lambda \in \mathbb{F}_{(2^2)^2}$. Although $\lambda \in \mathbb{F}_{2^2}$, the product $a_1 b_1 \in \mathbb{F}_{(2^2)^2}$, therefore we must perform $\times\gamma_2$ in $\mathbb{F}_{(2^2)^2}/\mathbb{F}_{2^2}$.*

Example 3.2

```
gap> K := GF(2);; listall := FindEDPLAllfromEDL([2,2,2]);
[ [ x^2+x+Z(2)^0 ], [ x^2+Z(2^2)*x+Z(2)^0,     ... OMITTED ... ],
[ x^2+Z(2^4)^3*x+Z(2)^0, ... OMITTED ... , x^2+Z(2^2)*x+Z(2^4),
... OMITTED ... , x^2+Z(2^4)^13*x+Z(2^4)^14 ] ]
gap> EDPlist := [ listall[1][1], listall[2][1], listall[3][25] ];
[ x^2+x+Z(2)^0, x^2+Z(2^2)*x+Z(2)^0, x^2+Z(2^2)*x+Z(2^4) ]
gap> f1 := EDPlist[1];; f2 := EDPlist[2];; f3 := EDPlist[3];;
gap> F1 := FieldExtension(K,  f1);; F2 := FieldExtension(F1, f2);;
gap> F3 := FieldExtension(F2, f3);; nu := RootOfDefiningPolynomial(F3);;
gap> B3 := GeneratePB(F3, nu);  ChooseFieldElms(F3);
Basis( AsField( AsField( GF(2^2), GF(2^4)), GF(2^8)), [ Z(2)^0, Z(2^8)^76 ])
```

```
variables
[ "a_0", "a_1" ]
... OMITTED ...
gap> multB3 := FFA_mult_matrixU(B3, avec, bvec);;
gap> for i in multB3 do Display(i); od;
a_0*b_0+Z(2^4)*a_1*b_1
a_0*b_1+a_1*b_0+Z(2^2)*a_1*b_1
gap> lambda := RootOfDefiningPolynomial(F1);;
gap> mu := RootOfDefiningPolynomial(F2);;
gap> lambda^2*mu; lambda;
Z(2^4)
Z(2^2)
gap> Mlist := [["PB", "to"], ["PB", "to"], ["PB", "to"]];
gap> TFB2 := GenerateTFBfromEDPLwithMB(EDPlist, Mlist); nu*mu*lambda;
Basis( GF(2^8), [ Z(2)^0, Z(2^2), Z(2^4)^6, Z(2^4)^11, Z(2^8)^76,
Z(2^8)^161, Z(2^8)^178, Z(2^8)^8 ] )
Z(2^8)^8
gap> ffe := Z(2^8)^15;; VecToString(Coefficients(TFB, ffe);
"11101010"
```

3.3 FFCSA profiling methods

Specialized search is closely linked to the design space exploration: theoretical estimates can be used to make architectural decisions early in the design flow and to reduce the design space. For this purpose we use a set of Hamming weights: a theoretical estimate of area as is obtained by `WeightMatrix(`M`)` method, and a theoretical estimate for delay by `WeightMatrixMaxRow(`M`)` method. For example, `ProfileGamma(`B`)` computes the profiles for constants $\gamma_i = \omega^{e_i}$. The profile is $[d, A, e_1, e_2, \dots]$, where d and A are the delay and area of the matrix-vector multiplier for γ_i. The exponents are grouped together when their delay and area are the same. The "special" element in this case would be γ_i with the smallest area.

A perfect example of optimizaing implementations based on search results is the well-known block cipher AES with constructions $\mathbb{F}_{(2^4)^2}$ and $\mathbb{F}_{((2^2)^2)^2}$ in [12, 13], and new results still appearing in the literature [10]. Currenlty, a lot of research is focusing on finding estimates that are more accurate than Hamming weight, e.g., sequential XOR count [3]. We will add more sophisticated offline profiling methods to FFCSA in the future.

4 Conclusion

The FFCSA construction methods allow generation of polynomial and normal bases, their dual bases, and bases for tower fields. The search algorithms can be classified as exhaustive search (e.g., find all normal bases), reduced search space (e.g., ignore conjugates), and specialized search (e.g., find a primitive polynomial with a specified number of nonzero coefficients). Specialized search is a form of reduced search space, but the reduction criteria is different. The main purpose of FFCSA algorithms is on-the-fly generation of expressions needed for hardware implementations, to enable automated Design Space Exploration. Future work entails Design Space Exploration, which requires two additional phases. First is the automated hardware generation of extracted submodules and the implementation of the top-level module. Second, the interaction with CAD tools to obtain post-synthesis results.

Acknowledgements

The authors would like to thank A. Hasan for discussions and advice on the Generalized algorithm for multiplication. The research of G. Gong and M. Aagaard were supported by NSERC Canada and the research of N. Zidaric by NSERC Canada and NWO Netherlands through the PROACT project. The research of O. Konovalov was supported by the OpenDreamKit Horizon 2020 European Research Infrastructures project.

References

[1] M. Aagaard, R. AlTawy, G. Gong, K. Mandal, R. Rohit, and N. Zidaric. WAGE: An authenticated cipher, *nist lwc* round 2, 2019.

[2] R. AlTawy, G. Gong, K. Mandal, and R. Rohit. WAGE: an authenticated encryption with a twist. *IACR Transactions on Symmetric Cryptology – Special Issue on Designs for the NIST Lightweight Standardisation Process*, pages 132–159, 2020.

[3] C. Beierle, T. Kranz, and G. Leander. Lightweight Multiplication in $gf(2^n)$ with Applications to MDS Matrices. In *CRYPTO'16*, pages 625–653, 2016.

[4] S.W. Golomb and G. Gong. *Signal design for good correlation.* Cambridge University Press, New York, NY, 2005.

[5] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.12.2*, 2022.

[6] A. Hasan. Selected topics in cryptographic computations. ECE-720/2, Lecture notes, University of Waterloo, 2017.

[7] G. L. Mullen and D. Panario, editors. *Handbook of finite fields.* Discrete mathematics and its applications. CRC Press, Boca Raton, FL, 2013.

[8] Y. Nawaz and G. Gong. WG: A family of stream ciphers with designed randomness properties. *Information Sciences*, 178(7):1903–1916, 2008.

[9] M. Pfeiffer, M. Martins, O. Konovalov, and the GAP Team. JupyterKernel, Version 1.5.0. `https://gap-packages.github.io/JupyterKernel/`, 2023.

[10] A. Pradeep, V. Mohanty, A. M. Subramaniam, and C. Rebeiro. Revisiting AES SBox Composite Field Implementations for FPGAs. *IEEE Embedded Systems Letters*, 11(3):85–88, 2019.

[11] A. Reyhani-Masoleh. A new construction of Massey-Omura parallel multiplier over GF(2^m). *IEEE Transactions on Computers*, 51(5):511–520, 2002.

[12] V. Rijmen. Efficient implementation of the rijndael s-box. Technical report, 2000.

[13] A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A compact rijndael hardware architecture with s-box optimization. In *ASIACRYPT'01*, pages 239–254, 2001.

[14] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.8)*, 2023. `https://www.sagemath.org`.

[15] N. Zidaric. *Automated Design Space Exploration and Datapath Synthesis for Finite Field Arithmetic with Applications to Lightweight Cryptography. UWSpace.* PhD thesis, 2020.

[16] N. Zidaric. FFCSA, Version 1.0.4. `https://nzidaric.github.io/ffcsa/`, 2023.

[17] N. Zidaric, M. Aagaard, and G. Gong. Rapid Hardware Design for Cryptographic Modules with Filtering Structures over Small Finite Fields. In *WAIFI'18, LNCS, vol. 11321*, pages 128–145, 2018.

[18] N. Zidaric, M. Aagaard, and G. Gong. FSR, Version 1.2.2. `https://nzidaric.github.io/fsr/`, 2019.

[19] Nuša Zidarič and Mark Aagaard. Poster: Tower field support for synthesis of datapaths. In *Computing Frontiers (CF '23)*, 2023.