Optimizing Implementations of Lightweight Building Blocks

Jérémy Jean[1], Thomas Peyrin[2], Siang Meng Sim[2] and Jade Tourteaux[1,3]

[1] ANSSI Crypto Lab, Paris, France
Jeremy.Jean@ssi.gouv.fr

[2] Nanyang Technological University, Singapore
Thomas.Peyrin@ntu.edu.sg, ssim011@e.ntu.edu.sg

[3] Paris Diderot University, Paris, France
Jade.Tourteaux@gmail.com

Abstract. We study the synthesis of small functions used as building blocks in lightweight cryptographic designs in terms of hardware implementations. This phase most notably appears during the ASIC implementation of cryptographic primitives. The quality of this step directly affects the output circuit, and while general tools exist to carry out this task, most of them belong to proprietary software suites and apply heuristics to *any* size of functions. In this work, we focus on *small* functions (4- and 8-bit mappings) and look for their optimal implementations on a specific weighted instructions set which allows fine tuning of the technology.

We propose a tool named `LIGHTER`, based on two related algorithms, that produces optimized implementations of small functions. To demonstrate the validity and usefulness of our tool, we applied it to two practical cases: first, linear permutations that define diffusion in most of SPN ciphers; second, non-linear 4-bit permutations that are used in many lightweight block ciphers. For linear permutations, we exhibit several new MDS diffusion matrices lighter than the state-of-the-art, and we also decrease the implementation cost of several already known MDS matrices. As for non-linear permutations, `LIGHTER` outperforms the area-optimized synthesis of the state-of-the-art academic tool ABC. Smaller circuits can also be reached when ABC and `LIGHTER` are used jointly.

Keywords: `LIGHTER` · Implementation · ASIC · Lightweight Block Ciphers · Boolean function · Meet-in-the-Middle · Sbox · MDS Matrix

1 Introduction

Pervasive computing is becoming increasingly important in many applications of our daily life. Lightweight devices such as RFID tags and wireless sensor nodes might manipulate sensitive data and thus usually require some security. Yet, classical cryptographic algorithms may not be very suitable for this type of applications, especially for very constrained environments. As a consequence, lightweight cryptography has become an extremely active research topic in the recent years, with several new lightweight symmetric-key primitives being proposed, e.g., [8, 9, 13, 41, 44]. In this context, there have been many advances in finding the best possible and in particular the most lightweight building bricks to design a symmetric-key primitive. In particular, diffusion matrices [10, 29, 31, 32, 42] and Sboxes [18, 19, 34, 38, 46] were thoroughly scrutinized as they are considered classical components of modern SPN ciphers.

What criterion to optimize for lightweight applications might significantly vary depending on the exact application (area, throughput, energy, power), but compact implementations, i.e. minimizing the area required by the hardware implementation, is generally considered of major importance as it represents a direct constraint in practice (even though the technology evolved since then, it was for example expected in 2005 that an RFID tag could spend at most 2000 GE for its security [28]), and also indirectly affect energy and power consumptions. Usual figures of area reported for cryptographic implementations usually contain the cost of storage – the registers that hold the key and the internal state – as well as the combinatorial logic, which implements the actual operations. Therefore, minimization of storage cost and combinatorial logic cost (by the means of reducing the number of operations required to apply the cryptographic function) is a very desirable goal. This can be achieved by design (for example by selecting a sparse diffusion matrix in a hope to reduce the total number of XORs required to compute it), but the study of their actual implementation can have a significant impact as well, and is often overlooked.

There are many different types of hardware implementations (serial, round-based, fully unrolled, and other variations) and many different technologies available, each having its own set of logic operations costs for example. All of these dimensions offer countless opportunities for optimization and it is not trivial for a researcher or an engineer to find the optimal hardware implementation for his own particular scenario. One can even extend this observation to other platforms and other criteria than area: How can I optimize this cipher on software since the logical operations' costs are very different than for hardware? How can I optimize the implementation of this lightweight cipher when I care about the delay and not the area? An optimized implementation from the designers might not be optimized for other use-cases, as we will see later.

For symmetric-key primitives, it is therefore interesting to study given an Sbox or given a certain linear diffusion layer what is the best possible (or at least a very good) implementation for a particular environment and how it can be obtained. Most of the time, this work is performed by synthesizers that convert small lookup tables (LUT) to boolean circuits: the implementer will design the general architecture of the cipher, but will leave the optimization of the small components to automated tools. While these tools do an excellent job at improving performances for generic functions, their output might not be optimal.

Our Contributions. In this article, we propose a new automated tool, `LIGHTER`,[1] that can either search for new lightweight cryptographic components or find optimized implementations of lightweight components, such as Sboxes or diffusion matrices. This tool is based on several advances.

First, we introduce a graph-based meet-in-the-middle (MITM) search algorithm that can generate an efficient implementation of a small (linear or non-linear) function given a certain set of available instructions and their corresponding costs. This algorithm is simple yet extremely generic and can be used for many purposes. The most obvious utilization is to compute the smallest implementation of lightweight encryption building blocks, by encoding the logical instruction area costs inside the algorithm's input costs. This allows an implementer to easily obtain very good implementations that exactly match his hardware configuration. The algorithm can also be used to optimize for delay instead of area, or to compute the best software implementation (by simply setting all the logic operations to the same cost). Another example would be to minimize the amount of linear or non-linear operations of a given function by setting the operations cost accordingly, which would be very useful for multi-party computation, zero-knowledge proofs, masking against side-channel attacks or fully homomorphic encryption schemes [2]. More utilizations are probably possible.

[1] http://jeremy.jean.free.fr/pub/fse2018_layer_implementations.tar.gz

The time and memory complexities of this first algorithm are directly related to the number of instructions considered in the implementation, which can therefore grow beyond practical ranges for high numbers of instructions. Thus, we propose in a second part another algorithm that computes good (but not-necessarily optimal) implementations of small functions, without specific limitation on the number of instructions. The idea is to perform a divide-and-conquer approach and improve locally sub-parts of a given starting implementation. This approach can therefore be seen as a tradeoff between optimality of the output and tractability of the computation.

Combining both algorithms leads to very good results. Namely, we first examine the area-optimized implementation of finite field multiplications over $\mathrm{GF}(2^4)$ and $\mathrm{GF}(2^8)$ and show that many multiplication coefficients are actually much cheaper than originally thought. This allowed us not only to find new best diffusion matrices (for example the new best 4×4 involutory diffusion matrix over $\mathrm{GF}(2^4)$), but also to improve the implementation of many existing ones (the diffusion matrices of the `AES` block cipher, the `WHIRLPOOL` or `Grøstl` hash functions being some of them).

Linear layers are not the only components that can be handled. We apply `LIGHTER` to search for area-optimized implementations of several 4-bit Sboxes and we show that our tool outperforms the state-of-the-art synthesis tool ABC almost all the time. Besides, when given the output from our tool, ABC improves over its own implementation obtained from the lookup table of the Sbox. We note that due to legal reasons, we only compare our algorithms to the academic tool ABC and not with other proprietary widely-used algorithms, whose usage are restricted by non-disclosure agreements. However, we are confident that similar conclusions could be drawn with such tools.

`LIGHTER` does have some limitations. For example, it does not always guarantee that a given implementation is optimal according to the constraints provided as input. This is particularly true for non-linear layers, as we need to combine several operations to ensure that the instructions used are invertible. Besides, unlike for general synthesizers, `LIGHTER` will mainly work for small functions (like 4-bit Sboxes) since for larger sizes, the tool will require too much memory and computation. Yet, this limitation does not apply to most lightweight designs for which small cryptographic components are a requirement. Finally, we emphasize that `LIGHTER` is currently not handling the case of optimizing FPGA implementations, which requires a very different optimization strategy.

Organization of the Paper. In Section 2, we first recall some previous works on the implementation of linear and non-linear layers and we introduce some preliminary notions. Then, in Section 3, we introduce two algorithms that search for efficient implementations of small functions, the core of our tool `LIGHTER`. In Section 4, we propose several heuristics to further help the efficient implementations search, and in Section 5 several implementation improvements of known matrices, but also new efficient matrix candidates. Eventually, we provide further results by application of the tool on non-linear layers in Section 6.

2 Preliminaries

2.1 Previous Work

In the past few years, there has been an important focus on lightweight cryptography with several new research directions. Among them, we can distinguish two main dimensions, which are the search/implementation of linear/non-linear layers. We recall briefly some of the main results related to our paper in the following.

Linear Layers. Most of the recent work focuses on searching for lightweight maximum distance separable (MDS) diffusion matrices to be used in SPN, that is matrices that provide maximum diffusion property. In [29], the authors proposed quantifying the hardware implementation cost of diffusion matrices by counting the number of XOR gates needed to implement the linear layer (simply called *XOR count* of the matrix). Several works like [31,32,40,42] adopted this metric and search for different types of lightweight diffusion matrices. In [10], the authors used a single field element to construct new lightweight left-circulant matrices. Notably in [31], the authors considered invertible binary matrices as the entries of the diffusion matrix rather than field elements. The non-commutative property of the invertible binary matrices allows them to construct involution (self-inverse) MDS circulant matrices of order 4 and 8, which was proven to be non-existent over finite fields GF(2^n) [26]. More details about these papers will be discussed in Section 5.2.

Non-linear Layers. The research line studying the implementation of cryptographic non-linear functions with few input variables (e.g., less than eight) falls within the much broader domain of logic synthesis [21]. There have been extensive research in this area, in particular with applications to Very Large Scale Integration (VLSI), see [16] for instance. While general circuit minimization problems are known to be Σ_2^P-complete [17], there are several heuristic algorithms that provide suboptimal solutions, such as BOOM [27] or the more commonly used ESPRESSO algorithm [37], which is probably implemented in many commercial synthesizers.

In the particular case of small cryptographic building blocks (e.g., 4-bit to 4-bit functions), the same heuristics algorithms seem to be used by the general synthesizers but it does not seem infeasible to reach optimal solutions due to the small dimension of the problem. Several academic papers carry out either full or partial exhaustive searches on the 4-bit permutations space from different points of view. Indeed, this search space has a size within practical reach (even without elaborated pruning strategies) as it "only" contains $2^4! \approx 2^{44.2}$ permutations.

For instance in [46], the authors look for efficient bit-sliced implementations of 4-bit permutations grouped in affine equivalence classes. The enumeration relies on a depth-first traversal of a tree labeled by permutations, where one goes down one level by applying one operation from AND, OR, XOR, NOT, and MOV. The search uses extra memory unit (in the form on an additional temporary register) and applies the rules from Osvik [34], which essentially provide ways to add cuts in the tree (idea initially applied to optimize the software implementation of `SERPENT` [12]). The results described by the authors cover about 90% of the search space, and they derive a very small 9-instruction software implementation of a 4-bit permutation with good cryptographic properties (differential probability and absolute linear bias being 2^{-2}).

In [38], Saarinen enumerates and classifies all 4-bit permutations up to permutation equivalence (about 2^{27} equivalence classes). One of the reasons to introduce this class pertains to the similarity of the implementations of the elements within the same class, which in general is not the case for the elements within a same affine equivalence class. The main contribution of the paper is the classification of the permutation equivalence classes with respect to cryptographic properties and the introduction of so-called Golden Sboxes which maximize all these criteria.

More recently, Boyar, Matthews and Peralta introduced in [14] a new technique for combinational logic optimization which essentially relies on a two-step algorithm successively reducing the number of non-linear then linear gates required. They apply this algorithm to the `AES` Sbox and improve upon previous results, most notably the Canright decomposition from [18]. Later, several papers improved under various metrics the design of the `AES` Sbox, for instance [1,49,52].

In [19], the authors construct a small implementation of an 8-bit Sbox from a small implementation of a 4-bit Sbox. The main contribution is to deduce cryptographic properties of the large Sbox from properties of the small Sbox while at the same time ensuring the implementation overhead is minimized.

In [3], the authors introduced a metric called *depth* to estimate the path delay of an Sbox, and designed two new Sboxes Sb_0 and Sb_1 with small delay as part of a new energy-efficient cipher called MIDORI. Based on this metric, the authors of [23] developed an Sbox depth evaluation tool, which computes the logical operation expression with least depth of a given 4-bit Sbox.

Finally in [43], Stoffelen models the problem of finding an efficient implementation (in terms of number of gates) of a lightweight Sbox as a SAT problem. Additional criteria like number of non-linear gates or depth can be injected in the SAT problem, which is then solved more or less efficiently using off-the-shelf SAT solvers. However, we note that this technique does not allow to optimize the implementation depending on the technology that will be used. Thus, it is likely to give implementations that will be rather good in general, but not really optimized for a particular technology, as we will see in our comparisons.

2.2 Preliminaries for Linear Layer

The linear layer of a Substitution-Permutation Network (SPN) can usually be represented by a so-called *diffusion matrix*, that mixes k words of c bits into k other c-bit words to create the diffusion. The coefficients of a diffusion matrix usually belong to some finite field $\mathrm{GF}(2^c)$ (while some exceptions like [31] exist), and the multiplication of the finite field elements are defined by some irreducible polynomial[2] of degree c. When necessary, we append the irreducible polynomial $p(X)$ in hexadecimal form to the finite field: $\mathrm{GF}(2^c)/p(X)$. We also denote the general linear group of degree c over a field $\mathbb{K}$ by $\mathrm{GL}(c, \mathbb{K})$.

A Maximum Distance Separable (MDS) diffusion matrix is one that has the maximum diffusion power and relies on an MDS code. This property offers perfect diffusion as changing m words of the inputs changes at least $k - m + 1$ of the outputs [47]. It is known that a necessary and sufficient condition for a matrix to be MDS is that all its square submatrices should be invertible (non-singular) [33], and as a consequence, it is necessary that all the coefficients of an MDS matrix are nonzero. A typical example of an MDS diffusion matrix appears in the MixColumns operation of the AES.

As proposed in [29], the hardware implementation cost of a diffusion matrix can be quantified by counting the number of XOR gates needed to implement it. Note that for MDS matrices, the $k-1$ many c-bit XORs per row (so-called *connecting XORs*) are a constant and incompressible cost. However, the variable costs are the implementation costs of the field multiplications, which is what we are interested in optimizing.

Example 1. The application of the MixColumns operation of the AES can be expressed as a multiplication by a diffusion matrix $\mathbf{M}$ over $\mathrm{GF}(2^8)/$`0x11b`,

$$\underbrace{\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2a \oplus 3b \oplus c \oplus d \\ a \oplus 2b \oplus 3c \oplus d \\ a \oplus b \oplus 2c \oplus 3d \\ 3a \oplus b \oplus c \oplus 2d \end{bmatrix},$$

where $(a, b, c, d)^\top$ is a column of the state and $\oplus$ denotes a c-bit XOR. In the case of the AES, each $\oplus$ costs 8 XOR gates.[3] Therefore, the total cost to implement this diffusion matrix is $4 \times (C(2) + C(3) + 24)$ XOR gates, where $C(\alpha)$ is the implementation cost of the field multiplication α. □

[2]As seen in [10, 39], the choice of basis also plays a part in the representation of the elements. In this work, we only consider the polynomial basis as it is the most popular choice of basis.

[3]In the sequel, an XOR gate refers to a 2-input XOR gate.

We address two different problems in the paper: finding new MDS matrices with smaller implementation costs, and optimizing these costs for already known matrices. In both cases, we start by improving the implementation cost of the multiplication of the field elements, and we then either construct new lightweight MDS diffusion matrices, or we derive an optimized implementation of a given MDS matrix based on these new implementation costs.

Therefore, the linear layer discussion in this paper is split into two parts. We first present the method to improve the implementation costs of the field elements, then based on this metric, we search for new lightweight diffusion matrices.

Multiplication Matrix of Finite Field Elements. Given a finite field $\mathrm{GF}(2^c)$, we can represent its elements using c-tuple vectors over $\mathrm{GF}(2)$. The multiplication of a nonzero element $\alpha \in \mathrm{GF}(2^c)$ can be viewed as a left *multiplication matrix* of order c over $\mathrm{GF}(2)$ that is often known as multiplication matrix and denoted $\mathbf{M}_\alpha$ [10,39,42]. One can quickly infer that multiplication matrices of nonzero element are invertible and pairwise-commutative, since nonzero elements in $\mathrm{GF}(2^c)$ are invertible and field multiplication is commutative.

d-XOR metric. In [29], the authors proposed to quantify the implementation cost of a field element multiplication by *directly* counting the number of '1's in each row of the multiplication matrix. We call this metric *d-XOR*.

Definition 1 (d-XOR, [29])**.** The d-XOR value of a finite field element $\alpha \in \mathrm{GF}(2^c)/p(X)$ is a metric to estimate the number of XOR operations needed to implement the field multiplication by α: $x \to \alpha x$. It is counted as the Hamming weight of the multiplication matrix $\mathbf{M}_\alpha$ minus the number of rows, and denoted by $\text{d-XOR}(\mathbf{M}_\alpha) = \omega(\mathbf{M}_\alpha) - c$, where $\omega(\mathbf{M}_\alpha)$ is the number of '1's in $\mathbf{M}_\alpha$ and c is the number of rows. When it is clear from the context that $\mathbf{M}_\alpha$ refers to the multiplication matrix for α: $x \to \alpha x$, one simply writes $\text{d-XOR}(\alpha)$.

Note that this only provides an overestimation of the minimal implementation cost of the finite field elements: in practice the number of XOR operations required could be smaller. In the sequel, we introduce another metric to quantify the implementation cost.

s-XOR metric. In practice, given an arbitrary input vector, we can implement any finite field multiplication *in place*, that is by updating the components of the vector without using extra memory storage. This can be realized by performing a *sequence* of XOR instructions (e.g., $R_i \leftarrow R_i \oplus R_j$ for some rows R_i and R_j).

Definition 2 (s-XOR)**.** The s-XOR of a field element α is the minimum number of XOR operations needed to implement the left multiplication of a multiplication matrix $\mathbf{M}_\alpha$, where the minimum is taken over all implementation sequences. We denote it by $\text{s-XOR}(\mathbf{M})$ for an invertible binary matrix $\mathbf{M}$, or simply $\text{s-XOR}(\alpha)$ if the matrix is $\mathbf{M}_\alpha$.

Example 2. Given the finite field $\mathrm{GF}(2^3)/$`0xb`, the multiplication of $\alpha = 7$ seen as $(1,1,1) \in (\mathrm{GF}(2))^3$ can be computed by:

$$\begin{aligned}(1,1,1)(b_2,b_1,b_0) &= (b_2 \oplus b_0, b_2 \oplus b_1, b_1) \oplus (b_1, b_2 \oplus b_0, b_2) \oplus (b_2, b_1, b_0)\\ &= (b_1 \oplus b_0, b_0, b_2 \oplus b_1 \oplus b_0),\end{aligned}$$

where (b_2, b_1, b_0) is an arbitrary element of $\mathrm{GF}(2^3) \cong (\mathrm{GF}(2))^3$. Expressing the same computation as a matrix multiplication, it rewrites as

$$\begin{bmatrix}0 & 1 & 1\\ 0 & 0 & 1\\ 1 & 1 & 1\end{bmatrix}\begin{bmatrix}b_2\\ b_1\\ b_0\end{bmatrix} = \begin{bmatrix}b_1 \oplus b_0\\ b_0\\ b_2 \oplus b_1 \oplus b_0\end{bmatrix}.$$

From the multiplication matrix, we can see that *d-XOR*$(\alpha) = 3$ XOR instructions. To find one s-XOR value for α, we note that $b_1 \oplus b_0$ appears in both the first and third components. Hence in practice, we can upward-rotate the input vector components, XOR the second component to the first, followed by XORing the first component to the third to obtain the desired output. We can easily verify that such sequence has the minimum number of XORs to implement the multiplication matrix. Therefore, in that case, we get *s-XOR*$(\alpha) = 2 <$ *d-XOR*(α). □

Consequently, our goal is to find an optimal sequence (with the least number of XOR instructions) for any given multiplication matrix, or more generally, for any invertible binary matrix.

3 A Graph-Based Search

We now describe two generic algorithms that produce implementations of functions given a set of Boolean instructions $\mathcal{B}$ relying on some bitwise Boolean operation like AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), NOT ($\neg$), etc. We make the distinction between *operation* and *instruction* to capture the need for an internal state to produce an implementation. For instance, $\mathcal{B}$ could contain the following instructions

$$x \leftarrow x \wedge y, \qquad x \leftarrow x \vee y, \qquad x \leftarrow x \oplus y, \qquad x \leftarrow \neg x,$$

if we are interested in software implementations, or instructions based on logical 2-input gates like NAND, NOR, XOR, XNOR if we consider hardware implementations:

$$x \leftarrow \neg(x \wedge y), \qquad x \leftarrow \neg(x \vee y), \qquad x \leftarrow x \oplus y, \qquad x \leftarrow \neg(x \oplus y).$$

Definitions and Notations. In the rest of the paper, we call an implementation only using instructions from a set $\mathcal{B}$ a *$\mathcal{B}$-implementation*. In the case where $\mathcal{B}$ only contains the XOR instruction, we call them XOR-implementations. Furthermore, since all the functions we consider are $\mathrm{GF}(2^c) \rightarrow \mathrm{GF}(2^c)$, we make no distinction between the function f and the ordered sequence $(f(x))_{x \in \mathrm{GF}(2^c)}$ for a predetermined ordering of $\mathrm{GF}(2^c)$, and we use the term *function* to refer to both objects. Additionally, we define *one*[4] $\mathcal{B}$-implementation $\mathcal{I}_{Id}^{f}$ of a function f as the sequence $((f_i, o_i))_{i=1,\ldots,n}$, transforming the identity function into the function f using the instructions $o_i \in \mathcal{B}$ and intermediate functions f_i, for $i = 1, \ldots, n$. We represent it by:

$$Id \xrightarrow{o_1} f_1 \cdots \xrightarrow{o_n} f_n = f.$$

More generally, we introduce $\mathcal{I}_f^g$ to represent the sequence transforming a function f into another function g, and denote $|\mathcal{I}_f^g|$ its length in terms of number of instructions the sequence contains. For simplicity, we write $\mathcal{I}_{Id}^{f}$ as $\mathcal{I}^f$. The concatenation of two implementations $\mathcal{I}_f^g$ and $\mathcal{I}_g^h$ is denoted by $\mathcal{I}_f^g + \mathcal{I}_g^h$. Finally, we denote by $\|o\|$ the cost assigned to the instruction $o \in \mathcal{B}$ and abuse notations to define the cost of an implementation $\mathcal{I}_g^f = ((f_i, o_i))_{i=1,\ldots,n}$ by $\|\mathcal{I}_g^f\| = \sum_i \|o_i\|$.

The first algorithm described below finds optimally small $\mathcal{B}$-implementations of functions for user-defined costs of each element in $\mathcal{B}$. The problem solved can be expressed as an optimization problem where all the feasible solutions consist in $\mathcal{B}$-implementations of the input function, and are weighted by summing the individual cost affected to each instructions from $\mathcal{B}$. The algorithm then finds (one of) the best solution(s) using a graph-based approach (Section 3.1).

[4]Note that a function can be implemented in more than one way.

The second algorithm comes into play when the first one fails at producing the optimal solution (Section 3.2). Instead, by starting from a heuristically found $\mathcal{B}$-implementation of the input function, the algorithm incrementally reduces it to a smaller one under the same metric by locally applying the first algorithm.

3.1 Optimal $\mathcal{B}$-Implementations Using MITM Technique

We explicit in this section an algorithm allowing to reach optimally small $\mathcal{B}$-implementations of a given function. We recall that by optimal, we mean an implementation that minimizes an integer objective function defined as the cumulative sum of each individual cost $\|o\|$ of the instructions o used.

The algorithm relies on a generic meet-in-the-middle strategy that can accommodate different logical instructions with possibly different costs (e.g., area, delay, energy consumption, etc). As a result, the technique applies to any linear and non-linear cryptographic building blocks and targets any technology for specific user-tuned parameters.

The algorithm takes a function $f : \mathrm{GF}(2^c) \to \mathrm{GF}(2^c)$ as input, and outputs the circuit implementing f using a set of logical instructions $\mathcal{B}$ parameterized by some costs. We restrict the functions to those defined over the binary field $\mathrm{GF}(2^c)$ of order c to rely on a bit-sliced representation of the functions (see representation details in Section 3.3).

At a high level, the algorithm independently starts from the identity function and from the target function f, and expands each of them to sets of functions constructed by applying all the logical instructions from $\mathcal{B}$. Going backwards from f may induce a high branching depending of the set $\mathcal{B}$: we address this in Section 4. The process stops when the two constructed sets have a non-empty intersection that defines a sequence transforming the identity function into f.

We describe the algorithm as a solution to a graph problem, where the vertex set V represents the functions and there is an edge $(v_1 \to v_2) \in E$ between two nodes if v_2 can be deduced from v_1 by applying an instruction from $\mathcal{B}$. Then, the algorithm looks for the shortest path between the source node S (representing the identity function) and the target node T (representing the function f), where the distance is defined by summing the cost of each traversed edge (instruction).

At every step of the execution, the algorithm maintains an integer counter λ and the subgraph of all functions reachable from the root node at a distance at most λ. The initialization simply sets the counter to 0 and inserts the root node as starting point. Then, the graph of reachable functions is incrementally constructed by computing the closest nodes (functions) to the root not yet included in the graph, to finally add and connect those new nodes. Internally, all the nodes at the same distance from the root are stored in a same (lexicographically sorted) list, and there are as many lists as different distances. This allows to efficiently generate the newly introduced nodes.

We give a simplified pseudo-code description of this algorithm in the following Algorithm 1. The MitM function takes the two functions f_0 and f_1 as inputs and possibly a limit Λ after which the algorithm will abort. The returned value is one implementation $\mathcal{I}_{f_0}^{f_1}$. In the current case where we look for one implementation of the f function, we call this algorithm with $\textsc{MitM}(Id, f, \infty)$, with Id the identity function. Later in Section 3.3, the first function f_0 will not necessarily be Id, and Λ will be useful. For simplicity, we omit the details of $\textsc{GetImplementation}(v_0, I, v_1)$, which simply retrieves the full sequence of instructions (i.e., the implementation) from the traversed edge between the root node v_0, the half-way node I belonging to the two subgraphs, and the target node v_1. Similarly, we skip the specifics of $\textsc{Conv}(f)$ which converts a function $f : \mathrm{GF}(2^c) \to \mathrm{GF}(2^c)$ into its bit-sliced representation and that of $\textsc{Succ}(v, o)$, which returns the functions reachable from v using the instructions o (both are detailed in Section 3.3).

The core of the algorithm lies in the Expand function, which generates new nodes in the graph. As mentioned before, we structure the nodes of the set V according to their

Algorithm 1 – Meet-in-the-Middle Implementation Search (Simplified).

```
function MitM($f_0, f_1[, \Lambda = \infty]$)
  $v_0 \leftarrow \textsc{Conv}(f_0)$,   $\lambda_0 \leftarrow 0$,   $V_0^0 \leftarrow \{v_0\}$,   $\mathcal{G}_0 \overset{\text{def}}{=} (V_0, E_0) \leftarrow (\{V_0^0\}, \emptyset)$
  $v_1 \leftarrow \textsc{Conv}(f_1)$,   $\lambda_1 \leftarrow 0$,   $V_1^0 \leftarrow \{v_1\}$,   $\mathcal{G}_1 \overset{\text{def}}{=} (V_1, E_1) \leftarrow (\{V_1^0\}, \emptyset)$
  $\sigma \leftarrow 0$
  while $\lambda_0 + \lambda_1 < \Lambda$ do            ▷ Possible limitation to abort the algorithm
    $(\mathcal{G}_\sigma, \lambda_\sigma) \leftarrow \textsc{Expand}(\mathcal{G}_\sigma, \lambda_\sigma)$
    $I \leftarrow V_0 \cap V_1$                          ▷ Look for meet-in-the-middle collision
    if $I \neq \emptyset$ then return $\textsc{GetImplementation}(v_0, I, v_1)$
    $\sigma \leftarrow \sigma \oplus 1$
  return $\emptyset$
```

```
function Expand($\mathcal{G} = (\{V^0, \ldots, V^{\lambda-1}\}, E), \lambda$)
  $V^\lambda \leftarrow \emptyset$,   $E^\lambda \leftarrow \emptyset$                         ▷ Initialize $(V^\lambda, E^\lambda)$
  for all $o \in \mathcal{B}$ do                                 ▷ For all possible instructions
    $c \leftarrow \|o\|$                                       ▷ Retrieve the cost of instruction o
    for all $v \in V^{\lambda-c}$ do              ▷ Check whether a node in $V^\lambda$ can be created
      $S \leftarrow \textsc{Succ}(v, o)$                   ▷ All successors of v using instruction o
      $V^\lambda \leftarrow V^\lambda \cup S$
      $E^\lambda \leftarrow E^\lambda \cup \{v \to w,\ w \in S\}$
  $\mathcal{G} \leftarrow (\{V^0, \ldots, V^{\lambda-1}, V^\lambda\}, E \cup E^\lambda)$       ▷ Insert and connect new nodes (if any)
  if $V^\lambda = \emptyset$ then        ▷ If there are no possible nodes at distance $\lambda$, try $\lambda + 1$
    return $\textsc{Expand}(\mathcal{G}, \lambda + 1)$
  else
    return $(\mathcal{G}, \lambda)$
```

distance to the root. Namely, if node v is at distance d from the root, we store it in V^d, and then $V = \{V^d,\ d\}$. Now, Expand$(\mathcal{G}, \lambda)$ fills V^λ from the nodes already present in $\mathcal{G}$ using the instructions in $\mathcal{B}$.

We emphasize that Algorithm 1 gives a simplified version of the actual algorithm implemented in our tools. Indeed, in this simple form, the algorithm may not return the optimal result as the search stops as soon as one collision has been found between the two subgraphs $\mathcal{G}_0$ and $\mathcal{G}_1$ (Line 8). We implement instead what can be seen as a bi-directional Dijsktra's shortest path finding algorithm, which continues executing the main while-loop of Line 5 to keep expanding the subgraphs until the current best implementation cannot be further improved.

We give a visual example of the graph construction in Figure 1.

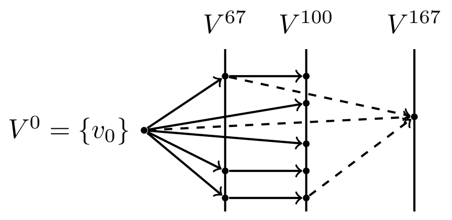

Figure 1: Example of a graph constructed by Expand. The set $\mathcal{B}$ contains three hypothetical instructions of cost 67, 100 and 167, which implies that nodes at distance 167 from the root v_0 can be reached from nodes in V^{100}, V^{67}, or V^0.

3.2 Efficient $\mathcal{B}$-Implementations

The meet-in-the-middle algorithm presented in the previous section outputs optimal $\mathcal{B}$-implementations of general functions $f : \mathrm{GF}(2^c) \to \mathrm{GF}(2^c)$, $c > 0$. However in practice, the computational and memory complexities may restrict its application only to small functions (e.g., $c \leq 4$), and do not apply to *all* functions for higher c (although in *some* particular cases, the algorithm can terminate, for instance for linear functions). In case the algorithm does not terminate, no solution is returned. The algorithm presented below overcomes this limitation.

The main idea is to start from a heuristically found $\mathcal{B}$-implementation of the target function f and iteratively reduce it locally using the previous MitM optimal algorithm for some finite limit value Λ. We assume in this section that a $\mathcal{B}$-implementation of a function f is known, and concentrate on minimizing it. We partially address the problem of finding one initial implementation in Section 4.

In the sequel, we assume that the known $\mathcal{B}$-implementation $\mathcal{I}^f$ equals the sequence $((f_i, o_i))_{i=1,\ldots,n}$. To find another implementation $\mathcal{I}'^f$ of f such that $\|\mathcal{I}'^f\| \leq \|\mathcal{I}^f\|$, we start by fixing a bound $\Lambda \geq 2$ for the time spent in the underlying meet-in-the-middle algorithm MitM. Then, we recursively consider all the decompositions of $\mathcal{I}^f$ by splitting it in parts containing between two and Λ instructions. For all the partial implementations $\mathcal{I}_{f_\alpha}^{f_\beta}$ of length at most Λ, we apply the previous algorithm with the Λ parameter: $\textsc{MitM}(f_\alpha, f_\beta, \Lambda)$. As we know there exists at least one $\mathcal{B}$-implementation transforming f_α into f_β in at most Λ instructions, this call necessarily terminates and may produce an implementation with a smaller cost. Then, for all the decompositions, the algorithm selects the one that minimizes the overall cost. We repeat this process as long as the implementation cost keeps reducing and return it if no further improvement occurred.

We give a simplified pseudo-code description of this algorithm that we call ChainMitM in the following Algorithm 2. The initial call to decrease the implementation cost of $\mathcal{I}$ is $\textsc{ChainMitM}(\mathcal{I}, \Lambda)$, which subsequently calls the recursive function Split. In case of success, the returned implementation has a cost strictly smaller than that of $\mathcal{I}$, otherwise, $\mathcal{I}$ is returned.

3.3 Graph Vertex and Edge Representations

We specify in this section details about the representation of the functions serving as nodes in the graph. We differentiate the general case where the vertices are all functions $\mathrm{GF}(2^c) \to \mathrm{GF}(2^c)$ from the situation where we restrict the space of functions to linear ones only. While the same algorithms apply for either type, the representation of linear functions can be more compact. In both cases, we also specify how the functions are linked together by the edges representing the logical instructions.

General Functions. To encode a function $f : \mathrm{GF}(2^c) \to \mathrm{GF}(2^c)$, at least $c \cdot 2^c$ bits are required as there are $2^{c \cdot 2^c}$ different functions having this signature. We use the technique called *bit-slicing*, which provides a natural way to optimally encode such functions. The idea is to see the function as a vectorial Boolean function and encodes the truth table of every c Boolean functions independently.

For example, as seen in Table 1 in the case of the `PRESENT` Sbox S (a permutation over $\mathrm{GF}(2^4)$), we can see S as a function $x \in \mathrm{GF}(2^4) \to (y_3, y_2, y_1, y_0) \in (\mathrm{GF}(2))^4$ and simply encode the c truth tables $y_i : \mathrm{GF}(2^4) \to \mathrm{GF}(2)$ as 2^c-bit words. In this example, we would represent S as the 4-tuple of 16-bit words (`0x9b70`, `0xe16c`, `0x32e5`, `0x59a6`). Similarly, the identity permutation over $\mathrm{GF}(2^4)$ would be represented by (`0x00ff`, `0x0f0f`, `0x3333`, `0x5555`).

Having the function encoded in this form allows to perform a single Boolean instruction on all input values from the domain at once. That way, a tuple $v_f = (v_f^1, v_f^2, \ldots, v_f^c)$ representing function $f : \mathrm{GF}(2^c) \to \mathrm{GF}(2^c)$ could be transformed to $(o(v_f^1, v_f^2), v_f^2, \ldots, v_f^c)$

Algorithm 2 – Improve Heuristic Implementation.

```
function ChainMitM($\mathcal{I}, \Lambda$)
  do
    $m \leftarrow \|\mathcal{I}\|$
    $\mathcal{I} \leftarrow$ Split($\mathcal{I}, \Lambda$)
  while $\|\mathcal{I}\| < m$
  return $\mathcal{I}$                          ▷ Return a possibly optimized implementation
```

```
function Split($\mathcal{I}_{f_\alpha}^{f_\beta}, \Lambda$)
  $\mathcal{I} \leftarrow \mathcal{I}_{f_\alpha}^{f_\beta}$                          ▷ Try to find a sequence smaller than $\mathcal{I}$
  $((f_i, o_i))_i \leftarrow \mathcal{I}$
  $l \leftarrow |\mathcal{I}|$                                   ▷ Length of the sequence
  $m \leftarrow \|\mathcal{I}\|$                                 ▷ Current best cost
  if $l \leq \Lambda$ then return MitM($f_\alpha, f_\beta, \Lambda$)       ▷ Base case
  for $\lambda = 2, \ldots, \Lambda$ do                          ▷ General case
    for $t = 0, \ldots, l - \lambda$ do
      $\mathcal{I}_0 \leftarrow$ Split($\mathcal{I}_{f_\alpha}^{f_{\alpha+t}}, \lambda$)
      $\mathcal{I}_1 \leftarrow$ Split($\mathcal{I}_{f_{\alpha+t}}^{f_{\alpha+t+\lambda}}, \lambda$)
      $\mathcal{I}_2 \leftarrow$ Split($\mathcal{I}_{f_{\alpha+t+\lambda}}^{f_\beta}, \lambda$)
      if $\|\mathcal{I}_0\| + \|\mathcal{I}_1\| + \|\mathcal{I}_2\| < m$ then
        $\mathcal{I} \leftarrow \mathcal{I}_0 + \mathcal{I}_1 + \mathcal{I}_2$          ▷ New shorter sequence found
        $m = \|\mathcal{I}\|$
  return $\mathcal{I}$                          ▷ Return a possibly optimized implementation
```

Table 1: Bit-sliced representation of the `PRESENT` Sbox $S : x \in \mathrm{GF}(2^4) \rightarrow (y_3, y_2, y_1, y_0) \in (\mathrm{GF}(2))^4$ used in our algorithms.

$S(x)$	12	5	6	11	9	0	10	13	3	14	15	8	4	7	1	2	Hex
y_3	1	0	0	1	1	0	1	1	0	1	1	1	0	0	0	0	`0x9b70`
y_2	1	1	1	0	0	0	0	1	0	1	1	0	1	1	0	0	`0xe16c`
y_1	0	0	1	1	0	0	1	0	1	1	1	0	0	1	0	1	`0x32e5`
y_0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	`0x59a6`

by applying the Boolean instruction $v_f^1 \leftarrow o(v_f^1, v_f^2)$. We give in the following Table 2 an example of the application of an XOR instruction.

In the following Section 4 on heuristics, we precise which specific instructions $\mathcal{B}$ we used in our search of non-linear function implementations.

Linear Functions. In the more particular case where the functions $f : \mathrm{GF}(2^c) \rightarrow \mathrm{GF}(2^c)$ are known to be linear, less than $c \cdot 2^c$ bits are required to encode f, as one can simply encode f as a binary matrix of order c. Therefore, only c^2 bits are required to encode all linear functions defined over $\mathrm{GF}(2^c)$.

Then, the graphs used in the algorithms only work with linear functions, and the set of instructions $\mathcal{B}$ only contains linear Boolean operations as well. Again, this representation enables to apply a single instruction to all the values of the domain, however in this case, one implementation simply encodes row matrix operations transforming one end of the path to the other. For the multiplication of finite field element, it can simply be represented

Table 2: Bit-sliced representation of the identify function Id, and example of an application of the instruction $y_0 \leftarrow y_0 \oplus y_1$.

$Id(x)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Hex
y_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0x00ff
y_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0x0f0f
y_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0x3333
y_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0x5555
y_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0x00ff
y_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0x0f0f
y_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0x3333
$y_0 \oplus y_1$	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0x6666

by its multiplication matrix as seen in Example 2.

Successors and Branching. In the Algorithm 1 previously described, the function $\textsc{Succ}(v, o)$ is used to generate the set of all functions reachable from v using the Boolean instruction $o \in \mathcal{B}$. Although the set of successors highly depends on the elements present in $\mathcal{B}$, we can still give an intuition about its cardinality. Indeed, what one usually call *branching* represents the number of successors per node in the graph (related to the term *out-degree* in graph theory).

For 2-variable instructions (e.g., $x \leftarrow x \oplus y$ or $y \leftarrow x \oplus y$), the set of successors for a function $f : \mathrm{GF}(2^c) \rightarrow \mathrm{GF}(2^c)$ would contain $c(c-1)$ functions. Depending on the instruction considered, the branching can be higher or smaller.

4 Heuristics

We explicit in this section several heuristics we use throughout our work. One can split them into two categories: first, the heuristic algorithms we use to generate $\mathcal{B}$-implementations of a given function (either linear or non-linear). This first type of heuristic is only used when the MitM algorithm (Algorithm 1) targeting optimal $\mathcal{B}$-implementations fails.

Then, we describe the restriction on the possible instructions due to the meet-in-the-middle nature of our graph algorithms. Indeed, since the implementations are evaluated in both direct and indirect directions, we heuristically impose the instructions in $\mathcal{B}$ to be invertible. While this constraint naturally means that the $\mathcal{B}$-implementations found by our algorithms necessarily have a cost higher or equal than the overall optimal implementation under the same metric (for any possible instruction set $\mathcal{S}$, $\mathcal{B} \subseteq \mathcal{S}$), we nevertheless observe that in practice, this heuristic already provides very good results.

4.1 Heuristics for Linear Layers

Heuristic Implementations for Linear Layers. To apply the ChainMitM algorithm (Algorithm 2), we need an initial implementation as input for the optimization process. If a starting implementation is unknown, we need a deterministic method to find it. When the function f to implement is linear, we can encode it in the form of a binary matrix $\mathbf{M}_f$ as seen before, and then finding an implementation of f reduces to finding a sequence of instructions transforming $\mathbf{M}_f$ to the identity matrix. In the sequel, we assume $\mathbf{M}_f$ to be invertible.

The problem of finding an initial implementation for a binary matrix can easily be solved using the Gauss-Jordan elimination (GJE) method, a long-standing algorithm in linear algebra used to solve systems of linear equation. Indeed, as the matrix is invertible, it can be regarded as a homogeneous system of linear equations with an unique solution. One refers to [30] for a description of the GJE.

In the GJE algorithm, there is no fixed rule on the choice of the row and pivot to perform the reduction. This freedom implies there are possibly several implementations for the same function and allows to choose one that minimizes the number of instructions. Hence, we exhaust all possible choices of row order and pivot point to search for the least number of XOR instructions needed to reduce a binary matrix to a row permutation of the identity matrix.

Example 3. Suppose we consider the linear function $f : \mathrm{GF}(2^4) \rightarrow \mathrm{GF}(2^4)$ encoded by the following binary matrix:

$$\underbrace{\begin{bmatrix}1&1&0&1\\1&1&1&0\\1&1&1&1\\1&0&1&0\end{bmatrix}}_{\mathbf{M}_f}\begin{bmatrix}b_3\\b_2\\b_1\\b_0\end{bmatrix} = \begin{bmatrix}b_3\oplus b_2\oplus b_0\\ b_3\oplus b_2\oplus b_1\\ b_3\oplus b_2\oplus b_1\oplus b_0\\ b_3\oplus b_1\end{bmatrix}.$$

where (b_3, b_2, b_1, b_0) is an arbitrary element of $(\mathrm{GF}(2))^4$.

However, by applying the GJE on the matrix, we find a sequence of 4 XOR operations that simplifies the multiplication matrix to a row permutation of the identity matrix. The instructions are $R_0 \leftarrow R_0 \oplus R_2$, $R_2 \leftarrow R_2 \oplus R_1$, $R_1 \leftarrow R_1 \oplus R_3$ and $R_3 \leftarrow R_3 \oplus R_0$, which gives the permutation $\pi = (3, 1, 0, 2)$ of the rows of the identity matrix.

Consequently, to implement the given binary matrix $\mathbf{M}_f$, we first start by applying π, followed by the reverse sequence of XOR instructions, and finally obtain the output vector, that is:

$$\begin{bmatrix}b_3\\b_2\\b_1\\b_0\end{bmatrix} \xrightarrow{\pi} \begin{bmatrix}b_1\\b_2\\b_0\\b_3\end{bmatrix} \xrightarrow{R_3\leftarrow R_3\oplus R_0} \begin{bmatrix}b_1\\b_2\\b_0\\b_3\oplus b_1\end{bmatrix} \xrightarrow{R_1\leftarrow R_1\oplus R_3} \begin{bmatrix}b_1\\b_3\oplus b_2\oplus b_1\\b_0\\b_3\oplus b_1\end{bmatrix}$$

$$\xrightarrow{R_2\leftarrow R_2\oplus R_1} \begin{bmatrix}b_1\\b_3\oplus b_2\oplus b_1\\b_3\oplus b_2\oplus b_1\oplus b_0\\b_3\oplus b_1\end{bmatrix} \xrightarrow{R_0\leftarrow R_0\oplus R_2} \begin{bmatrix}b_3\oplus b_2\oplus b_0\\b_3\oplus b_2\oplus b_1\\b_3\oplus b_2\oplus b_1\oplus b_0\\b_3\oplus b_1\end{bmatrix},$$

This implementations yields 4 XORs which is smaller than $d\text{-}XOR(\mathbf{M}_f) = 8$. □

Tweaking Gauss-Jordan Elimination Method. In GJE method, one may notice that when a pivot is chosen from some row, that row is continuously used to update the other rows. After which, that row will never be used again to update other rows. Therefore, there are instances where GJE method would not be optimal (with respect to the number of instructions).

In [11], Bernstein presented an algorithm (we call it DJB method) that also achieves optimized implementations of linear functions. However, similarly to the GJE, due to the nature of the algorithm there are instances where the output sequence is not optimal. To find the shortest sequence, we could apply both algorithms and pick the best one. However, we observe a simple way to simulate the behavior of DJB from the GJE strategy. Namely, one needs to apply the GJE method on the transpose of the given matrix. In contrast to the sequence obtained from GJE, a chosen row will be continuously updated by other rows.

Let $\mathbf{E}_{i,j}$ be the identity matrix of order n with an additional '1' on the i-th row and j-th column, where $i, j \in \{1, 2, ..., n\}$. Each XOR instruction can be represented by a left-multiplication of some $\mathbf{E}_{i,j}$ to an n-tuple column vector $(x_1, x_2, ..., x_n)^\top$, it is basically updating the vector component $x_i \leftarrow x_i \oplus x_j$ to the column vector. On the other hand, if a row vector is right-multiplied by the same matrix, the update becomes $x_j \leftarrow x_j \oplus x_i$.

Since a given invertible binary matrix $\mathbf{M}$ can be decomposed into a sequence of XOR operations and a bit permutation, we can express it as

$$\mathbf{M} = \mathbf{E}_{i_1,j_1}\mathbf{E}_{i_2,j_2}\cdots\mathbf{E}_{i_s,j_s}\mathbf{P},$$

where $\mathbf{P}$ is a permutation matrix. Let $\mathbf{u}$ be a column vector, the left multiplication yields:

$$\mathbf{M}\mathbf{u} = \mathbf{E}_{i_1,j_1}\left(\mathbf{E}_{i_2,j_2}\cdots(\mathbf{E}_{i_s,j_s}(\mathbf{P}\mathbf{u}))\cdots\right).$$

On the other hand, starting from the transpose matrix $\mathbf{M}^\top$, we can also rewrite it using the GJE as:

$$\mathbf{M}^\top = \mathbf{E}_{i'_1,j'_1}\mathbf{E}_{i'_2,j'_2}\cdots\mathbf{E}_{i'_t,j'_t}\mathbf{P}',$$

for another matrix permutation $\mathbf{P}'$. By rewriting the multiplications, this gives

$$\begin{aligned}\mathbf{M}\mathbf{u} = \left(\mathbf{u}^\top\mathbf{M}^\top\right)^\top &= \left((\cdots((\mathbf{u}^\top\mathbf{E}_{i'_1,j'_1})\mathbf{E}_{i'_2,j'_2})\cdots\mathbf{E}_{i'_t,j'_t})\mathbf{P}'\right)^\top \\ &= \mathbf{P}'^\top(\mathbf{E}_{j'_t,i'_t}\cdots(\mathbf{E}_{j'_2,i'_2}(\mathbf{E}_{j'_1,i'_1}\mathbf{u}))\cdots),\end{aligned}$$

where $\mathbf{P}'^\top$ also describes a permutation matrix. Hence, the decomposition of the transpose matrix $\mathbf{M}^\top$ can also be used to derive an implementation of the matrix $\mathbf{M}$.

As a result, to determine an initial implementation of a given invertible binary matrix $\mathbf{M}_f$ (encoding a linear function f), we simply apply the GJE algorithm to both $\mathbf{M}_f$ and $\mathbf{M}_f^\top$ and choose whichever sequence of XOR instructions is shorter.

4.2 Heuristics for Non-Linear Layers

Instruction Restriction in $\mathcal{B}$. Unlike the linear instructions used to implement linear functions, non-linear Boolean operations like NAND are not invertible. Consequently, it is non-trivial to use instructions relying on those operations as an edge in our MITM technique as we want to grow the graph in the backward direction too. Using non-linear instructions induces a very high branching in the graph associated to the backward expansion, and quickly reaches the practical memory complexity limit. Therefore, we require special invertible instructions in $\mathcal{B}$ for the edges of the graph: these instructions are built from combinations of linear and non-linear operations.

The general rules for creating the invertible instructions are as follows.

1. Only linear instructions are used to update the bits.
2. Non-linear instructions are only used to create temporary values from the bits.
3. Temporary values are used to update bits that are not used to create these temporary values.

The only invertible instructions involving a single variable is $x \leftarrow \neg x$ (NOT). For 2-variable instructions, the instructions are necessarily linear to be invertible, for instance $x \leftarrow x \oplus y$ (XOR) and $x \leftarrow \neg(x \oplus y)$ (XNOR). For instructions of three or more variables, we can compute any non-linear Boolean operation on all but one variable, store the output bit in a temporary variable, and use it to linearly mask the last variable.

To give a concrete example, one set $\mathcal{B}$ of instructions that could be used in the graph-based algorithms described is:

$$x \leftarrow \neg x, \qquad x \leftarrow x \oplus y, \qquad z \leftarrow \neg(x \wedge y) \oplus z, \qquad z \leftarrow \neg(x \vee y) \oplus z.$$

The main benefit of using such instructions is the reduced branching of the graph expansion in the backward direction. In addition, the inverse function can be easily implemented by simply reversing the instruction sequence. Examples of such construction appear in the `PICCOLO` Sbox [41], and in the `SKINNY` family [9].

Finding an Implementation for the Non-linear Layer. Similar to the linear layer, when the MitM algorithm cannot output an optimal $\mathcal{B}$-implementation, it is possible to start with an initial $\mathcal{B}$-implementation and apply the second ChainMitM algorithm. We devised a deterministic algorithm to reduce a given Sbox to an identity mapping using instructions conforming to the previous rules, however, for all the 4-bit Sboxes that we have tested, optimal $\mathcal{B}$-implementations could be found using MITM technique. This heuristic algorithm is not implemented in our current tool but investigating this direction might serve as basis for future works.

5 Results on Linear Layers

We present in this section the results of our search on linear layer consisting of MDS matrices. First, we detail the application of the graph-based search for (possibly optimal) implementations of the individual field multiplications $x \to \alpha x$, for all non-trivial element α in the field (Section 5.1). Next, we discuss about some of the recent work on MDS diffusion matrix search (Section 5.2), and describe our search parameters and some strategies for searching new lightweight MDS diffusion matrices (Section 5.3). Finally, in the rest of the section, we give the concrete implementations we obtained for several new and known MDS matrices (Section 5.4).

5.1 Implementations of Individual Field Multiplications

We ran the MitM algorithm to search for the optimal implementation for field elements over $\mathrm{GF}(2^4)$ and $\mathrm{GF}(2^8)$ defined by all possible irreducible polynomials of degree 4 and 8. For $\mathrm{GF}(2^4)$, the results show that all elements can be implemented with no more than 5 XOR operations. For $\mathrm{GF}(2^8)$, our MitM technique could reach up to $\Lambda = 12$ XOR instructions in reasonable time and memory complexities. To give concrete numbers, this search took a few hours and about 32-64 GB of RAM on a 16-core machine. For elements that have no solution within 12 XOR instructions, we applied the GJE method on both their multiplication matrix and its transpose to find an initial sequence and used ChainMitM algorithm with parameter $\Lambda = 12$ to further improve the implementation and obtain a (sub)optimal sequence.

Although the implementation of some field elements are not optimal, we can already observe the following proposition.

Proposition 1. *For any nonzero multiplication matrix* $\mathbf{M}_\alpha$ *of the field element* $\alpha \in \mathrm{GF}(2^c)$*, where* $c = 4$ *or* $c = 8$*, there exists an XOR-implementation of* $\mathbf{M}_\alpha$ *such that s-XOR*$(\alpha) \leq$ *d-XOR*(α)*.*

We checked for all the irreducible polynomials for $\mathrm{GF}(2^4)$ and $\mathrm{GF}(2^8)$ that the proposition holds, and the previous Example 2 shows a case where the inequality is strict. Besides, we conjecture that this property also holds for any invertible binary matrix of arbitrary order; that is:

Conjecture 1. *Let* c *be a positive integer and* $\mathbf{M} \in \mathrm{GL}(c, \mathrm{GF}(2))$*. Prove that there exists an XOR-implementation of* $\mathbf{M}$ *such that: s-XOR*$(\mathbf{M}) \leq$ *d-XOR*$(\mathbf{M})$*.*

In summary, the s-XOR count of the field elements is smaller or equal than its d-XOR count. This shows that in practice, most of the field elements can actually be implemented with less XOR gates than previously known. The implementations of the field elements (except trivial elements 0 and 1) from $\mathrm{GF}(2^4)$/`0x13` are presented in Appendix A.1.

After obtaining the lists of s-XOR counts of elements in $\mathrm{GF}(2^4)$ and $\mathrm{GF}(2^8)$ over all possible irreducible polynomials, we search for new lightweight MDS diffusion matrices based on this new metric. In this paper, we use the terms *d-lightest* and *s-lightest* to describe matrices that have the least XOR count under the d-XOR and the s-XOR metrics, respectively.

In recent years, there are several papers presenting different ways to search for new lightweight MDS diffusion matrices. In the following section, we discuss the strengths and limitations of their methodologies. Next, we present our strategy to search for lightweight MDS diffusion matrices.

5.2 Previous MDS Diffusion Matrix Searches

To the best of our knowledge, the authors of [29] were the first to propose the d-XOR metric. In addition, they proposed the sub-field construction to design a lightweight MDS matrix over some finite field $\mathbb{K}$ using a matrix of the same order but over some smaller sub-field $\mathbb{L}$ of $\mathbb{K}$. The idea of the sub-field technique is rather simple: to construct an MDS matrix over finite field $\mathrm{GF}(2^{mc})$, we use m copies of an MDS matrix[5] over $\mathrm{GF}(2^c)$. Hence, the implementation cost of the matrix over $\mathrm{GF}(2^{mc})$ is m times the cost of the matrix over $\mathrm{GF}(2^c)$.

In [32, 42], the authors proposed compact equivalence classes (CEC) of Hadamard and circulant matrices to reduce the exhaustive search space on these two types of matrices and presented the d-lightest MDS Hadamard and circulant matrices over $\mathrm{GF}(2^4)$ and $\mathrm{GF}(2^8)$. Using these equivalence classes, they could complete the search for MDS Hadamard and circulant matrices of order 8, which was previously intractable.

In [31], the authors extended the coefficients of a diffusion matrix from finite field $\mathrm{GF}(2^c)$ to invertible binary matrices $\mathrm{GL}(c, \mathrm{GF}(2))$, where $c \in \{4, 8\}$, and found new lightweight MDS non-involution and involution 4×4 Hadamard and circulant matrices over $\mathrm{GL}(c, \mathrm{GF}(2))$. However, it is non-trivial to extend their search on matrices of order 8, essentially because the cardinality of $\mathrm{GL}(c, \mathrm{GF}(2))$ is much larger than that of $\mathrm{GF}(2^c)$: $2^{c\cdot(c-1)/2} \prod_{i=1}^{c} (2^i - 1) \gg 2^c$.

Using our s-XOR metric, the authors [10] constructed lightweight MDS 4×4 and 8×8 left-circulant matrices using a single field element α. Focusing on a single field element allows them to explore all possible bases for representing finite field elements and search for one with the minimum s-XOR count, while in the conventional way, as we did in this paper, the polynomial basis is used to represent the finite field. The limitation of their methodology consists in the structure of the matrices, that must have elements of the form $\alpha^{\pm n}$ for a field element a, and for small n, say $n \leq \frac{k}{2}$. Consequently, they consider only a subclass of all the circulant matrices.

Recently, another paper [53] adopted our s-XOR metric to improve on the implementation of the `AES` diffusion matrix. They considered the `AES` diffusion matrix as a 32×32 binary matrix and applied a heuristic algorithm to find a sequence of XOR instructions. However, their approach is different from ours as we focus on optimizing the implementation of individual elements, which also allow us to construct new lightweight diffusion matrices.

In [40], the authors searched for lightweight MDS Toeplitz matrices over $\mathrm{GF}(2^4)$ and $\mathrm{GF}(2^8)$ based on the d-XOR metric. The advantage of Toeplitz structure over Hadamard and circulant is the larger degree of freedom for the choice of the coefficients. In addition, their empirical evidence showed that their MDS 4×4 Toeplitz matrices over $\mathrm{GF}(2^4)$ and $\mathrm{GF}(2^8)$ are the d-lightest possible MDS non-involution matrices over $\mathrm{GF}(2^4)$ and $\mathrm{GF}(2^8)$.

[5] One may also use different MDS matrices, the implementation cost is simply the sum of the cost of the matrices.

However, there are two limitations to the use of Toeplitz matrices. First, as proven by the authors, MDS involution Toeplitz matrices of order 2^m does not exist, which is a widespread practical choice for the order of diffusion matrices. Second, the search for an MDS Toeplitz matrix of order 8 seems intractable due to the large search space.

5.3 Our Search for Lightweight MDS Diffusion Matrices

Our search parameters for the lightweight MDS matrices have three dimensions: non-involution or involution matrices, matrices of order 4 or 8, and matrix coefficients over $\mathrm{GF}(2^4)$ or $\mathrm{GF}(2^8)$. These criteria have been chosen because they capture classical parameters for the diffusion matrices used in practice.

When we search for lightweight MDS matrices, we start with some threshold value, τ, for the cost of the matrix, say the XOR count of some existing lightweight MDS matrix, and search for MDS matrices that have lower cost than this threshold value. A simple search pruning strategy is to arrange the field elements in the increasing order of its s-XOR. When we pick an element for some coefficient, we check if the sum of the s-XOR of elements in the matrix exceeds the threshold value, if it does, we do not need to consider the remaining elements for that coefficient. In addition, recall that all coefficients of an MDS matrix are nonzero, thus we do not consider element zero as a coefficient of the matrix.

4×4 MDS (Involution) Diffusion Matrices Over $\mathrm{GF}(2^4)$. To search for the s-lightest MDS non-involution and involution 4×4 matrices over $\mathrm{GF}(2^4)$, we do not put restriction on the shape of the matrix (e.g., Hadamard, circulant, etc.). Instead, we apply an improved exhaustive search over the entire matrix space. Although the entire space of 4×4 matrices over $\mathrm{GF}(2^4)$ counts as many as 2^{64} elements, there are a couple of early-abort strategies that we can use to discard invalid candidates prematurely, including the pruning of field elements based on their s-XOR mentioned before.

The exhaustive search algorithm, ExhaustiveSearch, uses nested for-loops to enumerate the coefficients of the matrix in the following order:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 8 & 9 & 10 \\ 6 & 11 & 12 & 13 \\ 7 & 14 & 15 & 16 \end{bmatrix},$$

where coefficients with low indices are enumerated first.

When we select an element for the 8th coefficient, we can compute the determinant of the 2×2 submatrix (with 1st, 2nd, 5th and 8th coefficients). If it is zero, we already know that any choice for the remaining coefficients will not form an MDS matrix. Hence, we can directly discard this element and pick another. Otherwise, if this submatrix is non-singular, then we continue to pick an element for the next coefficient. Similarly, we check the determinant of all the square submatrices (of any sizes) involving that coefficient. This greatly reduces the search space as we prematurely discard many combinations of coefficients. When all the square submatrices are non-singular (including the entire 4×4 matrix), we have found an MDS matrix with lower cost than a threshold value. We store that matrix, update this threshold value and continue the search. At the end of the algorithm, we obtain the s-lightest MDS 4×4 matrix over $\mathrm{GF}(2^4)$.

For involution matrices, we have an additional condition that the dot product of the i-th row and j-th column of the matrix has to be one if $i = j$, and zero otherwise. Hence, when we pick an element for the 7-th coefficient, we can check whether it is a possible candidate for involution matrix. If the dot product of the 1st row and 1st column differs from one, we can discard this element and pick another. The subsequent checks occur for

Algorithm 3 – Lightest MDS Diffusion Matrix Search.

```
1: function MDSMatrixSearch($\mathcal{C}, \mathbb{K}, k, \tau, inv$)
2:   if ($\dim(\mathbb{K}) \le 4$ and $k \le 4$) then
3:     $M_{lightest} \leftarrow$ ExhaustiveSearch($\mathcal{C}, \mathbb{K}, k, \tau, inv$)
4:   else
5:     $M_{lightest} \leftarrow$ CirHadSearch($\mathcal{C}, \mathbb{K}, k, \tau, inv$)
6:   return $M_{lightest}$
```

```
1: function CirHadSearch($\mathcal{C}, \mathbb{K}, k, \tau, inv$)
2:   $M_{lightest} \leftarrow \emptyset$
3:   for all $S = \{x_0, \ldots, x_{k-1}\} \subset \mathbb{K}$ do     ▷ For all possible multisubsets of $\mathbb{K}$
4:     if $\|S\| < \tau$ then                    ▷ Cost of the elements are obtained from $\mathcal{C}$
5:       $Clist \leftarrow$ GenCir($S, inv$)          ▷ Generate CEC rep. of circulant matrices
6:       for all $M \in Clist$ do
7:         if CheckMDS($M$) then
8:           $\tau \leftarrow \|S\|$
9:           $M_{lightest} \leftarrow M$
10:      $Hlist \leftarrow$ GenHad($S, inv$)          ▷ Generate CEC rep. of Hadamard matrices
11:      for all $M \in Hlist$ do
12:        if CheckMDS($M$) then
13:          $\tau \leftarrow \|S\|$
14:          $M_{lightest} \leftarrow M$
15:  return $M_{lightest}$
```

the 10th, 13th, 14th, 15th and 16th coefficients. If all dot products of rows and columns satisfy the condition, the algorithm outputs the s-lightest MDS 4×4 involution matrix over $\mathrm{GF}(2^4)$.

We have implemented this algorithm, and the search for both non-involution and involution arbitrary matrices took less than a minute on a personal computer.

Other Parameters of Diffusion Matrices. Scaling the algorithm for general 8×8 matrices over $\mathrm{GF}(2^4)$ seems intractable as the input space contains 2^{256} elements. However, it has been experimentally verified by the authors from [32] that there is no MDS 8×8 circulant matrix over $\mathrm{GF}(2^4)$. Therefore, we detail in the following the result of our search on Hadamard matrices for both MDS non-involution and involution matrices over $\mathrm{GF}(2^4)$.

For matrices over $\mathrm{GF}(2^8)$, in addition to searching for lightweight MDS Hadamard and circulant matrices, we use the sub-field technique to construct MDS matrices from the s-lightest matrices over $\mathrm{GF}(2^4)$.

Using the concept of CEC of Hadamard and circulant matrices, the search space for lightweight MDS Hadamard and circulant matrices reduces significantly. This allows to perform an exhaustive search for these. The details of the CEC of Hadamard and circulant matrices can be found in [42] and [32], respectively.

We give a simplified pseudo-code description of the algorithm (Algorithm 3) used for searching lightweight MDS matrices. The MDSMatrixSearch function takes a list $\mathcal{C}$ containing the implementation cost of the field elements, a finite field $\mathbb{K} = \mathrm{GF}(2^c)/p(X)$, matrix order k, threshold value τ, and Boolean value for involution matrix inv. For our work, we focus on field dimensions 4 or 8, and matrix orders 4 or 8. If both the field dimension and matrix order are 4, we can apply the function ExhaustiveSearch as described above. Otherwise, we apply function CirHadSearch. First, a (multi)subset S

of k field elements, whose summation of the implementation costs is less than τ, is selected. Next, S is used to generate the potential MDS matrix candidates using the functions GenCir and GenHad. When the necessary condition for the candidate S to be MDS is not satisfied, the function returns an empty set. For instance, it is known that involution MDS circulant matrices do not exist [26], therefore whenever *inv* holds true, GenCir returns an empty set. We omitted the details of the functions GenCir, GenHad and CheckMDS as the conditions for MDS and construction of the CEC representatives of circulant and Hadamard matrices are discussed extensively in [32, 42]. When a candidate matrix is found to be MDS, we store it, update the threshold value and continue the search. At the end of the algorithm, we obtain the s-lightest possible MDS matrix of the given parameter. A simple early-abort strategy is to select subsets with the least possible implementation cost, and gradually increase the cost when no MDS matrix is found. The algorithm can be terminated prematurely once an MDS matrix is found.

5.4 Results on Linear Layers

In a nutshell, among the $2^3 = 8$ categories of MDS diffusion matrix (that consist of non-involution/involution matrices, matrices of order 4/8, and matrices over $\mathrm{GF}(2^4)/\mathrm{GF}(2^8)$), we found new lightweight MDS matrices which outperform existing lightweight diffusion matrices in four categories, and we improved the implementation of some existing lightweight matrices and obtained smaller s-XOR counts in three categories. The case of non-involution 4×4 matrices over $\mathrm{GF}(2^8)$ is the only one we do not improve: the best result is from [31].

Recall that we are interested in minimizing the implementation costs of the field multiplication matrices. We therefore only state the sum of the implementation costs of the coefficients plus the connecting XORs in one row, and if the rows have different implementation costs, we take the average cost for one row. For instance, if a 4×4 MDS matrix over $\mathrm{GF}(2^4)$ costs $2.5 + 12$, it means that the average cost of the coefficients in one row is 2.5, there are 12 connecting XORs in each row and the total cost is 58. We also include the cost of its inverse matrix for reference. However, we emphasize that for non-involution matrices, our searches focus on optimizing the direct cost.

In the following, we detail the results of our search in four paragraphs, first for matrices over $\mathrm{GF}(2^4)$ and then over $\mathrm{GF}(2^8)$.

Lightweight MDS 4×4 Matrices Over $\mathbf{GF(2^4)}$. In our exhaustive search for the s-lightest MDS 4×4 arbitrary matrices over $\mathrm{GF}(2^4)$ (described in Section 5.3), we found that the s-lightest matrix $\mathbf{M}_{4,n,4}$,

$$\mathbf{M}_{4,n,4} = \begin{bmatrix} \texttt{0x1} & \texttt{0x1} & \texttt{0x1} & \texttt{0x2} \\ \texttt{0x1} & \texttt{0x2} & \texttt{0xd} & \texttt{0x1} \\ \texttt{0x2} & \texttt{0xd} & \texttt{0x1} & \texttt{0x1} \\ \texttt{0xd} & \texttt{0x1} & \texttt{0x2} & \texttt{0x1} \end{bmatrix},$$

outperforms the circulant matrix presented in [10]. As mentioned before, we found this new matrix by an improved exhaustive search over the full space $\mathrm{GL}(4, \mathrm{GF}(2^4))$, while the search from [10] relies on particular matrices in this space. Similarly for MDS involution 4×4 matrices over $\mathrm{GF}(2^4)$, we found the s-lightest involution arbitrary matrix $\mathbf{M}_{4,i,4}$,

$$\mathbf{M}_{4,i,4} = \begin{bmatrix} \texttt{0x2} & \texttt{0x1} & \texttt{0x1} & \texttt{0x9} \\ \texttt{0x1} & \texttt{0x4} & \texttt{0xf} & \texttt{0x1} \\ \texttt{0xd} & \texttt{0x9} & \texttt{0x4} & \texttt{0x1} \\ \texttt{0x1} & \texttt{0xd} & \texttt{0x1} & \texttt{0x2} \end{bmatrix}.$$

It is known that MDS involution circulant matrices do not exist over any finite field $\mathrm{GF}(2^n)$. Therefore, Hadamard matrices are commonly used to construct MDS involution matrices. An example of such a matrix can be found in the linear layer of `Joltik`, and under our s-XOR metric, the results of our algorithms optimize the implementation of the coefficients in this matrix.

The results are summarized in Table 3. As mentioned in Section 5, the authors of [10] considered the representation of the field element under all possible bases. In their paper, only the multiplication matrix of the field element is provided but not the irreducible polynomial. Therefore in the table, we only state the size of the finite field but not the irreducible polynomial.

Table 3: Comparison of 4×4 MDS matrices over $\mathrm{GF}(2^4)$ and $\mathrm{GL}(4, \mathrm{GF}(2))$.

Matrix			**Implementation**					**Ref.**
Field/Ring	**Type**	**Inv.**	**d-XOR(M)**	**s-XOR(M)**	**s-XOR($\mathbf{M}^{-1}$)**	**Opt**	**Min**	
$\mathrm{GF}(2^4)$/`0x13`	Arbitrary	No	$2.5 + 12$	$2.5 + 12$	$9.25 + 12$	✓	s	$\mathbf{M}_{4,n,4}$
$\mathrm{GF}(2^4)$/`0x19`	Toeplitz	No	$2.5 + 12$	$2.5 + 12$	$11.5 + 12$	✓	d	[40]
$\mathrm{GL}(4, \mathrm{GF}(2))$	Circulant	No	$3 + 12$	$3 + 12$	–	✓	–	[31]
$\mathrm{GF}(2^4)$	Circulant	No	$4 + 12$	$3 + 12$	–	–	–	[10]
$\mathrm{GF}(2^4)$/`0x13`	Arbitrary	Yes	$5 + 12$	$3.75 + 12$		✓	s	$\mathbf{M}_{4,i,4}$
$\mathrm{GF}(2^4)$/`0x13`	Hadamard-like	Yes	$4 + 12$	$4 + 12$		✓	–	[40]
$\mathrm{GL}(4, \mathrm{GF}(2))$	Circulant	Yes	$5 + 12$	$5 + 12$		✓	–	[31]
$\mathrm{GF}(2^4)$/`0x13`	Hadamard	Yes	$6 + 12$	$5 + 12$		✓	–	`Joltik`

Opt: Implementations of coefficients have been optimized under the s-XOR metric.
Min: Lightest possible matrix under field/order/involution based on d/s-metric.
When the rows have different cost, we take the average cost of a row.

To have an idea of how an entire diffusion matrix can be implemented, we provide an example of one possible implementation of the diffusion matrix $\mathbf{M}_{4,n,4}$ in Appendix A.2. We emphasize that this implementation only gives an illustration of the implementation of the entire diffusion matrix: further improvements are possible, for instance in ASIC if 3- or 4-input `XOR` cells are available in the library used, or in software with less intermediate registers.

MDS 8×8 Matrices Over $\mathbf{GF(2^4)}$. We observed that the Hadamard matrices presented in [42] are among the s-lightest MDS Hadamard matrices that we have found. This means that if we optimize the implementation of the field multiplications of the coefficients used in these matrices, we obtain the s-lightest MDS non-involution/involution Hadamard 8×8 matrices over $\mathrm{GF}(2^4)$. To be consistent, we name the non-involution Hadamard 8×8 matrices over $\mathrm{GF}(2^4)$ as $\mathbf{M}_{8,n,4}$,

$$\mathbf{M}_{8,n,4} = \begin{bmatrix} \texttt{0x1} & \texttt{0x2} & \texttt{0x6} & \texttt{0x8} & \texttt{0x9} & \texttt{0xc} & \texttt{0xd} & \texttt{0xa} \\ \texttt{0x2} & \texttt{0x1} & \texttt{0x8} & \texttt{0x6} & \texttt{0xc} & \texttt{0x9} & \texttt{0xa} & \texttt{0xd} \\ \texttt{0x6} & \texttt{0x8} & \texttt{0x1} & \texttt{0x2} & \texttt{0xd} & \texttt{0xa} & \texttt{0x9} & \texttt{0xc} \\ \texttt{0x8} & \texttt{0x6} & \texttt{0x2} & \texttt{0x1} & \texttt{0xa} & \texttt{0xd} & \texttt{0xc} & \texttt{0x9} \\ \texttt{0x9} & \texttt{0xc} & \texttt{0xd} & \texttt{0xa} & \texttt{0x1} & \texttt{0x2} & \texttt{0x6} & \texttt{0x8} \\ \texttt{0xc} & \texttt{0x9} & \texttt{0xa} & \texttt{0xd} & \texttt{0x2} & \texttt{0x1} & \texttt{0x8} & \texttt{0x6} \\ \texttt{0xd} & \texttt{0xa} & \texttt{0x9} & \texttt{0xc} & \texttt{0x6} & \texttt{0x8} & \texttt{0x1} & \texttt{0x2} \\ \texttt{0xa} & \texttt{0xd} & \texttt{0xc} & \texttt{0x9} & \texttt{0x8} & \texttt{0x6} & \texttt{0x2} & \texttt{0x1} \end{bmatrix},$$

Table 4: Comparison of 8×8 MDS matrices Over GF(2^4).

Matrix			Implementation					Ref.
Field/Ring	Type	Inv.	d-XOR(M)	s-XOR(M)	s-XOR(M^{-1})	Opt	Min	
GF(2^4)/0x13	Hadamard	No	$26+28$	$20+28$	$28+28$	✓	–	[42], $\mathbf{M}_{8,n,4}$
GF(2^4)/0x13	Hadamard	No	$28+28$	$21+28$	$28+28$	✓	–	WHIRLWIND [4]
GF(2^4)/0x13	Hadamard	No	$36+28$	$23+28$	$24+28$	✓	–	WHIRLWIND [4]
GF(2^4)/0x13	Hadamard	Yes	$36+28$	$25+28$		✓	–	[42], $\mathbf{M}_{8,i,4}$

Opt: Implementations of coefficients have been optimized under the s-XOR metric.
Min: Lightest possible matrix under field/order/involution.

and the involution Hadamard 8×8 matrices over GF(2^4) as $\mathbf{M}_{8,i,4}$,

$$\mathbf{M}_{8,i,4} = \begin{bmatrix} \texttt{0x2} & \texttt{0x3} & \texttt{0x4} & \texttt{0xc} & \texttt{0x5} & \texttt{0xa} & \texttt{0x8} & \texttt{0xf} \\ \texttt{0x3} & \texttt{0x2} & \texttt{0xc} & \texttt{0x4} & \texttt{0xa} & \texttt{0x5} & \texttt{0xf} & \texttt{0x8} \\ \texttt{0x4} & \texttt{0xc} & \texttt{0x2} & \texttt{0x3} & \texttt{0x8} & \texttt{0xf} & \texttt{0x5} & \texttt{0xa} \\ \texttt{0xc} & \texttt{0x4} & \texttt{0x3} & \texttt{0x2} & \texttt{0xf} & \texttt{0x8} & \texttt{0xa} & \texttt{0x5} \\ \texttt{0x5} & \texttt{0xa} & \texttt{0x8} & \texttt{0xf} & \texttt{0x2} & \texttt{0x3} & \texttt{0x4} & \texttt{0xc} \\ \texttt{0xa} & \texttt{0x5} & \texttt{0xf} & \texttt{0x8} & \texttt{0x3} & \texttt{0x2} & \texttt{0xc} & \texttt{0x4} \\ \texttt{0x8} & \texttt{0xf} & \texttt{0x5} & \texttt{0xa} & \texttt{0x4} & \texttt{0xc} & \texttt{0x2} & \texttt{0x3} \\ \texttt{0xf} & \texttt{0x8} & \texttt{0xa} & \texttt{0x5} & \texttt{0xc} & \texttt{0x4} & \texttt{0x3} & \texttt{0x2} \end{bmatrix}.$$

In addition, we improve the implementation of the diffusion matrices from the hash function WHIRLWIND using our s-XOR metric and make comparison with them in Table 4.

Lightweight MDS 4×4 Matrices Over GF(2^8). Our search for MDS 4×4 Hadamard and circulant matrices over GF(2^8) shows that the Hadamard and circulant matrices presented in [42] and [32] are among the s-lightest matrices found. Hence, we state the improved XOR count of these matrices in Table 5.

On the other hand, the sub-field construction of MDS 4×4 matrices over GF(2^8) using $\mathbf{M}_{4,n,4}$ and $\mathbf{M}_{4,i,4}$ generates new lightweight MDS matrices. Apart from the MDS circulant matrix over GL(8, GF(2)), which is presented in [31], our sub-field constructed MDS matrices outperform the existing lightweight MDS matrices under the s-XOR metric. We also include for comparison the diffusion matrices of AES and ANUBIS with optimized coefficient implementations.

MDS 8×8 Matrices Over GF(2^8). In contrast to MDS 4×4 matrices over GF(2^8), the search for MDS 8×8 Hadamard and circulant matrices over GF(2^8) has better results than the sub-field construction. For the non-involution MDS matrices, the circulant matrix presented in [32] denoted $\mathbf{M}_{8,n,8}$,

$$\mathbf{M}_{8,n,8} = \begin{bmatrix} \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0xe1} & \texttt{0x08} & \texttt{0xe0} & \texttt{0x01} & \texttt{0xa9} \\ \texttt{0xa9} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0xe1} & \texttt{0x08} & \texttt{0xe0} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0xa9} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0xe1} & \texttt{0x08} & \texttt{0xe0} \\ \texttt{0xe0} & \texttt{0x01} & \texttt{0xa9} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0xe1} & \texttt{0x08} \\ \texttt{0x08} & \texttt{0xe0} & \texttt{0x01} & \texttt{0xa9} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0xe1} \\ \texttt{0xe1} & \texttt{0x08} & \texttt{0xe0} & \texttt{0x01} & \texttt{0xa9} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0xe1} & \texttt{0x08} & \texttt{0xe0} & \texttt{0x01} & \texttt{0xa9} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0xe1} & \texttt{0x08} & \texttt{0xe0} & \texttt{0x01} & \texttt{0xa9} & \texttt{0x01} \end{bmatrix},$$

is also an s-lightest MDS circulant matrix.

It is not surprising that it is lighter than the circulant matrix presented in [10] as our search is an exhaustive search on circulant matrices while the latter considered a subclass

Table 5: Comparison of 4×4 MDS matrices over $\mathrm{GF}(2^8)$ and $\mathrm{GL}(8, \mathrm{GF}(2))$.

Matrix			**Implementation**					**Ref.**
Field/Ring	**Type**	**Inv.**	**d-XOR(M)**	**s-XOR(M)**	**s-XOR($\mathbf{M}^{-1}$)**	**Opt**	**Min**	
$\mathrm{GL}(8, \mathrm{GF}(2))$	Circulant	No	$3 + 24$	$3 + 24$	–	–	–	[31]
$\mathrm{GF}(2^4)$/`0x13`	Sub-field	No	$2 \cdot (2.5 + 12)$	$2 \cdot (2.5 + 12)$	$2 \cdot (9.25 + 12)$	✓	s	$\mathbf{M}_{4,n,4}$
$\mathrm{GF}(2^8)$/`0x1c3`	Toeplitz	No	$6.75 + 24$	$6 + 24$	$53.25 + 24$	✓	d	[40]
$\mathrm{GF}(2^8)$	Circulant	No	$7 + 24$	$6 + 24$	–	–	–	[10]
$\mathrm{GF}(2^8)$/`0x1c3`	Circulant	No	$8 + 24$	$7 + 24$	$51 + 24$	✓	–	[32]
$\mathrm{GF}(2^8)$/`0x1c3`	Hadamard	No	$13 + 24$	$11 + 24$	$52 + 24$	✓	–	[42]
$\mathrm{GF}(2^8)$/`0x11b`	Circulant	No	$14 + 24$	$12 + 24$	$53 + 24$	✓	–	`AES` [20]
$\mathrm{GF}(2^4)$/`0x13`	Sub-field	Yes	$2 \cdot (5 + 12)$	$2 \cdot (3.75 + 12)$		✓	s	$\mathbf{M}_{4,i,4}$
$\mathrm{GL}(8, \mathrm{GF}(2))$	Circulant	Yes	$9 + 24$	$9 + 24$		–	–	[31]
$\mathrm{GF}(2^4)$/`0x13`	Sub-field	Yes	$2 \cdot (6 + 12)$	$2 \cdot (5 + 12)$		✓	–	[42]
$\mathrm{GF}(2^8)$/`0x165`	Hadamard	Yes	$16 + 24$	$14 + 24$		✓	–	[42]
$\mathrm{GF}(2^8)$/`0x11d`	Hadamard	Yes	$22 + 24$	$20 + 24$		✓	–	`ANUBIS` [6]

Opt: Implementations of coefficients have been optimized under the s-XOR metric.
Min: Lightest possible matrix under field/order/involution based on d/s-metric.
When the rows have different cost, we take the average cost of a row.

of circulant matrices by building the circulant matrix from a single field element. We also include the s-XOR count of the diffusion matrix from `WHIRLPOOL` and `Grøstl` for comparison.

For MDS 8×8 involution matrices over $\mathrm{GF}(2^8)$, we found a new Hadamard matrix $\mathbf{M}_{8,i,8}$,

$$\mathbf{M}_{8,i,8} = \begin{bmatrix} \texttt{0x01} & \texttt{0x02} & \texttt{0x04} & \texttt{0x91} & \texttt{0x6a} & \texttt{0xb5} & \texttt{0xe1} & \texttt{0xa9} \\ \texttt{0x02} & \texttt{0x01} & \texttt{0x91} & \texttt{0x04} & \texttt{0xb5} & \texttt{0x6a} & \texttt{0xa9} & \texttt{0xe1} \\ \texttt{0x04} & \texttt{0x91} & \texttt{0x01} & \texttt{0x02} & \texttt{0xe1} & \texttt{0xa9} & \texttt{0x6a} & \texttt{0xb5} \\ \texttt{0x91} & \texttt{0x04} & \texttt{0x02} & \texttt{0x01} & \texttt{0xa9} & \texttt{0xe1} & \texttt{0xb5} & \texttt{0x6a} \\ \texttt{0x6a} & \texttt{0xb5} & \texttt{0xe1} & \texttt{0xa9} & \texttt{0x01} & \texttt{0x02} & \texttt{0x04} & \texttt{0x91} \\ \texttt{0xb5} & \texttt{0x6a} & \texttt{0xa9} & \texttt{0xe1} & \texttt{0x02} & \texttt{0x01} & \texttt{0x91} & \texttt{0x04} \\ \texttt{0xe1} & \texttt{0xa9} & \texttt{0x6a} & \texttt{0xb5} & \texttt{0x04} & \texttt{0x91} & \texttt{0x01} & \texttt{0x02} \\ \texttt{0xa9} & \texttt{0xe1} & \texttt{0xb5} & \texttt{0x6a} & \texttt{0x91} & \texttt{0x04} & \texttt{0x02} & \texttt{0x01} \end{bmatrix},$$

which outperforms the existing matrices like the Hadamard matrix from [32] and `KHAZAD` [7]. Interestingly, one of the coefficients in $\mathbf{M}_{8,i,8}$, namely `0x6a`, has d-XOR of 36 but has optimal s-XOR of 10. Thus, based on the d-XOR metric, it is unlikely that this element will be considered in the search for lightweight MDS matrices. But now, knowing that it can be implemented in 10 XOR operations, it became one of the coefficients for the s-lightest MDS Hadamard matrix. Notably, some of the field elements in the diffusion matrix of `Grøstl` and `KHAZAD` have s-XOR count more than 12, which implies that the implementation cost is suboptimal and further improvements might be possible (since we ran ChainMitM with parameter $\Lambda = 12$). The comparison of MDS 8×8 non-involutive and involutive matrices over $\mathrm{GF}(2^8)$ are summarized in Table 6.

6 Results on Non-Linear Layers

We now give our results related to small cryptographic non-linear permutations (Sboxes). Our goal is to find small circuits implementing those permutations with respect to the overall area for an ASIC implementation. Consequently, we tune the graph-based algorithms previously described in Section 3 by selecting the logical instructions among the ones available in some standard cell libraries, and calibrate their costs to represent their respective area. As a usual unit, we measure the gate sizes in terms of Gate Equivalent (GE), which is a normalized ratio using the area of a 2-input NAND gate as common reference. Our C++ implementations of the MitM algorithm usually find the optimal $\mathcal{B}$-implementation of 4-bit permutations in a few minutes to a few hours depending on the permutation, using several gigabytes of RAM on a 24-core machine (with CPUs at a frequency of 2 Ghz).

6.1 Previous Work

A notable previous work in the field of logic synthesis applied to cryptography is [43], where the authors are interested in finding "small" implementations of 4- and 5-bit Sboxes. In particular, they define "small" in different ways, by measuring the gate complexity (number of combinatorial gates in the circuits), depth complexity (number of traversed gates), bit-sliced complexity (number of software instructions), etc. Our work somehow relates to the gate complexity optimizations performed in [43] with a major difference: in our case, we do not stop at counting the number of gates, but we weight them differently according to their area. We note that by applying the gate area to the SAT-produced implementations, the overall area is higher than the area reached by our optimization technique (see below). One simple way to reproduce the results from [43] using our algorithms would be to set each operations in $\mathcal{B}$ to a constant cost, e.g. one. To obtain efficient implementations, the authors from [43] rely on decisional SAT problems solved using dedicated algorithms. However, after an offline discussion with the authors of [43], one emphasizes that writing SAT problems encoding different costs for various gates (e.g., area) is highly non-trivial and to the best of our knowledge, is not available in publicly available tools.

This research line area also relates to exact synthesis of combinatorial circuits. Exact synthesis comprises two different yet connected notions: exact combinatorial optimization and exact technology mapping. The former is independent of any library or technology and relies on an assumption which basically defines a criterion to minimize. For instance, in the case of the academic state-of-the-art synthesis tool ABC [15], this criterion minimizes the size of the And-Inverter network representing a Boolean function. This can also be compared to the optimizations performed in [43]. The latter adds the details of the technology in the optimization process. The objective function to minimize at this stage generally consists of a trade-off between area and latency of the overall circuits. While the two optimization steps, first without and then with the technology details, can provide sufficiently good implementations, we develop in the following the results of our study when we conduct both steps at the same time.

Table 6: Comparison of 8×8 MDS matrices over GF(2^8).

Matrix			**Implementation**					**Ref.**
Field/Ring	**Type**	**Inv.**	**d-XOR(M)**	**s-XOR(M)**	**s-XOR(M^{-1})**	**Opt**	**Min**	
GF(2^8)/`0x1c3`	Circulant	No	$30+56$	$24+56$	$104+56$	✓	–	[32], $\mathbf{M}_{8,n,8}$
GF(2^8)	Circulant	No	$40+56$	$26+56$	–	–	–	[10]
GF(2^8)/`0x1c3`	Hadamard	No	$40+56$	$32+56$	$123+56$	✓	–	[42]
GF(2^8)/`0x11d`	Circulant	No	$49+56$	$38+56$	$89+56$	✓	–	`WHIRLPOOL` [5]
GF(2^8)/`0x11b`	Circulant	No	$83+56$	$63+56$	$107+56$	–	–	`Grøstl` [22]
GF(2^8)/`0x1c3`	Hadamard	Yes	$72+56$	$36+56$		✓	–	$\mathbf{M}_{8,i,8}$
GF(2^8)/`0x1c3`	Hadamard	Yes	$46+56$	$40+56$		✓	–	[42]
GF(2^8)/`0x11d`	Hadamard	Yes	$98+56$	$73+56$		–	–	`KHAZAD` [7]

Opt: Implementations of coefficients have been optimized under the s-XOR metric.
Min: Lightest possible matrix under field/order/involution.

6.2 Context of the Comparison

Our main goal is to compare our meet-in-the-middle algorithm MitM to the state-of-the-art synthesis tools available to the academy and to the industry. By comparing the output results of both algorithms, we measure the quality of the synthesis in the setting where area only should be minimized. At the end of this section, we mention different optimizations scenarios to address other parameters.

We provide comparisons of our algorithms with the academic state-of-the-art logic synthesis package ABC [15] on two different technologies, namely UMC 180nm [48] and TMSC 65nm [45]. We emphasize that the choice of standard cell libraries used is almost irrelevant for our study as we are mainly interested in the quality of the area-optimized synthesis itself.

To conduct the comparison, we select a few 4-bit permutations that appear in the literature (see Table 7). In particular, we selected the Sboxes used in the lightweight block ciphers `PICCOLO` [41], `SKINNY` [9], `TWINE` [44], `PRESENT` [13], `Rectangle` [51] and one of the ten Sboxes present in `LBlock` [50].

Usual ASIC design can be long and complex, however our work only relates to the early phase of the process, where RTL code is converted to the gate-level netlist connecting the actual combinatorial components together to implement the Sbox. The synthesizer performs this job by running an algorithm that selects and links gates from the library, while optimizing various criteria like the overall area of the circuit, its delay, etc. In our experimental comparison, we measure area-only optimization and we restrict our algorithm to only use a small number of gates, but we allow ABC synthesis tools to use any of the gates available in the library. Consequently, the results produced by our algorithm described below can only be improved if more gates are available.

We list the combinatorial gates we use in our experiments in the following Table 8 and compare their respective area in two different libraries. Most of the selected combinatorial cells implement classical Boolean operations, whose functional behavior is recalled in Table 9.

We note that the two operations ANDN and ORN simply invert one of the two inputs of AND and OR, respectively, while MAOI1 and MOAI1 perform a slightly more complex operation. These two last 4-input gates are particularly interesting when we replicate their inputs. Indeed,

$$\begin{aligned} \mathrm{MAOI1}(a,b,a,b) &= \neg((a \wedge b) \vee (\neg(a \vee b)) = (\neg a \vee \neg b) \wedge (a \vee b) = \mathrm{XOR}(a,b),\\ \mathrm{MOAI1}(a,b,a,b) &= \neg((a \vee b) \wedge (\neg(a \wedge b)) = (\neg a \vee b) \wedge (a \vee \neg b) = \mathrm{XNOR}(a,b), \end{aligned}$$

which usually provides a smaller alternative to the 2-input XOR and XNOR gates. Hence, whenever these cells are available in the library we use, we replace the XOR and XNOR areas accordingly. We however do not allow our algorithms to use these gates with inputs of other shapes than (a,b,a,b).

Table 7: Sboxes used for comparison.

Sbox	Lookup Table	Reference
`PICCOLO`	`14,4,11,2,3,8,0,9,1,10,7,15,6,12,5,13`	[41]
`SKINNY`	`12,6,9,0,1,10,2,11,3,8,5,13,4,14,7,15`	[9]
`TWINE`	`12,0,15,10,2,11,9,5,8,3,13,7,1,14,6,4`	[44]
`PRESENT`	`12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2`	[13]
`Rectangle`	`6,5,12,10,1,14,7,9,11,0,3,13,8,15,4,2`	[51]
`LBlock` S_0	`14,9,15,0,13,4,10,11,1,2,8,3,7,6,12,5`	[50]

Table 8: Comparisons of several standard cell libraries for typical combinatorial cells. The values are given in GE.

Library	Logic process	NAND NOR	NOT	XOR XNOR	AND OR	ANDN ORN	NAND3 NOR3	XOR3 XNOR3	MAOI1	MOAI1
UMC	180nm	1.00	0.67	3.00	1.33	1.67	1.33	4.67	2.67	2.00
TSMC	65nm	1.00	0.50	3.00	1.50	1.50	1.50	5.50	2.50	2.50

6.3 Comparison with ABC using UMC 180nm Logic Process

We now compare our algorithm to the synthesis performed by ABC configured with the UMC 180nm library. We tune our algorithms with the costs of the components from the subset of cells listed in Table 8. In particular, this library provides the two 4-input cells MAOI1 and MOAI1, so we adapt the cost of XOR and XNOR to 2.67 and 2.00 GE, respectively. The results are shown in Table 10.

We remark that the PRESENT Sbox is implemented using 28.03 GE using the same UMC 180nm logic process in [13]: our area-optimized implementation therefore saves 6.70 GE per Sbox, which spares about 100 GE for the round-based implementation. This saving should however be mitigated since area was the only parameter taken into account in our synthesis: while this setting is relevant for some applications, it may not for some others, as in particular, the critical path of this implementation is probably higher than the one from [13]. We address the notion of tradeoff in Section 6.5.

In the case of PICCOLO, our algorithm finds a circuit with the same structure as proposed by the designers. However, we note that the NOR/XOR combination mentioned

Table 9: List of Boolean operators implemented by standard cells from the libraries. We recall that $\wedge, \vee, \oplus, \neg$ respectively stand for: logical and, or, exclusive or, not.

Operation	Function	Operation	Function
NAND	$(a,b) \to \neg(a \wedge b)$	XOR	$(a,b) \to a \oplus b$
NOR	$(a,b) \to \neg(a \vee b)$	XNOR	$(a,b) \to \neg(a \oplus b)$
AND	$(a,b) \to a \wedge b$	NAND3	$(a,b,c) \to \neg(a \wedge b \wedge c)$
OR	$(a,b) \to a \vee b$	NOR3	$(a,b,c) \to \neg(a \vee b \vee c)$
NOT	$a \to \neg a$	ANDN	$(a,b) \to \neg a \wedge b$
MAOI1	$(a,b,c,d) \to \neg((a \wedge b) \vee (\neg(c \vee d))$	ORN	$(a,b) \to \neg a \vee b$
MOAI1	$(a,b,c,d) \to \neg((a \vee b) \wedge (\neg(c \wedge d))$		

Table 10: Comparison of area-optimized synthesis on the UMC 180nm library.

Sbox	UMC 180nm Logic Process		
	ABC (from LUT)	Ours (from LUT)	ABC (from ours)
PICCOLO	21.00 GE	**13.00 GE**	–
SKINNY	22.33 GE	**13.33 GE**	–
TWINE	26.33 GE	**21.67 GE**	–
PRESENT	24.33 GE	**21.33 GE**	–
Rectangle	25.33 GE	**18.33 GE**	–
LBlock S_0	20.33 GE	**16.33 GE**	–

Table 11: Comparison of area-optimized synthesis on the TSMC 65nm library.

Sbox	**TSMC 65nm Logic Process**		
	ABC (from LUT)	**Ours (from LUT)**	**ABC (from ours)**
`PICCOLO`	18.00 GE	**14.00 GE**	–
`SKINNY`	20.00 GE	14.00 GE	**13.00 GE**
`TWINE`	24.50 GE	25.00 GE	**22.50 GE**
`PRESENT`	23.00 GE	24.00 GE	**22.50 GE**
`Rectangle`	23.50 GE	21.50 GE	**19.00 GE**
`LBlock` S_0	19.50 GE	19.00 GE	**17.50 GE**

in the `PICCOLO` specifications can be rewritten as an OR/XNOR combination, which saves some area (3.67 GE vs. 3.33 GE). Indeed, for all $x, y, z \in \mathrm{GF}(2)$, $\mathrm{XOR}(\mathrm{NOR}(x, y), z) = \mathrm{XNOR}(\mathrm{OR}(x, y), z)$. This allows to reach an implementation of the `PICCOLO` Sbox with 13 GE using this library.

By affecting the same costs to the implementations given in [43], we reach 22.67 GE for the `Rectangle` Sbox and 18.67 GE for the `LBlock` Sbox. Recall that this is expected as the optimization performed in this paper simply minimizes the number of gates, and it is nontrivial to consider their actual area instead, as our tools do.

We give all the implementations produced by our synthesis algorithm for the costs of the UMC 180nm logic process in Appendix B.1. In all the cases, we have attempted to launch ABC synthesizer using the full UMC 180nm library on the implementations produced by our tool: ABC could not reduce the area further.

6.4 Comparison with ABC using TSMC 65nm Logic Process

We now compare our algorithm to the synthesis performed by ABC configured with the TSMC 65nm library. Again, we adapt the cost of the components to be used by our algorithms to the subset of cells listed in Table 8. This library also provides MAOI1 and MOAI1, so we adapt the costs of XOR and XNOR to 2.50 GE each. The results are shown in Table 11.

Again, the `PICCOLO` circuit found by our tool is the same as the one proposed by the designers. In this case, the combined costs of NOR/XOR and OR/XNOR are the same, which yields an overall implementation that requires 14 GE.

As before, we have used ABC algorithms on the circuits produced by our tool to check for further improvements. While we restricted our tool to a subset of gates available in the TSMC library, we let ABC benefit from the full set of cells. In several cases, the joint use of the two tools allowed to reduced the area (see last column of Table 11).

We give all the implementations produced by our synthesis algorithm for the same costs as the TSMC 65nm logic process in Appendix B.2, as well as the ones reached after further optimizations by ABC.

6.5 Synthesis with Area/Delay Tradeoffs

In all this section, we were interested in providing small *area-optimized circuits* that implement non-linear Sboxes. While this scenario is of practical interest and was our optimization strategy throughout this paper, one may also want to optimize for delay and/or both delay and area. Again, commercial synthesizers do provide these options, but probably apply general heuristic algorithms regardless of the size of the input function to optimize.

As a rough estimation of the delay, we can count the length of the critical path in terms of number of gates traversed. For instance, the previous PRESENT Sbox implementation using 21.33 GE on UMC 180nm logic process has a critical path of length 12, as one output bit is only available after sequentially evaluating 12 gates. In comparison, the PICCOLO Sbox implementation using 4 NOR and 4 XNOR/XOR in a Feistel-like structure has a maximal path of length 4.

The algorithms presented in this paper are very generic and allow to optimize nearly all aspects of an hardware implementation, including area, delay, area/delay, etc., simply by affecting the correct costs to some allowed operations. We suggest here a simple heuristic to tweak the MitM from Section 3.1 to roughly factor in the cell latency. The main idea consists in affecting a penalty to all the gates trying to read the output of another gate, which in practice might not be directly available due to delays. As a result, the implementations outputed are no longer optimal $\mathcal{B}$-implementations, but instead trade some area for shorter delay.

More precisely, we attach an additional information to all the nodes to represent "when" the output will be ready, and model this delay as a percentage of the surface of the gate applied (e.g., 10%) to relate to its combinatorial complexity. Note that this metric relies on the same unit (GE), hence does not capture perfectly the actual delay of the gates. Then, to insert a new node in the graph, the Expand algorithm checks whether all the input values for the instructions are ready, and if not, injects the delay accordingly.

Another approach that we leave out of the paper due to space limitation is delay-only optimizations, which might be critical in some other particular applications. We however emphasize once again that the algorithms presented here can be tweaked easily to evaluate more precisely the latency of the overall constructions.

7 Conclusions and Future Work

In this article, we have described new algorithms and heuristics to generically improve implementation of lightweight block cipher components such as Sboxes or diffusion matrices. In practice, our tool, LIGHTER, managed to improve the implementations of many Sboxes for various technologies, but also to optimize the computation of existing linear layers. We also present new lightest diffusion matrices that are found through exhaustive search with early-abort strategies. The tool will be online and free to use after publication. We believe it will be very useful for cryptographic designers.

There are several future works worth considering. First, it would be interesting to look at the implementation of a linear diffusion matrix as a whole rather than element by element. This is challenging in the general case due to the dimension of the problem, but some heuristics might be able to cut parts of the search space. Similarly, for non-linear layers, one could try to reach Sboxes sizes larger than 4 bits (a natural target would be the AES 8-bit Sbox using only heuristics).

Secondly, we believe there are several further applications to our tool that we have not fully explored yet. For example, improvements of 2- and 3-instructions bit-sliced software implementations, low-latency implementations, low non-linearity implementations, delay/area tradeoffs, etc. Besides, enabling more complex gates from the standard cell libraries would allow our modeling to better fit reality. An additional feature targeting hardware implementations would be to integrate a notion of distance between wires to somehow model the place-and-route stage, but this aspect appears to be extremely hard to incorporate in our current algorithms.

Finally, more results might be obtained by looking at serial diffusion matrices (i.e. matrices computed as a power of a companion matrix), like the ones used in the PHOTON [24] hash function or the LED [25] block cipher, since they offer a natural trade-off between throughput and area.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. Additionally, we thank Luca Amaru, Anupam Chattopadhyay, Amir Moradi and Pascal Sasdrich for their help in the experimental results produced. We also thank Meicheng Liu for his help in the search for lightweight linear layer. This work has been partially supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06) and by the French Agence Nationale de la Recherche through the BRUTUS project under Contract ANR-14-CE28-0015.

References

[1] Abd-El-Barr, M., Al-Farhan, A.: A Highly Parallel Area Efficient S-Box Architecture for AES Byte-Substitution. International Journal of Engineering and Technology **6**(5) (2014) 346

[2] Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In Oswald, E., Fischlin, M., eds.: EUROCRYPT 2015, Part I. Volume 9056 of LNCS., Springer, Heidelberg (April 2015) 430–454

[3] Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A Block Cipher for Low Energy. In Iwata, T., Cheon, J.H., eds.: ASIACRYPT 2015, Part II. Volume 9453 of LNCS., Springer, Heidelberg (November / December 2015) 411–436

[4] Barreto, P., Nikov, V., Nikova, S., Rijmen, V., Tischhauser, E.: Whirlwind: a new cryptographic hash function. Designs, Codes and Cryptography **56**(2) (2010) 141–162

[5] Barreto, P.S.L.M., Rijmen, V.: The WHIRLPOOL Hashing Function. Submitted to NESSIE, September 2000 (2000)

[6] Barreto, P.S., Rijmen, V.: The Anubis Block Cipher. Submission to the NESSIE Project

[7] Barreto, P.S., Rijmen, V.: The Khazad Legacy-Level Block Cipher. Submission to the NESSIE Project

[8] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404 (2013)

[9] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In Robshaw, M., Katz, J., eds.: CRYPTO 2016, Part II. Volume 9815 of LNCS., Springer, Heidelberg (August 2016) 123–153

[10] Beierle, C., Kranz, T., Leander, G.: Lightweight Multiplication in GF(2^n) with Applications to MDS Matrices. In Robshaw, M., Katz, J., eds.: CRYPTO 2016, Part I. Volume 9814 of LNCS., Springer, Heidelberg (August 2016) 625–653

[11] Bernstein, D.J.: Optimizing linear maps modulo 2. In: Workshop Record of SPEED-CC: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers

[12] Biham, E., Anderson, R.J., Knudsen, L.R.: Serpent: A New Block Cipher Proposal. In Vaudenay, S., ed.: FSE'98. Volume 1372 of LNCS., Springer, Heidelberg (March 1998) 222–238

[13] Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In Paillier, P., Verbauwhede, I., eds.: CHES 2007. Volume 4727 of LNCS., Springer, Heidelberg (September 2007) 450–466

[14] Boyar, J., Matthews, P., Peralta, R.: Logic Minimization Techniques with Applications to Cryptology. Journal of Cryptology **26**(2) (April 2013) 280–312

[15] Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: International Conference on Computer Aided Verification, Springer (2010) 24–40

[16] Brayton, R.K., Hachtel, G.D., McMullen, C., Sangiovanni-Vincentelli, A.: Logic minimization algorithms for VLSI synthesis. Volume 2. Springer Science & Business Media (1984)

[17] Buchfuhrer, D., Umans, C. In: The Complexity of Boolean Formula Minimization. Springer Berlin Heidelberg, Berlin, Heidelberg (2008) 24–35

[18] Canright, D.: A Very Compact S-Box for AES. In Rao, J.R., Sunar, B., eds.: CHES 2005. Volume 3659 of LNCS., Springer, Heidelberg (August / September 2005) 441–455

[19] Canteaut, A., Duval, S., Leurent, G.: Construction of Lightweight S-Boxes Using Feistel and MISTY Structures. In Dunkelman, O., Keliher, L., eds.: SAC 2015. Volume 9566 of LNCS., Springer, Heidelberg (August 2016) 373–393

[20] Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer (2002)

[21] Devadas, S., Ghosh, A., Keutzer, K.: Logic Synthesis. McGraw-Hill, Inc., New York, NY, USA (1994)

[22] Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., ren S. Thomsen, S.: Grøstl - a SHA-3 candidate. Candidate of SHA-3 competition

[23] Guo, J., Jean, J., Nikolić, I., Qiao, K., Sasaki, Y., Sim, S.M.: Invariant Subspace Attack Against Midori64 and The Resistance Criteria for S-box Designs. To appear in FSE2017 (2017)

[24] Guo, J., Peyrin, T., Poschmann, A.: The PHOTON Family of Lightweight Hash Functions. In Rogaway, P., ed.: CRYPTO 2011. Volume 6841 of LNCS., Springer, Heidelberg (August 2011) 222–239

[25] Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.J.B.: The LED Block Cipher. [36] 326–341

[26] Gupta, K.C., Ray, I.G.: Cryptographically significant MDS matrices based on circulant and circulant-like matrices for lightweight applications. Cryptography and Communications **7**(2) (2015) 257–287

[27] Hlavička, J., Fišer, P.: BOOM: A Heuristic Boolean Minimizer. In: Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design. ICCAD '01, Piscataway, NJ, USA, IEEE Press (2001) 439–442

[28] Juels, A., Weis, S.A.: Authenticating Pervasive Devices with Human Protocols. In Shoup, V., ed.: CRYPTO 2005. Volume 3621 of LNCS., Springer, Heidelberg (August 2005) 293–308

[29] Khoo, K., Peyrin, T., Poschmann, A.Y., Yap, H.: FOAM: Searching for Hardware-Optimal SPN Structures and Components with a Fair Comparison. In Batina, L., Robshaw, M., eds.: CHES 2014. Volume 8731 of LNCS., Springer, Heidelberg (September 2014) 433–450

[30] Knop, L.E..: Linear Algebra: A First Course with Applications. Textbooks in mathematics (Boca Raton, Fla.). Chapman and Hall/CRC (2008)

[31] Li, Y., Wang, M.: On the Construction of Lightweight Circulant Involutory MDS Matrices. [35] 121–139

[32] Liu, M., Sim, S.M.: Lightweight MDS Generalized Circulant Matrices. [35] 101–120

[33] MacWilliams, F., Sloane, N.: The Theory of Error-Correcting Codes. 2nd edn. North-holland Publishing Company (1986)

[34] Osvik, D.A.: Speeding up Serpent. In: AES Candidate Conference. (2000) 317–329

[35] Peyrin, T., ed.: FSE 2016. In Peyrin, T., ed.: FSE 2016. Volume 9783 of LNCS., Springer, Heidelberg (March 2016)

[36] Preneel, B., Takagi, T., eds.: CHES 2011. In Preneel, B., Takagi, T., eds.: CHES 2011. Volume 6917 of LNCS., Springer, Heidelberg (September / October 2011)

[37] Rudell, R.L.: Multiple-Valued Logic Minimization for PLA Synthesis. Technical report, EECS Department, University of California, Berkeley (1986)

[38] Saarinen, M.J.O.: Cryptographic Analysis of All 4×4-Bit S-Boxes. In Miri, A., Vaudenay, S., eds.: SAC 2011. Volume 7118 of LNCS., Springer, Heidelberg (August 2012) 118–133

[39] Sarkar, S., Sim, S.M.: A Deeper Understanding of the XOR Count Distribution in the Context of Lightweight Cryptography. In Pointcheval, D., Nitaj, A., Rachidi, T., eds.: AFRICACRYPT 16. Volume 9646 of LNCS., Springer, Heidelberg (April 2016) 167–182

[40] Sarkar, S., Syed, H.: Lightweight Diffusion Layer: Importance of Toeplitz Matrices. IACR Trans. Symmetric Cryptol. **2016**(1) (2016) 95–113

[41] Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: An Ultra-Lightweight Blockcipher. [36] 342–357

[42] Sim, S.M., Khoo, K., Oggier, F.E., Peyrin, T.: Lightweight MDS Involution Matrices. In Leander, G., ed.: FSE 2015. Volume 9054 of LNCS., Springer, Heidelberg (March 2015) 471–493

[43] Stoffelen, K.: Optimizing S-Box Implementations for Several Criteria Using SAT Solvers. [35] 140–160

[44] Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE : A Lightweight Block Cipher for Multiple Platforms. In Knudsen, L.R., Wu, H., eds.: SAC 2012. Volume 7707 of LNCS., Springer, Heidelberg (August 2013) 339–354

[45] TSMC: 65 nm Standard Cell Library (August 2006)

[46] Ullrich, M., De Canniere, C., Indesteege, S., Küçük, Ö., Mouha, N., Preneel, B.: Finding optimal bitsliced implementations of $4\times$ 4-bit s-boxes. In: SKEW 2011. (2011) 16–17

[47] Vaudenay, S.: On the Need for Multipermutations: Cryptanalysis of MD4 and SAFER. In Preneel, B., ed.: FSE'94. Volume 1008 of LNCS., Springer, Heidelberg (December 1995) 286–297

[48] Virtual Silicon Inc.: 0.18 μm VIP Standard Cell Library Tape Out Ready, Part Number: UMCL18G212T3, Process: UMC Logic 0.18 μm Generic II Technology: 0.18μm (July 2004)

[49] Wamser, M.S., Holzbaur, L., Sigl, G.: A petite and power saving design for the AES S-Box. In: Digital System Design (DSD), 2015 Euromicro Conference on, IEEE (2015) 661–667

[50] Wu, W., Zhang, L.: LBlock: A Lightweight Block Cipher. In Lopez, J., Tsudik, G., eds.: ACNS 11. Volume 6715 of LNCS., Springer, Heidelberg (June 2011) 327–344

[51] Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. SCIENCE CHINA Information Sciences **58**(12) (2015) 1–15

[52] Zhang, X., Wu, N., Yan, G., DONG, L.: Hardware implementation of compact AES S-box. IAENG International Journal of Computer Science **42**(2) (2015)

[53] Zhao, R., Wu, B., Zhang, R., Zhang, Q.: Designing Optimal Implementations of Linear Layers (Full Version). Cryptology ePrint Archive, Report 2016/1118 (2016)

A Implementation of Linear Layer

A.1 Implementation of the Field Elements in $\mathrm{GF}(2^4)/$0x13

```
#define mul2_GF16_0x13(x0,x1,x2,x3) do {  \
 x3 = XOR(x3,x0);                         \
} while(0); /* Output: (MSB) x1,x2,x3,x0 (LSB) */
#define mul3_GF16_0x13(x0,x1,x2,x3) do {  \
 x3 = XOR(x3,x0); x0 = XOR(x0,x1);        \
 x1 = XOR(x1,x2); x2 = XOR(x2,x3);        \
} while(0); /* Output: (MSB) x0,x1,x2,x3 (LSB) */
#define mul4_GF16_0x13(x0,x1,x2,x3) do {  \
 x3 = XOR(x3,x0); x0 = XOR(x0,x1);        \
} while(0); /* Output: (MSB) x2,x3,x0,x1 (LSB) */
#define mul5_GF16_0x13(x0,x1,x2,x3) do {  \
 x2 = XOR(x2,x0); x3 = XOR(x3,x1);        \
 x1 = XOR(x1,x2); x0 = XOR(x0,x3);        \
} while(0); /* Output: (MSB) x2,x0,x1,x3 (LSB) */
#define mul6_GF16_0x13(x0,x1,x2,x3) do {  \
 x3 = XOR(x3,x1); x1 = XOR(x1,x0);        \
 x2 = XOR(x2,x1); x0 = XOR(x0,x2);        \
 x2 = XOR(x2,x3);                         \
} while(0); /* Output: (MSB) x0,x2,x3,x1 (LSB) */
#define mul7_GF16_0x13(x0,x1,x2,x3) do {  \
 x2 = XOR(x2,x0); x1 = XOR(x1,x2);        \
 x3 = XOR(x3,x1); x0 = XOR(x0,x3);        \
 x2 = XOR(x2,x0);                         \
} while(0); /* Output: (MSB) x1,x3,x0,x2 (LSB) */
#define mul8_GF16_0x13(x0,x1,x2,x3) do {  \
 x3 = XOR(x3,x0); x0 = XOR(x0,x1);        \
 x1 = XOR(x1,x2);                         \
} while(0); /* Output: (MSB) x3,x0,x1,x2 (LSB) */
#define mul9_GF16_0x13(x0,x1,x2,x3) do {  \
 x2 = XOR(x2,x3);                         \
} while(0); /* Output: (MSB) x3,x0,x1,x2 (LSB) */
#define mul10_GF16_0x13(x0,x1,x2,x3) do { \
 x0 = XOR(x0,x2); x1 = XOR(x1,x0);        \
 x3 = XOR(x3,x1); x2 = XOR(x2,x3);        \
} while(0); /* Output: (MSB) x2,x1,x3,x0 (LSB) */
#define mul11_GF16_0x13(x0,x1,x2,x3) do { \
 x2 = XOR(x2,x0); x1 = XOR(x1,x3);        \
 x0 = XOR(x0,x1); x3 = XOR(x3,x2);        \
} while(0); /* Output: (MSB) x1,x2,x0,x3 (LSB) */
#define mul12_GF16_0x13(x0,x1,x2,x3) do { \
 x0 = XOR(x0,x2); x2 = XOR(x2,x1);        \
 x1 = XOR(x1,x3); x3 = XOR(x3,x0);        \
} while(0); /* Output: (MSB) x3,x1,x0,x2 (LSB) */
#define mul13_GF16_0x13(x0,x1,x2,x3) do { \
 x2 = XOR(x2,x3); x1 = XOR(x1,x2);        \
} while(0); /* Output: (MSB) x2,x3,x0,x1 (LSB) */
#define mul14_GF16_0x13(x0,x1,x2,x3) do { \
 x2 = XOR(x2,x3); x1 = XOR(x1,x2);        \
 x0 = XOR(x0,x1); x3 = XOR(x3,x0);        \
} while(0); /* Output: (MSB) x0,x1,x2,x3 (LSB) */
#define mul15_GF16_0x13(x0,x1,x2,x3) do { \
 x2 = XOR(x2,x3); x1 = XOR(x1,x2);        \
 x0 = XOR(x0,x1);                         \
} while(0); /* Output: (MSB) x1,x2,x3,x0 (LSB) */
```

Figure 2: Minimal implementations of $GF(16)/$0x13 field elements under the s-XOR metric. The inputs are given as x0,x1,x2,x3, where x0 is the MSB and x3 the LSB.

A.2 Implementation of $M_{4,n,4}$

```
/* Bitsliced Implementation of the full matrix M_{4,n,4} */
#define MixCol_M_4n4(x0,...,x15,y0,...,y15) do {  \
 y0  = XOR(x2,x3);                                \
 y1  = XOR(x1,y0);                                \
 y2  = XOR(x11,x8);                               \
 y12 = XOR(XOR(XOR(y0,x4),x9),x12);               \
 y13 = XOR(XOR(XOR(x3,x5),x10),x13);              \
 y14 = XOR(XOR(XOR(x0,x6),y2),x14);               \
 y15 = XOR(XOR(XOR(y1,x7),x8),x15);               \
 y0  = XOR(x3,x0);                                \
 y1  = XOR(x6,x7);                                \
 y2  = XOR(x5,y1);                                \
 y8  = XOR(XOR(XOR(x1,y1),x8),x12);               \
 y9  = XOR(XOR(XOR(x2,x7),x9),x13);               \
 y10 = XOR(XOR(XOR(y0,x4),x10),x14);              \
 y11 = XOR(XOR(XOR(x0,y2),x11),x15);              \
 y0  = XOR(x7,x4);                                \
 y1  = XOR(x10,x11);                              \
 y2  = XOR(x9,y1);                                \
 y4  = XOR(XOR(XOR(x0,x5),y1),x12);               \
 y5  = XOR(XOR(XOR(x1,x6),x11),x13);              \
 y6  = XOR(XOR(XOR(x2,y0),x8),x14);               \
 y7  = XOR(XOR(XOR(x3,x4),y2),x15);               \
 y3  = XOR(x15,x12);                              \
 y0  = XOR(XOR(XOR(x0,x4),x8),x13);               \
 y1  = XOR(XOR(XOR(x1,x5),x9),x14);               \
 y2  = XOR(XOR(XOR(x2,x6),x10),y3);               \
 y3  = XOR(XOR(XOR(x3,x7),x11),x12);              \
} while(0); /* Output: y0, y1, ..., y15 */
```

Figure 3: Implementation of the matrix $\mathbf{M}_{4,n,4}$. The input vector is stored in `x0,x1,...,x15` and output vector in `y0,y1,...,y15` where `x0,y0` are the MSB and `x15,y15` the LSB.

B Implementation of Some Sboxes

In this section, we give the implementations of several Sboxes mapped on the two standard cell libraries used in this paper. We have selected the BLIF format.

B.1 Using UMC 180nm Logic Process

```
.model sbox_piccolo
.inputs in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate or     a=in0  b=in1               O=n9
.gate moai1  a=in3  b=n9   c=in3 d=n9   O=out0
.gate or     a=in1  b=in2               O=n11
.gate moai1  a=in0  b=n11  c=in0 d=n11  O=out1
.gate nor    a=in2  b=out0              O=n13
.gate moai1  a=in1  b=n13  c=in1 d=n13  O=out2
.gate or     a=out0 b=out1              O=n15
.gate moai1  a=in2  b=n15  c=in2 d=n15  O=out3
.end
```

Figure 4: Area-optimized hardware implementation of PICCOLO Sbox using 13.00 GE with the standard cell library of the UMC 180nm logic process.

```
.model   sbox_skinny
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate or     a=in0  b=in1               O=n9
.gate moai1  a=in3  b=n9   c=in3 d=n9   O=out0
.gate or     a=in1  b=in2               O=n11
.gate moai1  a=in0  b=n11  c=in0 d=n11  O=out1
.gate or     a=in2  b=out0              O=n13
.gate moai1  a=in1  b=n13  c=in1 d=n13  O=out2
.gate or     a=out0 b=out1              O=n15
.gate moai1  a=in2  b=n15  c=in2 d=n15  O=out3
.end
```

Figure 5: Area-optimized hardware implementation of SKINNY Sbox using 13.33 GE with the standard cell library of the UMC 180nm logic process.

```
.model   sbox_twine
.inputs  in0  in1  in2  in3
.outputs out0 out1 out2 out3
.gate nor    a=in1   b=in2                   O=tmp1
.gate moai1  a=tmp1  b=in3   c=tmp1   d=in3   O=t1
.gate or     a=in0   b=t1                    O=tmp2
.gate moai1  a=tmp2  b=in1   c=tmp2   d=in1   O=t2
.gate moai1  a=in2   b=t1    c=in2    d=t1    O=t3
.gate moai1  a=t3    b=in0   c=t3     d=in0   O=t4
.gate nand3  a=in2   b=t2    c=t4             O=tmp3
.gate moai1  a=tmp3  b=t3    c=tmp3   d=t3    O=out0
.gate nand   a=out0  b=t2                    O=tmp4
.gate moai1  a=tmp4  b=in2   c=tmp4   d=in2   O=out1
.gate moai1  a=t2    b=t4    c=t2     d=t4    O=out2
.gate nor    a=out1  b=out2                  O=tmp5
.gate moai1  a=tmp5  b=t2    c=tmp5   d=t2    O=out3
.end
```

Figure 6: Area-optimized hardware implementation of TWINE Sbox using 21.67 GE with the standard cell library of the UMC 180nm logic process.

```
.model   sbox_present
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate   or     a=in1   b=in3                     O=tmp1
.gate   moai1  a=tmp1  b=in2   c=tmp1   d=in2    O=tmp2
.gate   nor    a=in1   b=tmp2                    O=tmp3
.gate   moai1  a=tmp3  b=in0   c=tmp3   d=in0    O=tmp4
.gate   moai1  a=tmp4  b=in3   c=tmp4   d=in3    O=tmp5
.gate   invx   a=tmp4                            O=tmp6
.gate   nor    a=tmp2  b=tmp6                    O=tmp7
.gate   nor    a=tmp7  b=tmp5                    O=tmp8
.gate   moai1  a=tmp8  b=in1   c=tmp8   d=in1    O=out2
.gate   moai1  a=tmp2  b=tmp5  c=tmp2   d=tmp5   O=out3
.gate   nor    a=out2  b=tmp6                    O=tmp9
.gate   moai1  a=tmp9  b=tmp2  c=tmp9   d=tmp2   O=out1
.gate   nor3   a=out1  b=out2  c=out3            O=tmp10
.gate   moai1  a=tmp10 b=tmp6  c=tmp10  d=tmp6   O=out0
.end
```

Figure 7: Area-optimized hardware implementation of PRESENT Sbox using 21.33 GE with the standard cell library of the UMC 180nm logic process.

```
.model    sbox_lblock
.inputs   in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  nor   a=in2  b=in3                 O=tmp1
.gate  moai1 a=tmp1 b=in0  c=tmp1 d=in0   O=tmp2
.gate  nor   a=in1  b=tmp2                O=tmp3
.gate  moai1 a=tmp3 b=in3  c=tmp3 d=in3   O=tmp4
.gate  moai1 a=in2  b=in1  c=in2  d=in1   O=tmp5
.gate  moai1 a=tmp2 b=tmp5 c=tmp2 d=tmp5  O=out3
.gate  moai1 a=tmp4 b=in2  c=tmp4 d=in2   O=out1
.gate  and   a=out3 b=tmp2                O=tmp6
.gate  moai1 a=tmp6 b=tmp4 c=tmp6 d=tmp4  O=out2
.gate  nand  a=out3 b=out1                O=tmp7
.gate  moai1 a=tmp7 b=tmp2 c=tmp7 d=tmp2  O=out0
.end
```

Figure 8: Area-optimized hardware implementation of `Rectangle` Sbox using 18.33 GE with the standard cell library of the UMC 180nm logic process.

```
.model sbox_lblock_S0
.inputs in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  xnor  a=in3  b=in2                  O=tmp1
.gate  nand  a=in1  b=tmp1                 O=tmp2
.gate  moai1 a=tmp2 b=in3  c=tmp2  d=in3   O=tmp3
.gate  or    a=in0  b=in1                  O=tmp4
.gate  moai1 a=tmp4 b=tmp1 c=tmp4  d=tmp1  O=out3
.gate  moai1 a=in0  b=tmp3 c=in0   d=tmp3  O=out2
.gate  nor   a=out2 b=out3                 O=tmp5
.gate  moai1 a=tmp5 b=in0  c=tmp5  d=in0   O=out0
.gate  nand  a=out0 b=out2                 O=tmp6
.gate  moai1 a=tmp6 b=in1  c=tmp6  d=in1   O=out1
.end
```

Figure 9: Area-optimized hardware implementation of `LBlock` S_0 Sbox using 16.33 GE with the standard cell library of the UMC 180nm logic process.

B.2 Using TSMC 65nm Logic Process

```
.model   sbox_piccolo
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  nor    a=in0  b=in1              O=n9
.gate  maoi22 a=in3  b=n9   c=in3 d=n9  O=out0
.gate  nor    a=in1  b=in2              O=n11
.gate  maoi22 a=in0  b=n11  c=in0 d=n11 O=out1
.gate  nor    a=in2  b=out0             O=n13
.gate  moai22 a=in1  b=n13  c=in1 d=n13 O=out2
.gate  nor    a=out0 b=out1             O=n15
.gate  maoi22 a=in2  b=n15  c=in2 d=n15 O=out3
.end
```

Figure 10: Area-optimized hardware implementation of PICCOLO Sbox using 14.00 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_skinny_opt_tsmc_1400
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  nor    a=in0  b=in1              O=n9
.gate  maoi22 a=in3  b=n9   c=in3 d=n9  O=out0
.gate  nor    a=in1  b=in2              O=n11
.gate  maoi22 a=in0  b=n11  c=in0 d=n11 O=out1
.gate  nor    a=in2  b=out0             O=n13
.gate  maoi22 a=in1  b=n13  c=in1 d=n13 O=out2
.gate  nor    a=out0 b=out1             O=n15
.gate  maoi22 a=in2  b=n15  c=in2 d=n15 O=out3
.end
```

Figure 11: Area-optimized hardware implementation of SKINNY Sbox using 14.00 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_twine
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  invx    a=in2                            O=tmp1
.gate  nor     a=in3    b=tmp1                  O=tmp2
.gate  nor     a=tmp2   b=in0                   O=tmp3
.gate  maoi22  a=tmp3   b=in1    c=tmp3  d=in1  O=tmp4
.gate  nand    a=tmp1   b=in3                   O=tmp5
.gate  nand    a=tmp5   b=tmp4                  O=tmp6
.gate  moai22  a=tmp6   b=in0    c=tmp6  d=in0  O=tmp7
.gate  nand    a=tmp7   b=tmp4                  O=tmp8
.gate  maoi22  a=tmp8   b=tmp1   c=tmp8  d=tmp1 O=out1
.gate  nor     a=in3    b=out1                  O=tmp9
.gate  maoi22  a=tmp9   b=tmp4   c=tmp9  d=tmp4 O=tmp10
.gate  nor     a=out1   b=tmp10                 O=tmp11
.gate  nor     a=tmp11  b=tmp7                  O=tmp12
.gate  maoi22  a=tmp12  b=in3    c=tmp12 d=in3  O=out2
.gate  invx    a=tmp10                          O=out3
.gate  nand    a=out3   b=out2                  O=tmp13
.gate  moai22  a=tmp13  b=tmp7   c=tmp13 d=tmp7 O=out0
.end
```

Figure 12: Area-optimized hardware implementation of TWINE Sbox using 25 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_present
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  nor     a=in1   b=in3                  O=tmp1
.gate  maoi22  a=tmp1  b=in2   c=tmp1  d=in2  O=tmp2
.gate  nor     a=in1   b=tmp2                 O=tmp3
.gate  maoi22  a=tmp3  b=in0   c=tmp3  d=in0  O=tmp4
.gate  maoi22  a=tmp4  b=in3   c=tmp4  d=in3  O=tmp5
.gate  nor     a=tmp2  b=tmp4                 O=tmp6
.gate  nor     a=tmp6  b=tmp5                 O=tmp7
.gate  moai22  a=tmp7  b=in1   c=tmp7  d=in1  O=out2
.gate  moai22  a=tmp2  b=tmp5  c=tmp2  d=tmp5 O=out3
.gate  nor     a=out2  b=tmp4                 O=tmp8
.gate  moai22  a=tmp8  b=tmp2  c=tmp8  d=tmp2 O=out1
.gate  nor3    a=out2  b=out1  c=out3         O=tmp9
.gate  moai22  a=tmp9  b=tmp4  c=tmp9  d=tmp4 O=out0
.end
```

Figure 13: Area-optimized hardware implementation of `PRESENT` Sbox using 24.00 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_rectangle
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  nor     a=in2   b=in3                   tmp1
.gate  maoi22  a=tmp1  b=in0   c=tmp1  d=in0   O=tmp2
.gate  maoi22  a=tmp2  b=in1   c=tmp2  d=in1   O=tmp3
.gate  nor     a=in2   b=tmp2                  O=tmp4
.gate  moai22  a=tmp4  b=in3   c=tmp4  d=in3   O=tmp5
.gate  moai22  a=tmp3  b=in2   c=tmp3  d=in2   O=out3
.gate  moai22  a=tmp5  b=tmp3  c=tmp5  d=tmp3  O=out2
.gate  nand    a=tmp2  b=out3                  O=tmp6
.gate  moai22  a=tmp6  b=tmp5  c=tmp6  d=tmp5  O=out1
.gate  nand    a=out1  b=out3                  O=tmp7
.gate  maoi22  a=tmp7  b=tmp2  c=tmp7  d=tmp2  O=out0
.end
```

Figure 14: Area-optimized hardware implementation of `Rectangle` Sbox using 21.50 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_lblock
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  nor    a=in0  b=in1                O=tmp1
.gate  moai22 a=tmp1 b=in2  c=tmp1 d=in2  O=tmp2
.gate  maoi22 a=in3  b=tmp2 c=in3  d=tmp2 O=out3
.gate  nand   a=in1  b=out3               O=tmp3
.gate  moai22 a=tmp3 b=in3  c=tmp3 d=in3  O=tmp4
.gate  nor    a=in0  b=tmp4               O=tmp5
.gate  maoi22 a=tmp5 b=in1  c=tmp5 d=in1  O=out1
.gate  moai22 a=in0  b=tmp4 c=in0  d=tmp4 O=out2
.gate  nor    a=out3 b=out2               O=tmp6
.gate  moai22 a=tmp6 b=in0  c=tmp6 d=in0  O=out0
.end
```

Figure 15: Area-optimized hardware implementation of `LBlock` S_0 Sbox using 19 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_skinny
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  nor     a=in0  b=in1                 O=n9
.gate  iao22   a=in3  b=n9   c=in3  d=n9   O=out0
.gate  nor     a=in1  b=in2                 O=n11
.gate  iao22   a=in0  b=n11  c=in0  d=n11  O=out1
.gate  nor     a=in2  b=out0                O=n13
.gate  iao22   a=in1  b=n13  c=in1  d=n13  O=out2
.gate  or      a=out0 b=out1                O=n15
.gate  invx    a=n15                        O=n16
.gate  iao22   a=in2  b=n16  c=in2  d=n16  O=out3
.end
```

Figure 16: Combination of LIGHTER and ABC: Area-optimized hardware implementation of SKINNY Sbox using 13.00 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_twine
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  invx    a=in2                         O=n9
.gate  invx    a=in3                         O=n10
.gate  aoi21   a=in2  b=n10  c=in0           O=n11
.gate  iao22   a=in1  b=n11  c=in1  d=n11   O=n12
.gate  oai21   a=in2  b=n10  c=n12           O=n13
.gate  moai22  a=in0  b=n13  c=in0  d=n13   O=n14
.gate  nand    a=n12  b=n14                  O=n15
.gate  iao22   a=n9   b=n15  c=n9   d=n15   O=out1
.gate  invx    a=out1                        O=n17
.gate  nor     a=in3  b=out1                 O=n18
.gate  moai22  a=n12  b=n18  c=n12  d=n18   O=out3
.gate  aoi21   a=n17  b=out3 c=n14           O=n20
.gate  iao22   a=in3  b=n20  c=in3  d=n20   O=out2
.gate  nand    a=out3 b=out2                 O=n22
.gate  moai22  a=n14  b=n22  c=n14  d=n22   O=out0
.end
```

Figure 17: Combination of LIGHTER and ABC: Area-optimized hardware implementation of TWINE Sbox using 22.50 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_present
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  nor     a=in1  b=in3                  O=n9
.gate  iao22   a=in2  b=n9   c=in2  d=n9    O=n10
.gate  nor     a=in1  b=n10                  O=n11
.gate  iao22   a=in0  b=n11  c=in0  d=n11   O=n12
.gate  iao22   a=in3  b=n12  c=in3  d=n12   O=n13
.gate  iao21   a=n10  b=n12  c=n13           O=n14
.gate  moai22  a=in1  b=n14  c=in1  d=n14   O=out2
.gate  nor     a=n12  b=out2                 O=n16
.gate  moai22  a=n10  b=n16  c=n10  d=n16   O=out1
.gate  moai22  a=n10  b=n13  c=n10  d=n13   O=out3
.gate  nor3    a=out2 b=out3 c=out1          O=n19
.gate  moai22  a=n12  b=n19  c=n12  d=n19   O=out0
.end
```

Figure 18: Combination of LIGHTER and ABC: Area-optimized hardware implementation of PRESENT Sbox using 22.50 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_rectangle
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  invx    a=in2                    O=n9
.gate  nor     a=in2  b=in3             O=n10
.gate  iao22   a=in0  b=n10  c=in0  d=n10  O=n11
.gate  iao22   a=in1  b=n11  c=in1  d=n11  O=n12
.gate  invx    a=n12                    O=n13
.gate  oai22   a=in2  b=n12  c=n9   d=n13  O=out3
.gate  invx    a=out3                   O=n15
.gate  oai22   a=in0  b=in2  c=n9   d=in3  O=n16
.gate  invx    a=n16                    O=n17
.gate  iao22   a=n15  b=n17  c=n11  d=n15  O=out0
.gate  nand    a=n11  b=out3            O=n19
.gate  invx    a=n19                    O=n20
.gate  ao22    a=n16  b=n19  c=in3  d=n20  O=out1
.gate  oai22   a=n13  b=n17  c=n12  d=n16  O=out2
.end
```

Figure 19: Combination of `LIGHTER` and ABC: Area-optimized hardware implementation of `Rectangle` Sbox using 19.00 GE with the standard cell library of the TSMC 65nm logic process.

```
.model   sbox_lblock_S0
.inputs  in0 in1 in2 in3
.outputs out0 out1 out2 out3
.gate  invx    a=in0                    O=n9
.gate  nor     a=in0  b=in1             O=n10
.gate  nand    a=in2  b=n10             O=n11
.gate  oai21   a=in2  b=n10  c=n11      O=n12
.gate  iao22   a=in3  b=n12  c=in3  d=n12  O=out3
.gate  nand    a=in1  b=out3            O=n14
.gate  iao22   a=in3  b=n14  c=in3  d=n14  O=n15
.gate  nand    a=n9   b=n15             O=n16
.gate  invx    a=n16                    O=n17
.gate  oai21   a=n9   b=n15  c=n16      O=out2
.gate  nor     a=out3 b=out2            O=n19
.gate  moai22  a=in0  b=n19  c=in0  d=n19  O=out0
.gate  iao22   a=in1  b=n17  c=in1  d=n17  O=out1
.end
```

Figure 20: Combination of `LIGHTER` and ABC: Area-optimized hardware implementation of `LBlock` Sbox S_0 using 17.50 GE with the standard cell library of the TSMC 65nm logic process.