IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2024, No. 1, pp. 441–458. DOI:10.46586/tosc.v2024.i1.441-458

Design of a Linear Layer Optimised for Bitsliced 32-bit Implementation

Gaëtan Leurent and Clara Pernot

Inria, Paris, France
gaetan.leurent@inria.fr, clpernot@laposte.net

Abstract. The linear layer of block ciphers plays an important role in their security. In particular, ciphers designed following the wide-trail strategy use the branch number of the linear layer to derive bounds on the probability of linear and differential trails.

At FSE 2014, the LS-design construction was introduced as a simple and regular structure to design bitsliced block ciphers. It considers the internal state as a bit matrix, and applies alternatively an identical S-Box on all the columns, and an identical L-Box on all the lines. Security bounds are derived from the branch number of the L-Box.

In this paper, we focus on bitsliced linear layers inspired by the LS-design construction and the Spook AEAD algorithm. We study the construction of bitsliced linear transformations with efficient implementations using XORs and rotations (optimized for bitsliced ciphers implemented on 32-bit processors), and a high branch number. In order to increase the density of the activity patterns, the linear layer is designed on the whole state, rather than using multiple parallel copies of an L-Box.

Our main result is a linear layer for 128-bit ciphers with branch number 21, improving upon the best 32-bit transformation with branch number 12, and the one of Spook with branch number 16.

Keywords: Bitsliced cipher · Linear layer · Branch number

1 Introduction

Block ciphers are one of the most versatile primitives in symmetric cryptography. In particular, the AES [AES01] is currently used to secure the majority on online communication. Most block ciphers are constructed with linear layers and SBoxes: the linear layer mixes the state to provide diffusion; while the SBoxes apply a non-linear operation in parallel to small chunks of the state to provide confusion. This echoes the confusion and diffusion properties of Shannon [Sha45].

A common strategy to design a block cipher is to alternate substitution layers, linear layers, and key additions. This is denoted SPN, and many common ciphers follow this construction: AES [AES01], PRESENT [BKL+07], Skinny [BJK+16], GIFT [BPP+17], ... In order to evaluate the security of those ciphers against differential and linear cryptanalysis, designers typically show bounds on the number of active SBoxes in a differential or linear trail. Combined with the cryptographic properties of the SBox (differential uniformity and linearity), this implies bounds on the probability (or bias) of linear and differential trails. Over the years, various methods have been proposed to obtain bounds on the number of active SBoxes, such as running Matsui's branch and bound algorithm [Mat95], using the wide-trail strategy [DR01], or modeling the search as a MILP problem.

Received: 2023-11-23 Accepted: 2024-01-23 Published: 2024-03-01

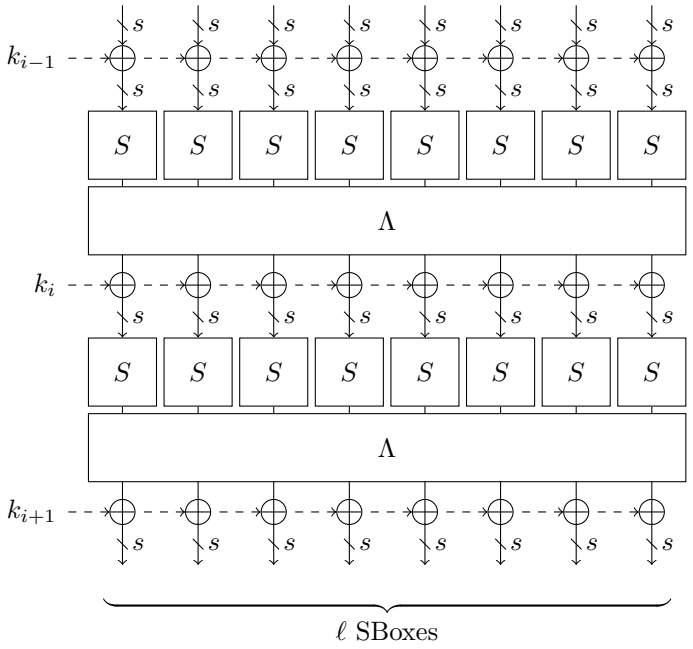

Figure 1: A SPN cipher

1.1 Design strategies for SPN ciphers

We consider an SPN cipher with ℓ parallel SBoxes, each operating on s bits; the block size is $n = \ell s$ (see Figure 1). In this paper, we denote $\mathbb{B}_s$ the set of all s-bit elements: $\mathbb{B}_s = \{0,1\}^s$. The internal state is considered as an element in $\mathbb{B}_s^\ell = (\{0,1\}^s)^\ell$, *i.e.*, a sequence of ℓ s-bit elements $x = (x_0, x_1, \ldots, x_{\ell-1})$. We define the Hamming weight of x as $|x| = \#\{i \in \{0,1,\ldots,\ell-1\} \ : \ x_i \neq 0\}$ (Hamming weight over $\mathbb{B}_s^\ell$).

The round function iterates three transformations:

SBox layer: an invertible SBox $S : \mathbb{B}_s \to \mathbb{B}_s$ is applied in parallel to each state element

$$x_i \leftarrow S(x_i), \forall i$$

Linear layer: an invertible linear transformation $\Lambda : \mathbb{B}_s^\ell \to \mathbb{B}_s^\ell$ is applied to the state

$$(x_0, x_1, \ldots, x_{\ell-1}) \leftarrow \Lambda(x_0, x_1, \ldots, x_{\ell-1})$$

In some ciphers such as the AES, Λ is a linear operation over $\mathbb{F}_{2^s}$ (identified with $\mathbb{B}_s$) but we consider a more general definition where Λ is only required to be linear over $\mathbb{F}_2$.

Key addition: a round key $k \in \mathbb{B}_s^\ell$ is added (XORed) to the state

$$(x_0, x_1, \ldots, x_{\ell-1}) \leftarrow (x_0, x_1, \ldots, x_{\ell-1}) \oplus k$$

SPN ciphers with a full key addition are usually seen as Markov ciphers [LMM91]. Therefore, we can compute the probability of a differential characteristic (resp. correlation of a linear trail) by multiplying the probability (resp. correlation) of each round. In particular, if we obtain a lower bound on the number of active SBoxes, this translates into an upper bound for the probability of differential characteristics (resp, bias of linear trails).

1.1.1 The Wide-Trail Strategy

The wide-trail strategy [DR01] is an important method to design SPN ciphers, used in the AES and some of its predecessors. The focus of the wide-trail strategy is to design a linear layer that guarantees a large number of active SBoxes, and to select SBoxes with good cryptographic properties. When the two are combined, they provide bounds on the probability of differential characteristics (and correlation of linear trails).

The differential branch number of a linear transformation Λ is defined as

$$\mathcal{B}_d(\Lambda) = \min_{x \neq 0} \big(|\Lambda(x)| + |x|\big).$$

This is an important security property, measuring the diffusion of a linear layer: any non-trivial differential characteristics in two consecutive rounds has at least $\mathcal{B}_d(\Lambda)$ active SBoxes [DR01, Theorem 1]. Therefore, if the SBox has differential uniformity δ, the probability of any r-round differential trail is at most $(\delta/2^s)^{\mathcal{B}_d \cdot \lfloor r/2 \rfloor}$.

Similarly, the linear branch number is defined as

$$\mathcal{B}_l(\Lambda) = \min_{x \neq 0} \big(|\Lambda^\top(x)| + |x|\big),$$

where $\Lambda^\top$ is the linear transformation whose matrix representation over $\mathbb{F}_2$ is the transpose of the matrix representation of Λ (note that we must consider the matrix representation as an $\ell s \times \ell s$ matrix over $\mathbb{F}_2$, not as an $\ell \times \ell$ matrix over $\mathbb{F}_{2^s}$). Any non-trivial linear trail in two consecutive rounds has at least $\mathcal{B}_l(\Lambda)$ active SBoxes. Therefore, if the SBox has linearity λ, the expected squared correlation of any r-round linear trail is at most $(\lambda^2/2^{2s})^{\mathcal{B}_d \cdot \lfloor r/2 \rfloor}$.

We have $2 \leq \mathcal{B}_d(\Lambda) \leq \ell + 1$ and $2 \leq \mathcal{B}_l(\Lambda) \leq \ell + 1$. A linear layer is called an MDS matrix when $\mathcal{B}_d(\Lambda) = \ell + 1$; this condition is equivalent to $\mathcal{B}_l(\Lambda) = \ell + 1$. Besides, when the linear layer is orthogonal (*i.e.* when $\Lambda^\top = \Lambda^{-1}$), we have $\mathcal{B}_l(\Lambda) = \mathcal{B}_d(\Lambda)$.

Alternatively, the differential branch number can be seen as the minimal distance of the code with codewords $x \| \Lambda(x)$ for $x \in \mathbb{B}_s^\ell$. If Λ is linear over $\mathbb{F}_{2^s}$, this is a linear code over $\mathbb{F}_{2^s}$ of length 2ℓ and dimension ℓ. However, the code is only $\mathbb{F}_2$-linear in general; as an $\mathbb{F}_2$-linear code it has length $2n = 2\ell s$ and dimension $n = \ell s$, but the branch number is the minimal distance using the Hamming distance over $\mathbb{B}_s$.

In the AES, the linear layer has branch number 5. This guarantees at least 5 active SBoxes every two rounds. Moreover, the AES has a stronger property, due to the interaction between the MixColumns and ShiftRows operations: any 4-round trail has at least 25 active SBoxes (the square of the branch number). In this work we focus on two-round bounds.[1]

1.1.2 Bitsliced Ciphers and LS-Designs

Conceptually, a software implementation of an SPN cipher uses a memory cell for each SBox input (ℓ memory units of s bits), so that an SBox application corresponds to a table lookup. Alternatively, SPN ciphers can be implemented in a *bitsliced* way with s memory units of ℓ bits: a memory cell gathers bits from each SBox, and the SBox is computed with a series of arithmetic operation (AND, OR, XOR, ...). The CPU arithmetic operations operate in parallel on each bit of a memory cell, so that several SBoxes are computed simultaneously. Bitsliced implementations of AES and DES have been proposed with good performances [Bih97, KS09]. They have the advantage of avoiding table lookups; it is well known that table lookups cause side channels because of memory cache, and this leads to practical attacks [TSS+03].

[1] Obtaining bounds on more than two rounds would likely require different techniques than considered in this work. As a concrete example, [GLSV15] computes bounds over several rounds in the paragraph "From Involutive to Non-involutive Components", but their approach will not scale to 32-bit words.

In general, a bitsliced implementation requires reorganizing the memory layout. However, some ciphers are designed with a bitsliced operation in mind, and are described with a bitsliced state, such as Noekeon [DPVAR00].

The LS-designs [GLSV15] are a family of ciphers optimized for bitsliced implementation. The state is considered as an $s \times \ell$ matrix of bits; the SBox layer applies the same SBox $S : \mathbb{B}_s \to \mathbb{B}_s$ on each column, and the linear layer applies a fixed LBox $L : \mathbb{B}_\ell \to \mathbb{B}_\ell$ on each line. In practice, each line is stored in an ℓ-bit register, so that the SBox is implemented as a series of bitsliced operations. The linear layer Λ operates independently on each register, applying the same transformation $L : \mathbb{F}_2^\ell \to \mathbb{F}_2^\ell$ on each line (corresponding to a given bit of all SBoxes):

$$(x_0[j], x_1[j], \ldots, x_{\ell-1}[j]) \leftarrow L(x_0[j], x_1[j], \ldots, x_{\ell-1}[j]), \forall j < s$$

The branch number of Λ (over $\mathbb{F}_{2^s}$) is equal to the branch number of L (over $\mathbb{F}_2$). Simple bounds on the security are derived directly from the properties of the SBox and the branch number.

Two 128-bit ciphers are proposed in [GLSV15], Robin and Fantomas, using $s = 8$ and $\ell = 16$. The linear layer is build from an optimal code over $\mathbb{F}_2$ with length $2\ell = 32$ and dimension $\ell = 16$: the resulting LBox is orthogonal, with distance 8. The linear layer is implemented as a series of table lookups. Since the LBox is linear, table lookups are easy to protect against side-channel using masking.

1.1.3 Spook

Spook [BBB+20] is an AEAD scheme using primitives that build upon the ideas of LS-designs. In particular, Spook uses a 128-bit tweakable block cipher (Clyde-128) that is strongly inspired by the LS-design construction, with two important differences.

First, the LBox is designed so that it has an efficient implementation using XORs and rotations, rather than using table lookups. The block cipher PRIDE [ADK+14] also uses a linear layer optimized for implementation with XORs and rotations, but focuses on more efficient implementations and lower branch number. Clyde-128 uses $s = 4$ and $\ell = 32$ so that a natural implementation uses 32-bit words and is efficient on 32-bit processors.

Second, Clyde-128 differs from LS-designs by using an interleaved LBox operating on two ℓ-bit registers at once, rather than independently on each register. This enables a larger branch number with a modest increase in implementation cost. In particular the best known linear code with length 64 and dimension 32 has distance 12 [Gra07], but the linear layer of Clyde-128 has branch number 16 (as a reference point, the best known linear code over $\mathbb{F}_4$ with length 64 and dimension 32 has minimal distance 17). The linear layer of Clyde-128 and its inverse have an implementation with just 6 XORs and 6 rotations per word (see Algorithm 4 and 5).

1.2 Our Results

In this work we follow the strategy of the Spook linear layer to design an efficient linear layer for 128-bit ciphers with good cryptographic properties. We assume that the cipher uses bitsliced 4-bit SBoxes, and we search for a linear layer with a good implementation on 32-bit processors.

We extend the construction of Spook by considering a linear layer that operates on 3 or 4 words at a time, rather than 2. This presents two difficulties, compared to what was done by the Spook designers: bounding the branch number of such a linear layer requires a lot of computation, and the techniques to obtain an efficient implementation of the inverse are not applicable with those parameters.

Eventually, we manage to build a linear transformation with branch number 21, improving from the branch number 16 used in Spook. Our linear layer is as expensive to

Table 1: Linear transformations based on XORs and rotations operating on one to four 32-bit words. $c(L)$ corresponds to the number of XORs per 32-bit word in the implementation we propose.

LBox	w	Branch number	$c(L)$	$c(L^{-1})$	Ref
L_{32}	1	12	5	5	[Leu19]
$L_{32\times2}$	2	16	6	6	[BBB+20]
$L_{32\times3}$	3	19	6	13	Subsection 4.1
$L_{32\times4}$	4	21	6	18	Subsection 4.2

compute than Spook's in the forward direction, but the inverse is around three times more expensive. The code used to search for LBoxes is available from: https://github.com/Lbox-ToSC/lbox-search.

To compare the efficiency of the different linear transformations, we define the notion of cost c, which corresponds to the number of XORs to be performed per 32-bit word. Since our linear transformations alternate XORs and rotations, the actual implementation cost is proportional to the number of XORs.

2 Linear Transformations Based on XORs and Rotations

An efficient way to build linear transformations is to use a sequence of XORs and rotations. This has been used as a component in several symmetric ciphers, such as SHA-2 [NIS02], PRIDE [ADK+14], Spook [BBB+20], or Ascon [DEMS21]. In this paper, we use $\oplus$ to denote XOR, and $\ggg$ (resp. $\lll$) to denote circular rotation to the right (resp. left).

2.1 Operating on a Single Word

We first consider a linear transformation operating on a single ℓ-bit word, *i.e.* an LBox, implemented as a series of rotations, and XORs. For instance, the SHA-2 Σ_0 function is defined over 32-bit words as:

$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$

Another example given in Algorithm 2 reuses internal values.

By construction this type of linear transformation satisfies $L(x \ggg r) = L(x) \ggg r$, $\forall x \forall r$. Therefore, when written in matrix form, we obtain a circulant matrix: each line is a rotation of the previous line. The set of circulant matrices has a strong structure; in particular the product of two circulant matrices is a circulant matrix. A circulant matrix can be identified with a polynomial in $\mathbb{F}_2[X]/(X^\ell + 1)$, by taking the content of the first row as the coefficients; the product of two circulant matrices is the same as the polynomial product in the ring $\mathbb{F}_2[X]/(X^\ell + 1)$. In particular, a polynomial is invertible in the ring if and only if it is relatively prime with $X^\ell + 1$; when ℓ is a power of two, we have $X^\ell + 1 = (X + 1)^\ell$ and a polynomial is invertible if and only if the sum of the coefficients is non-zero. Therefore, when ℓ is a power of two, a circulant matrix is invertible if and only if the sum of the elements in the first row is non-zero, and its inverse is also a circulant matrix.

The linear code associated to the LBox is a quasi-cyclic (double circulant) code, with one block being the matrix L, and one block being the identity. This class of codes is often used in code-based cryptography because they have a compact description, and they include codes with good properties. In our case we are interested in this class because they have an efficient implementation without using lookup tables.

In this paper we focus on linear transformations defined over $\ell = 32$-bit words; a sequence of XORs and rotations defines a 32×32 binary circulant matrix. As a concrete example, the code given in Algorithm 2 corresponds to the following matrix:

$$L_{32} = \begin{bmatrix}
1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&1&1\\
1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0&0\\
0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0&0\\
0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0&0\\
0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1&0\\
1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0&1\\
0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1&0\\
1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0&1\\
0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1&0\\
1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1&1\\
1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&1\\
1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1\\
1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1&1\\
1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1\\
0&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1\\
1&0&1&1&1&1&1&0&1&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0
\end{bmatrix}$$

The branch number of this matrix can be computed easily, by evaluating it on $2^{32} - 1$ inputs: we obtain $\mathcal{B}(L_{32}) = 12$. When considered as a linear code, this matches the best known linear code with the same parameters (length 64, dimension 32, distance 12 [Gra07]). This confirms that quasi-cyclic codes are good candidates to find linear transformations with good properties.

In terms of implementation, a naive implementation of the LBox L_{32} would require 10 XORs and 10 rotations:

$$\begin{aligned} L_{32}(x) = (x) \oplus (x \ggg 1) \oplus (x \ggg 3) \oplus (x \ggg 4) \oplus (x \ggg 5) \oplus (x \ggg 6) \\ \oplus (x \ggg 7) \oplus (x \ggg 9) \oplus (x \ggg 11) \oplus (x \ggg 15) \oplus (x \ggg 16) \end{aligned}$$

However, the implementation of Algorithm 2 requires only 5 XORs and 5 rotations, by reusing some intermediate variables.

As a comparison point, a table-based implementation would use linearity of the LBox to split the computation into several sub-tables. For instance, splitting the input into 4 bytes would result in an implementation using 4 tables with 256 32-bits entries each; evaluating L_{32} requires 4 table lookups and 3 XORs, as well as operations to extract the individual bytes. On many processors, this would be slower than the implementation of Algorithm 2 using XORs and rotations. Moreover a table-based implementation creates issues related to cache timing.

2.2 Operating on Several Words

The LS-design construction uses several copies of an LBox applied independently to the bits of the SBoxes. On the other hand, Spook uses an *interleaving* construction: the linear transformation is defined on two words simultaneously. This generalization includes linear transformation with a better branch number.

In general, a linear transformation L operating on w words (with w dividing s) based on XORs and rotations corresponds to a $\ell w \times \ell w$ matrix with $\ell \times \ell$ circulant blocks $M_{i,j}$.

$$L = \begin{bmatrix} M_{1,1} & \dots & M_{1,w} \\ \vdots & \ddots & \vdots \\ M_{w,1} & \dots & M_{w,w} \end{bmatrix} \qquad \begin{aligned} L : \quad & \mathbb{B}_\ell^w \to \mathbb{B}_\ell^w \\ & (x_1, \dots x_w) \mapsto (y_1, \dots y_w) \\ & y_i = \sum M_{i,j} x_j \end{aligned}$$

Alternatively, the input of the linear transformation can be considered in interleaved form. The interleaving function $\mathcal{I}$ takes w ℓ-bit words as input and turns them into a single $w\ell$-bit word by transposing the bits:

$$\mathcal{I}(x_1, \dots x_w) = x_1[0]x_2[0] \dots x_w[0]\ x_1[1]x_2[1] \dots x_w[1]\ \dots$$

We define the interleaved matrix $\bar{L}$ of a linear transformation L as the matrix such that

$$\mathcal{I}\left(L \times (x_1, \dots x_w)\right) = \bar{L} \times \mathcal{I}(x_1, \dots x_w)$$

As a simplification, the designers of Spook [Leu19] focused on a subset of linear transformations where $\bar{L}$ is a circulant $w\ell \times w\ell$ matrix. This is a subset of the block-circulant linear transformations, because an implementation of $\bar{L}$ using rotations and XORs on one $w\ell$-bit register can be rewritten as an implementation using rotations and XOR on w ℓ-bit registers, using the following properties:

$$\mathcal{I}(x_1, x_2, \dots, x_w) \ggg wr = \mathcal{I}(x_1 \ggg r, x_2 \ggg r, \dots, x_w \ggg r)$$
$$\mathcal{I}(x_1, x_2, \dots, x_w) \ggg 1 = \mathcal{I}(x_w \ggg 1, x_1, \dots, x_{w-1})$$

As a concrete example, the Spook LBox is shown in Algorithm 4; it takes two words as input and uses a sequence of rotations and XORs to produce two words of output. It is actually built from the following interleaved implementation (given a 64-bit input x):

$a \leftarrow x \oplus \mathsf{rot}(x, 24)$
$a \leftarrow a \oplus \mathsf{rot}(a, 6)$
$a \leftarrow a \oplus \mathsf{rot}(x, 34)$
$b \leftarrow a \oplus \mathsf{rot}(a, 62)$
$a \leftarrow a \oplus \mathsf{rot}(b, 51)$
$a \leftarrow a \oplus \mathsf{rot}(b, 30)$
return $\mathsf{rot}(a, 1)$

2.3 Differential and Linear Branch Numbers

An important property of LBoxes built from a circulant matrix $\bar{L}$ is that the linear and differential branch number are equal.

We use the following notation for a circulant matrix C, with $c_0, \dots, c_{w\ell-1} \in \mathbb{B} = \{0, 1\}$:

$$C = \mathrm{Circ}(c_0, c_1, \dots, c_{w\ell-1}) = \begin{bmatrix} c_0 & c_1 & \dots & c_{w\ell-1} \\ c_{w\ell-1} & c_0 & \dots & c_{w\ell-2} \\ \vdots & \vdots & \ddots & \vdots \\ c_1 & c_2 & \dots & c_0 \end{bmatrix}$$

In particular, we have $C^\top = \mathrm{Circ}(c_0, c_1, \dots, c_{w\ell-1})^\top = \mathrm{Circ}(c_0, c_{w\ell-1}, \dots, c_1)$.

Case $w = 1$. We first consider the case $w = 1$. We obtain similar expressions for the differential and linear branch number:

$$\begin{aligned}
\mathcal{B}_d(C) &= \min_{x \neq 0} \left(\left| \mathrm{Circ}(c_0, c_1, \dots, c_{\ell-1})(x) \right| + \left| x \right| \right) \\
\mathcal{B}_l(C) &= \min_{x \neq 0} \left(\left| \mathrm{Circ}(c_0, c_1, \dots, c_{\ell-1})^\top(x) \right| + \left| x \right| \right) \\
&= \min_{x \neq 0} \left(\left| \mathrm{Circ}(c_0, c_{\ell-1}, \dots, c_1)(x) \right| + \left| x \right| \right) \\
&= \min_{x \neq 0} \left(\left| P\big(\mathrm{Circ}(c_0, c_1, \dots, c_{\ell-1})(x)\big) \right| + \left| x \right| \right),
\end{aligned}$$

with P a transformation that permutes its elements:

$$\begin{aligned} P: \quad & \mathbb{B}^{\ell} \to \mathbb{B}^{\ell} \\ & (x_0, x_1, \ldots, x_{\ell-1}) \mapsto (x_0, x_{\ell-1}, \ldots, x_1) \end{aligned}$$

Since the Hamming weight is invariant up to permutation of the elements, we have $\left|P(x)\right| = \left|x\right|$ and we obtain $\mathcal{B}_d(C) = \mathcal{B}_l(C)$.

General case. In general, the linear transformation is defined as $L(x) = \mathcal{I}^{-1}(C(\mathcal{I}(x)))$. We obtain:

$$\begin{aligned} \mathcal{B}_d(C) &= \min_{x \neq 0} \left(\left| \mathcal{I}^{-1}\big(\mathrm{Circ}(c_0, c_1, \ldots, c_{w\ell-1})(x)\big) \right| + \left|x\right| \right) \\ \mathcal{B}_l(C) &= \min_{x \neq 0} \left(\left| \mathcal{I}^{-1}\big(\mathrm{Circ}(c_0, c_1, \ldots, c_{w\ell-1})^{\top}(x)\big) \right| + \left|x\right| \right) \\ &= \min_{x \neq 0} \left(\left| \mathcal{I}^{-1}\big(\mathrm{Circ}(c_0, c_{w\ell-1}, \ldots, c_1)(x)\big) \right| + \left|x\right| \right) \\ &= \min_{x \neq 0} \left(\left| \mathcal{I}^{-1}\big(P\big(\mathrm{Circ}(c_0, c_1, \ldots, c_{w\ell-1})(x)\big)\big) \right| + \left|x\right| \right), \end{aligned}$$

with P a transformation that permutes its elements:

$$\begin{aligned} P: \quad & \mathbb{B}^{w\ell} \to \mathbb{B}^{w\ell} \\ & (x_0, x_1, \ldots, x_{w\ell-1}) \mapsto (x_0, x_{w\ell-1}, \ldots, x_1) \end{aligned}$$

We observe that P preserves the interleaved Hamming weight over $\mathbb{B}_w^{\ell}$ by writing it over $\mathbb{B}^{w\ell}$:

$$\begin{aligned} \left|x\right| &= \left|(x_0, x_1, \ldots, x_{w\ell-1})\right| \\ &= \max(x_0, x_1, \ldots, x_{w-1}) + \max(x_w, x_{w+1}, \ldots, x_{2w-1}) + \ldots \\ \left|\mathcal{I}^{-1}(x)\right| &= \left|\mathcal{I}^{-1}(x_0, x_1, \ldots, x_{w\ell-1})\right| \\ &= \max(x_0, x_{\ell}, \ldots, x_{(w-1)\ell}) + \max(x_1, x_{\ell+1}, \ldots, x_{(w-1)\ell+1}) + \ldots \\ \left|\mathcal{I}^{-1}(P(x))\right| &= \left|\mathcal{I}^{-1}(x_0, x_{w\ell-1}, \ldots, x_1)\right| \\ &= \max(x_0, x_{(w-1)\ell}, \ldots, x_{\ell}) + \max(x_{w\ell-1}, x_{(w-1)\ell-1}, \ldots, x_{w\ell-\ell-1}) + \ldots \\ &= \left|\mathcal{I}^{-1}(x)\right| \end{aligned}$$

Therefore we have $\mathcal{B}_d(L) = \mathcal{B}_l(L)$.

2.4 Search for Good LBoxes

In order to find an LBox with a good implementation, the Spook designers ran a search over potential implementations until obtaining an LBox with a good branch number [Leu19], rather than first searching a good matrix and then searching a good implementation of this matrix. The implementation is considered as a sequence of operations $x_i \leftarrow x_{a_i} \oplus \mathsf{rot}(x_{b_i}, r_i)$ (with $a_i, b_i < i$), where the input is x_0 and the output is the last computed value. Starting with a fixed number of operations k, the search algorithm generates random parameters $(a_i, b_i, r_i)_{i=1}^{k}$, and evaluates the branch number of the resulting linear transformation. The best linear transformation is kept as the result.

The most computationally intensive part of the search is to evaluate the branch number of a given LBox. Finding the branch number of L is equivalent to finding the value $x \neq 0$ that minimizes $|L(x)| + |x|$. A naive approach does exhaustive search over all possible

x; this is practical for a 32-bit linear transformation. An improved variant tries x by increasing weight: if $|L(x)| + |x| \geq b$ for all x with $|x| \leq b-2$, then the branch number is at least b (because L is invertible). We define the set $\mathcal{X}_h = \{x \in \mathbb{F}_{2^w}^{\ell} : |x| \leq h\}$ of words of weight at most h; in order to show that the branch number is at least b, we only have to search over the set $\mathcal{X}_{b-2}$. However, this is not feasible with the Spook parameters: with $\ell = 32$, $w = 2$ and branch number 16, we would have to consider $\mathcal{X}_{14}$ of size $\sum_{h=1}^{14} \binom{32}{h} 3^h \approx 2^{51.4}$.

To further reduce the search space, we observe that if $|L(x)| + |x| \leq b$ then $|x| \leq b/2$ or $|L(x)| \leq b/2$. Therefore, to show that the branch number is at least b it is sufficient to exhaustively search all x in $\mathcal{X}_{\lfloor b/2 \rfloor}$ and in $\{L^{-1}(x) : x \in \mathcal{X}_{\lfloor b/2 \rfloor}\}$. With the Spook parameters, we only have to consider $\mathcal{X}_8$ of size roughly $2^{36.2}$, so that this approach is practical. The algorithm is given as Algorithm 1. Moreover, since L is circulant, we can skip candidates that are equivalent up to rotation. This technique works well for Spook [Leu19], but it does not scale to larger values of $w > 2$.

Algorithm 1 Compute the branch number of an LBox

Input: L
  b $\leftarrow |L(1)| + 1$
  $h \leftarrow 1$
  while $h \leq b/2$ **do**
    for $x \in \mathcal{X}_h$ **do**
      if $|x| + |L(x)| < b$ **then**
        $b \leftarrow |x| + |L(x)|$
      if $|x| + |L^{-1}(x)| < b$ **then**
        $b \leftarrow |x| + |L^{-1}(x)|$
    $h \leftarrow h + 1$
  return b

2.5 Efficient Inverse

In order to use a linear transformation in an SPN cipher, it is useful to also have an efficient implementation of the inverse (this is not always a requirement; for instance the CTR mode of encryption does not use the inverse of the block cipher). When using the approach of the previous section we obtain an LBox with an efficient implementation, but we cannot derive an efficient implementation of L^{-1} from an efficient implementation of L.

Instead, the Spook designers ran a birthday search to find two efficient implementations such that the resulting matrices L and L' are each other's inverses. The space of circulant matrices is of size $2^{w\ell}$; $2^{w\ell-1}$ are invertible and after identifying matrices that are rotations of each others there are approximately $2^{w\ell-1}/w\ell$ equivalence classes. Therefore the birthday search is expected to find one matrix with efficient implementation in both directions after trying roughly $2^{w\ell/2}/\sqrt{w\ell}$ random implementations (2^{29} with the Spook parameters). In order to obtain 2^t candidates, the complexity is roughly $\sqrt{t} \times 2^{w\ell/2}/\sqrt{w\ell}$. With $\ell = 32$, this approach is feasible for $w \leq 2$ but not for $w > 2$.

2.6 Known Results

We now summarize the state of the art on efficient LBoxes defined for $\ell = 32$-bit words.

Operating on 1 word [Leu19]. When operating on one word, there are around $2^{31}/32 = 2^{26}$ invertible circulant matrices up to equivalence by rotation. The branch number of a candidate can be computed with at most $2 \times |\mathcal{X}_6| \approx 2^{21.1}$ operations, assuming it is smaller

than 12. Therefore, it is practical to exhaustively search all circulant matrices, showing that the best possible branch number is 12. This matches the best known linear codes with dimension 32 and length 64.

Moreover, there exist LBoxes in this set with an efficient inverse: Algorithm 2 and 3 give the implementation of an LBox with branch number 12 using 5 rotations and 5 XORs, and its inverse using 6 rotations and 5 XORs.

Algorithm 2 L_{32} LBox

Input: x
$a \leftarrow x \oplus \mathsf{rot}(x, 1)$
$b \leftarrow a \oplus \mathsf{rot}(a, 4)$
$a \leftarrow b \oplus \mathsf{rot}(a, 9)$
$b \leftarrow a \oplus \mathsf{rot}(x, 3)$
return $b \oplus \mathsf{rot}(a, 6)$

Algorithm 3 L_{32} LBox inverse

Input: x
$a \leftarrow x \oplus \mathsf{rot}(x, 2)$
$b \leftarrow x \oplus \mathsf{rot}(a, 3)$
$a \leftarrow b \oplus \mathsf{rot}(a, 16)$
$b \leftarrow a \oplus \mathsf{rot}(b, 6)$
return $\mathsf{rot}(a, 25) \oplus \mathsf{rot}(b, 15)$

Operating on 2 words [BBB+20, Leu19]. When operating on two words, exhaustive search is no longer possible. The Spook authors ran a search for random circulant matrices, and the best branch number obtained was 16. They also found an LBox with branch number 16 with efficient implementation and efficient inverse, which is used in Spook; Algorithm 4 gives its implementation using 13 rotations and 12 XORs, while Algorithm 5 gives its inverse using 14 rotations and 12 XORs.

Algorithm 4 Spook L-box

Input: (x, y)
$a \leftarrow x \oplus \mathsf{rot}(x, 12)$
$b \leftarrow y \oplus \mathsf{rot}(y, 12)$
$a \leftarrow a \oplus \mathsf{rot}(a, 3)$
$b \leftarrow b \oplus \mathsf{rot}(b, 3)$
$a \leftarrow a \oplus \mathsf{rot}(x, 17)$
$b \leftarrow b \oplus \mathsf{rot}(y, 17)$
$c \leftarrow a \oplus \mathsf{rot}(a, 31)$
$d \leftarrow b \oplus \mathsf{rot}(b, 31)$
$a \leftarrow a \oplus \mathsf{rot}(d, 26)$
$b \leftarrow b \oplus \mathsf{rot}(c, 25)$
$a \leftarrow a \oplus \mathsf{rot}(c, 15)$
$b \leftarrow b \oplus \mathsf{rot}(d, 15)$
$b \leftarrow \mathsf{rot}(b, 1)$
return (b, a)

Algorithm 5 Spook L-box inverse

Input: (x, y)
$a \leftarrow x \oplus \mathsf{rot}(x, 25)$
$b \leftarrow y \oplus \mathsf{rot}(y, 25)$
$c \leftarrow x \oplus \mathsf{rot}(a, 31)$
$d \leftarrow y \oplus \mathsf{rot}(b, 31)$
$c \leftarrow c \oplus \mathsf{rot}(a, 20)$
$d \leftarrow d \oplus \mathsf{rot}(b, 20)$
$a \leftarrow c \oplus \mathsf{rot}(c, 31)$
$b \leftarrow d \oplus \mathsf{rot}(d, 31)$
$c \leftarrow c \oplus \mathsf{rot}(b, 26)$
$d \leftarrow d \oplus \mathsf{rot}(a, 25)$
$a \leftarrow a \oplus \mathsf{rot}(c, 17)$
$b \leftarrow b \oplus \mathsf{rot}(d, 17)$
$a \leftarrow \mathsf{rot}(a, 15)$
$b \leftarrow \mathsf{rot}(b, 16)$
return (b, a)

3 Search for New Linear Transformations

In this work, we consider linear transformations with $\ell = 32$ for efficiency reasons, and we attempt to design linear transformations operating on more than 2 words at a time ($w > 2$). This should increase the branch number that we can achieve, but there are two issues that make the previous methods inapplicable:

- The algorithm to compute the branch number described in Section 2.4 is not applicable to $w > 2$;

- The birthday search described in Section 2.5 to find a matrix with efficient implementation and efficient inverse is not applicable to $w > 2$.

3.1 Computing the Branch Number Efficiently

As mentioned above, computing the branch number of an LBox is equivalent to computing the minimal distance of a linear code. In particular, we observe that the algorithm described in Section 2.4 is actually the classical Brouwer-Zimmermann [Zim96, BBF+06] minimum distance algorithm, applied to the code generated by $I \| \bar{L}$. As far as we know this is the best deterministic algorithm to compute the branch number of an arbitrary linear code, but coding theory offers probabilistic algorithms that are more efficient.

3.1.1 Information Set Decoding Algorithm

To efficiently compute the branch number, we use tools from coding theory. More specifically, we use an Information Set Decoding (ISD) algorithm [Pra62] to find a non-zero codeword with the lowest possible weight.

This is an iterative algorithm which, given a matrix, a weight $\mathbf{w}$, a number of iterations t, and a parameter d, returns a codeword of weight $\mathbf{w}$ if it finds one.

Case $w = 1$. We first describe how the algorithm in the case $w = 1$, so that the code is linear over the field $\mathbb{F}_2$ and the weight is the standard Hamming weight. The code has dimension ℓ and length 2ℓ, and is given as a $\ell \times 2\ell$ generating matrix.

We assume that there is a word W of weight $\mathbf{w}$ in the code. Each iteration tries to find such a word, and succeeds with some probability p. Each iteration works as follows. First, starting from the matrix $\bar{L}$, a random permutation of the columns is performed, and a Gaussian reduction is performed. After the permutation, the ℓ columns on the left are called an information set.

We hope that W has weight exactly 1 in the information set. This happens with probability:

$$p = \frac{\binom{\ell}{\mathbf{w}-1} \times \binom{\ell}{1}}{\binom{2\ell}{\mathbf{w}}}$$

In this case, W is one of the rows of the matrix, if the matrix is in row echelon form.

We have a good probability to find W after $1/p$ iterations: after $(1/p) \times \epsilon$ iterations, we find it with probability $1 - (1-p)^{(1/p)\times\epsilon} \approx 1 - e^{-\epsilon}$.

Knowing that the complexity of this algorithm is dominated by the Gaussian elimination, we consider combinations of d rows. We therefore increase the probability of finding a word of weight $\mathbf{w}$ if one exists. This succeeds if W has weight lower or equal than d in the information set. The success probability is:

$$p = \sum_{j=1}^{d} \frac{\binom{\ell}{\mathbf{w}-j} \times \binom{\ell}{j}}{\binom{2\ell}{\mathbf{w}}}$$

General case. When $w > 1$, we consider a code defined by $2^{w\ell}$ basis vectors in $\mathbb{B}_w^{2\ell}$, given a $w\ell \times 2w\ell$ matrix. The code is linear over $\mathbb{F}_2$ but the Hamming weight is defined over w-bit words. Therefore, we don't process columns and rows directly, but packets of w columns and w rows:

- instead of randomly permuting the columns, we need to swap packets of w columns, because this permutation corresponds to the selection of an information set, which is defined in association with the Hamming weight.

- instead of enumerating rows in the binary case, i.e., scanning all rows of weight 1 on the information set, here we have to scan all words of weight 1 in the information set: the $2^w - 1$ words with weight 1 on the first packet of w columns, the $2^w - 1$ words with weight 1 on the second packet of w columns, ... In the case $d = 1$, there will therefore be $\ell \times (2^w - 1)$ elements to enumerate.

Algorithm 6 Information Set Decoding Algorithm to compute the distance of a linear code

Input: C: linear code,
d: maximum number of rows to XOR to verify weight,
t: number of iterations,
w: the desired weight.
for $i = 1, ..., t$ **do**
  $C \leftarrow$ Random_permutation_of_columns (C)
  $C \leftarrow$ Gaussian_elimination (C)
  for all combination x of at most d rows **do**
    if $|x| <$ **w then**
      return x
return False

3.2 Best Distance Obtained

First, we looked for the branch number achievable for random circulant matrices: for 3 32-bit words, after 2^{27} tests, the best branch number we obtained is 19, which is achieved by 7% of random circulant matrices. For 4 32-bit words, after 2^{18} tests, the maximum we obtained is 21, and it is reached by 42% of the random circulant matrices. In the next section, we look for linear transformations with efficient implementation (based on XOR and rotations, as described above) that reach those maximums.

3.3 Implementing the Inverse

Simultaneous Search of L and L^{-1}. As explained in Section 2.5, the strategy used to find linear transformations that can be implemented as well as their inverses using a sequence of operations $x_i \leftarrow x_{a_i} \oplus \mathsf{rot}(x_{b_i}, r_i)$ is impractical for 3 or more words. So we use another strategy: we generate matrices using a sequence of operations $x_i \leftarrow x_{a_i} \oplus \mathsf{rot}(x_{b_i}, r_i)$, and then, for candidates with a high branch number, we look for an efficient implementation of their inverse. This implementation of the inverse will be less efficient than those proposed above, but nonetheless more efficient than a naive implementation.

Heuristic Search of Good Implementation. In this section, we describe how to build an implementation of the inverse from its matrix description. We use the notation $|x|_2$ to denote the binary Hamming weight (as opposed to the Hamming weight $|x|$ over $\mathbb{B}_w$). A naive implementation of an LBox L requires $|L(1)|_2$ rotations and $|L(1)|_2 - 1$ XORs, but we can use heuristics to find a more efficient implementation.

Since we work with circulant matrices, we only consider the $w\ell$-bit word corresponding to the first row of the matrix. Our approach is to start with the target $L(1)$ and to decompose it into several words. For simplicity, we only count the number of XORs in an implementation. We start with the following decomposition:

$\boldsymbol{x = a \oplus (a \ggg r) \oplus b}$**.** Given a target x, we iterate over all r $(1 \leq r < w\ell)$ and we construct a value a to obtain a small value $|a|_2 + |b|_2$ with $b = x \oplus a \oplus (a \ggg r)$.

A good candidate is $z = x \wedge (x \lll r)$; when z and $z \ggg r$ are disjoint we take $a = z$ and obtain $|b|_2 = |x|_2 - 2|a|_2$ with $|a|_2 + |b|_2$ minimal. In general, we use a greedy algorithm to iteratively build a.

We can implement a and b naively, and we obtain an implementation of x with cost $(|a|_2 - 1) + (|b|_2 - 1) + 2$; most of the time this is smaller than the cost of a naive implementation $(|x|_2 - 1)$. We can often further reduce the cost of the implementation by recursively decomposing a and b.

More generally, we build a recursive algorithm to find an implementation of a vector of target values $(x_0, \dots, x_v)$. At each step, we apply several heuristics to attempt to decompose elements of the vector. We start the algorithm with the vector $(L(1))$, and eventually we implement the final vector naively and reconstruct the target by inverting the decomposition operations.

We consider the following decompositions:

$x_i = a \oplus (a \ggg r) \oplus b$. We exhaustively consider all x_i's (and r) and use the same decomposition as above. This reduces the implementation cost if $|a|_2 + |b|_2 < |x_i|_2$.

The new vector to decompose is $(x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_v, a, b)$.

$x_i = a \oplus (x_j \ggg r)$. We exhaustively consider all pairs (x_i, x_j) and all values r, and define $a = x_i \oplus x_j \ggg r$. If $|a|_2 + 1 < |x_i|_2$ this reduces the implementation cost.

The new vector to decompose is $(x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_v, a)$.

$x_i = a \oplus b$, $x_j = (a \ggg r) \oplus c$. We exhaustively consider all pairs (x_i, x_j) and all values r, and define $a = x_i \wedge (x_j \lll r)$, $b = x_i \oplus a$, $c = x_j \oplus (a \ggg r)$. We have $|b|_2 = |x_i|_2 - |a|_2$ and $|c|_2 = |x_j|_2 - |a|_2$, so that this reduces the implementation cost when $|a|_2 > 1$.

The new vector to decompose is $(x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_{j-1}, x_{j+1}, \dots, x_v, a, b, c)$.

When several decompositions are possible, we try various heuristics to select a decomposition, from a simple greedy heuristic to more advanced heuristics (with a recursive call to the greedy heuristic) to get a more accurate estimate of the cost of the new vector.

We also considered a decomposition that adds an arbitrary bit to one of the objectives; this random noise generally increases the implementation cost but it can sometimes simplify a later decomposition.

4 Results

Using the new described methods, we managed to obtain matrices with efficient implementations that meet the best distances obtained with random circulant matrices for 3 and 4 words.

4.1 Considering 3 Words

We first consider a linear layer operating on three 32-bit words, to be used for a cipher with a 96-bit or 192-bit internal state. We found an LBox operating on three 32-bit words with branch number 19 and an efficient implementation (18 rotations and 18 XORs). It matches the best branch number observed with random circulant matrices, and the number of XORs per 32-bit word is 6, as in Spook. The implementation of the inverse has been found using the strategy described in Section 3.3: it has 42 rotations and 39 XORs. The implementations of the linear transformation and its inverse are given in algorithms 7 and 9. This candidate was tested with 2^{20} iterations of the information set decoding algorithm 6 with $\mathbf{w} = 18$ and $d = 3$. Each iteration costs $\binom{32}{3} \times 7^3 + \binom{32}{2} \times 7^2 + 32 \times 7 \approx 2^{20.7}$, and

the probability of finding a word of weight 18 at each iteration is $2^{-10.17}$. By repeating 2^{20} times, the probability of failing to find this word is $e^{-2^{20-10.17}} \approx 2^{-1313}$ if it exists.

4.2 Considering 4 Words

Next, we consider a linear layer operating on four 32-bit words, to be used for a cipher with a 128-bit internal state (our main target), or a 256-bit internal state. We found an LBox operating on four 32-bit words with branch number 21 and an efficient implementation (24 rotations and 24 XORs). It matches the best branch number observed with random circulant matrices, and the number of XORs per 32-bit word is 6, as in Spook. The implementation of the inverse has been found using the strategy described in Section 3.3: it has 76 rotations and 72 XORs. The implementations of the linear transformation and its inverse are given in algorithms 8 and 10. This candidate was tested with 2^{25} iterations of the information set decoding algorithm 6 with $\mathbf{w} = 20$ and $d = 2$. Each iteration costs $\binom{32}{2} \times 15^2 + 32 \times 15 \approx 2^{16.8}$, and the probability of finding a word of weight 20 at each iteration is $2^{-16.29}$. By repeating 2^{25} times, the probability of failing to find this word is $e^{-2^{25-16.29}} \approx 2^{-604}$ if it exists.

Algorithm 7 $L_{32\times3}$ Lbox

Input: (x, y, z)
$a \leftarrow x \oplus \mathsf{rot}(y, 17)$
$b \leftarrow y \oplus \mathsf{rot}(z, 17)$
$c \leftarrow z \oplus \mathsf{rot}(x, 18)$
$d \leftarrow a \oplus \mathsf{rot}(c, 30)$
$e \leftarrow b \oplus \mathsf{rot}(a, 31)$
$c \leftarrow c \oplus \mathsf{rot}(b, 31)$
$a \leftarrow d \oplus \mathsf{rot}(x, 21)$
$b \leftarrow e \oplus \mathsf{rot}(y, 21)$
$f \leftarrow c \oplus \mathsf{rot}(z, 21)$
$d \leftarrow d \oplus \mathsf{rot}(b, 30)$
$e \leftarrow e \oplus \mathsf{rot}(f, 30)$
$c \leftarrow c \oplus \mathsf{rot}(a, 31)$
$a \leftarrow d \oplus \mathsf{rot}(c, 23)$
$b \leftarrow e \oplus \mathsf{rot}(d, 24)$
$f \leftarrow c \oplus \mathsf{rot}(e, 24)$
$a \leftarrow a \oplus \mathsf{rot}(d, 26)$
$b \leftarrow b \oplus \mathsf{rot}(e, 26)$
$c \leftarrow f \oplus \mathsf{rot}(c, 26)$
return (a, b, c)

Algorithm 8 $L_{32\times4}$ LBox

Input: (x, y, z, t)
$a \leftarrow x \oplus \mathsf{rot}(y, 19) \oplus \mathsf{rot}(t, 24)$
$b \leftarrow y \oplus \mathsf{rot}(z, 19) \oplus \mathsf{rot}(x, 25)$
$c \leftarrow z \oplus \mathsf{rot}(t, 19) \oplus \mathsf{rot}(y, 25)$
$d \leftarrow t \oplus \mathsf{rot}(x, 20) \oplus \mathsf{rot}(z, 25)$
$e \leftarrow a \oplus \mathsf{rot}(c, 2)$
$f \leftarrow b \oplus \mathsf{rot}(d, 2)$
$g \leftarrow c \oplus \mathsf{rot}(a, 3)$
$h \leftarrow d \oplus \mathsf{rot}(b, 3)$
$i \leftarrow e \oplus \mathsf{rot}(h, 8)$
$j \leftarrow f \oplus \mathsf{rot}(e, 9)$
$k \leftarrow g \oplus \mathsf{rot}(f, 9)$
$l \leftarrow h \oplus \mathsf{rot}(g, 9)$
$e \leftarrow i \oplus \mathsf{rot}(d, 30)$
$f \leftarrow j \oplus \mathsf{rot}(a, 31)$
$g \leftarrow k \oplus \mathsf{rot}(b, 31)$
$h \leftarrow l \oplus \mathsf{rot}(c, 31)$
$a \leftarrow e \oplus \mathsf{rot}(k, 3)$
$b \leftarrow f \oplus \mathsf{rot}(l, 3)$
$c \leftarrow g \oplus \mathsf{rot}(i, 4)$
$d \leftarrow h \oplus \mathsf{rot}(j, 4)$
return (a, b, c, d)

4.3 Results on Smaller Word Sizes

Our technique can easily be applied to smaller word sizes. In Table 2, we list results on 16-bit words and 8-bit words. In particular, with 16-bit words we obtain branch number 14 with 8 words, corresponding to a 128-bit state. With 8-bit words, we obtain branch number 8 with 8 words, corresponding to an almost-MDS diffusion layer.

Algorithm 9 $L_{32\times3}$ LBox inverse

Input: (x, y, z)
$a \leftarrow \mathsf{rot}(y, 14) \oplus \mathsf{rot}(y, 26)$
$b \leftarrow \mathsf{rot}(z, 14) \oplus \mathsf{rot}(z, 26)$
$c \leftarrow \mathsf{rot}(x, 15) \oplus \mathsf{rot}(x, 27)$
$d \leftarrow \mathsf{rot}(a, 8)$
$e \leftarrow \mathsf{rot}(b, 8)$
$f \leftarrow \mathsf{rot}(c, 8)$
$a \leftarrow a \oplus \mathsf{rot}(a, 1)$
$b \leftarrow b \oplus \mathsf{rot}(b, 1)$
$c \leftarrow c \oplus \mathsf{rot}(c, 1)$
$g \leftarrow \mathsf{rot}(z, 6)$
$h \leftarrow \mathsf{rot}(x, 7)$
$i \leftarrow \mathsf{rot}(y, 7)$
$g \leftarrow g \oplus \mathsf{rot}(c, 10)$
$h \leftarrow h \oplus \mathsf{rot}(a, 11)$
$i \leftarrow i \oplus \mathsf{rot}(b, 11)$
$j \leftarrow \mathsf{rot}(x, 21)$
$k \leftarrow \mathsf{rot}(y, 21)$
$l \leftarrow \mathsf{rot}(z, 21)$
$j \leftarrow j \oplus a$
$k \leftarrow k \oplus b$
$l \leftarrow l \oplus c$
$g \leftarrow g \oplus \mathsf{rot}(e, 26)$
$h \leftarrow h \oplus \mathsf{rot}(f, 26)$
$i \leftarrow i \oplus \mathsf{rot}(d, 27)$
$a \leftarrow \mathsf{rot}(y, 1) \oplus \mathsf{rot}(z, 22)$
$b \leftarrow \mathsf{rot}(z, 1) \oplus \mathsf{rot}(x, 23)$
$c \leftarrow \mathsf{rot}(x, 2) \oplus \mathsf{rot}(y, 23)$
$j \leftarrow j \oplus \mathsf{rot}(c, 18)$
$k \leftarrow k \oplus \mathsf{rot}(a, 19)$
$l \leftarrow l \oplus \mathsf{rot}(b, 19)$
$d \leftarrow d \oplus a \oplus \mathsf{rot}(d, 8)$
$e \leftarrow e \oplus b \oplus \mathsf{rot}(e, 8)$
$f \leftarrow f \oplus c \oplus \mathsf{rot}(f, 8)$
$a \leftarrow d \oplus g \oplus \mathsf{rot}(e, 11)$
$b \leftarrow e \oplus h \oplus \mathsf{rot}(f, 11)$
$c \leftarrow g \oplus i \oplus \mathsf{rot}(d, 12)$
$a \leftarrow a \oplus j \oplus \mathsf{rot}(a, 30)$
$b \leftarrow b \oplus k \oplus \mathsf{rot}(b, 30)$
$c \leftarrow c \oplus l \oplus \mathsf{rot}(c, 30)$
return (a, b, c)

Algorithm 10 $L_{32\times4}$ LBox inverse

Input: (x, y, z, t)
$a \leftarrow \mathsf{rot}(t, 2) \oplus \mathsf{rot}(x, 28)$
$b \leftarrow \mathsf{rot}(x, 3) \oplus \mathsf{rot}(y, 28)$
$c \leftarrow \mathsf{rot}(y, 3) \oplus \mathsf{rot}(z, 28)$
$d \leftarrow \mathsf{rot}(z, 3) \oplus \mathsf{rot}(t, 28)$
$e \leftarrow \mathsf{rot}(y, 15) \oplus a$
$f \leftarrow \mathsf{rot}(z, 15) \oplus b$
$g \leftarrow \mathsf{rot}(t, 15) \oplus c$
$h \leftarrow \mathsf{rot}(x, 16) \oplus d$
$a \leftarrow \mathsf{rot}(y, 25) \oplus \mathsf{rot}(a, 21)$
$b \leftarrow \mathsf{rot}(z, 25) \oplus \mathsf{rot}(b, 21)$
$c \leftarrow \mathsf{rot}(t, 25) \oplus \mathsf{rot}(c, 21)$
$d \leftarrow \mathsf{rot}(x, 26) \oplus \mathsf{rot}(d, 21)$
$i \leftarrow \mathsf{rot}(x, 9) \oplus \mathsf{rot}(t, 19)$
$j \leftarrow \mathsf{rot}(y, 9) \oplus \mathsf{rot}(x, 20)$
$k \leftarrow \mathsf{rot}(z, 9) \oplus \mathsf{rot}(y, 20)$
$l \leftarrow \mathsf{rot}(t, 9) \oplus \mathsf{rot}(z, 20)$
$m \leftarrow i \oplus \mathsf{rot}(k, 2) \oplus \mathsf{rot}(e, 7)$
$n \leftarrow j \oplus \mathsf{rot}(l, 2) \oplus \mathsf{rot}(f, 7)$
$o \leftarrow k \oplus \mathsf{rot}(i, 3) \oplus \mathsf{rot}(g, 7)$
$p \leftarrow l \oplus \mathsf{rot}(j, 3) \oplus \mathsf{rot}(h, 7)$
$q \leftarrow \mathsf{rot}(t, 3) \oplus \mathsf{rot}(y, 7)$
$r \leftarrow \mathsf{rot}(x, 4) \oplus \mathsf{rot}(z, 7)$
$s \leftarrow \mathsf{rot}(y, 4) \oplus \mathsf{rot}(t, 7)$
$u \leftarrow \mathsf{rot}(z, 4) \oplus \mathsf{rot}(x, 8)$
$e \leftarrow e \oplus \mathsf{rot}(q, 12)$
$f \leftarrow f \oplus \mathsf{rot}(r, 12)$
$g \leftarrow g \oplus \mathsf{rot}(s, 12)$
$h \leftarrow h \oplus \mathsf{rot}(u, 12)$
$a \leftarrow a \oplus q \oplus \mathsf{rot}(r, 3)$
$b \leftarrow b \oplus r \oplus \mathsf{rot}(s, 3)$
$c \leftarrow c \oplus s \oplus \mathsf{rot}(u, 3)$
$d \leftarrow d \oplus u \oplus \mathsf{rot}(q, 4)$
$q \leftarrow a \oplus e \oplus \mathsf{rot}(d, 0)$
$r \leftarrow b \oplus f \oplus \mathsf{rot}(a, 1)$
$s \leftarrow c \oplus g \oplus \mathsf{rot}(b, 1)$
$u \leftarrow d \oplus h \oplus \mathsf{rot}(c, 1)$
$k \leftarrow \mathsf{rot}(x, 1) \oplus \mathsf{rot}(t, 26) \oplus \mathsf{rot}(k, 4)$
$l \leftarrow \mathsf{rot}(y, 1) \oplus \mathsf{rot}(x, 27) \oplus \mathsf{rot}(l, 4)$
$i \leftarrow \mathsf{rot}(z, 1) \oplus \mathsf{rot}(y, 27) \oplus \mathsf{rot}(i, 5)$
$j \leftarrow \mathsf{rot}(t, 1) \oplus \mathsf{rot}(z, 27) \oplus \mathsf{rot}(j, 5)$
$a \leftarrow k \oplus m \oplus \mathsf{rot}(m, 12)$
$b \leftarrow l \oplus n \oplus \mathsf{rot}(n, 12)$
$c \leftarrow i \oplus o \oplus \mathsf{rot}(o, 12)$
$d \leftarrow j \oplus p \oplus \mathsf{rot}(p, 12)$
$m \leftarrow q \oplus a \oplus \mathsf{rot}(c, 18)$
$n \leftarrow r \oplus b \oplus \mathsf{rot}(d, 18)$
$o \leftarrow s \oplus c \oplus \mathsf{rot}(a, 19)$
$p \leftarrow u \oplus d \oplus \mathsf{rot}(b, 19)$
return (m, n, o, p)

Table 2: Linear transformations based on XORs and rotations operating on 8-bit words and 16-bit words. $c(L)$ is the number of XORs per word.

LBox	ℓ	w	$\mathcal{B}$	$c(L)$
L_8	8	1	4	2
$L_{8\times2}$	8	2	6	3
$L_{8\times4}$	8	4	7	4
$L_{8\times6}$	8	6	7	4
$L'_{8\times6}$	8	6	8	5
$L_{8\times8}$	8	8	8	4

LBox	ℓ	w	$\mathcal{B}$	$c(L)$
L_{16}	16	1	8	4
$L_{16\times2}$	16	2	10	5
$L_{16\times4}$	16	4	12	5
$L_{16\times6}$	16	6	13	5
$L_{16\times8}$	16	8	14	6

5 Conclusion

We obtain a 128-bit linear transformation which is a very good candidate for the linear layer of a bitsliced 128-bit cipher with 32 4-bit SBoxes. It has an efficient implementation as a series of 32-bit XORs and rotations, and a branch number of 21.

As a comparison, our new linear layer has a better branch number than the one used in Spook (with branch number 16), and the implementation in the forward direction has the same efficiency. However, the inverse is about three time more expensive. Using our linear layer in Spook would significantly improve the security margin, with a limited impact on performance (because Spook only requires the inverse for the tag verification).

We obtain this result by designing an LBox operating on all four state words, while Spook uses two copies of an LBox operating on 2 words. We had to solve two problems. Firstly, how to compute efficiently the branch number of a linear transformation? For this, we used tools from coding theory: an Information Set Decoding algorithm is used to compute the branch number of a linear transformation. Secondly, how to find linear transformations with an efficient implementation and/or whose inverse has an efficient implementation? To do this, we proposed linear transformations from sequences of XORs and rotations of 32-bit words.

References

[ADK+14] Martin R. Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçin. Block ciphers - focus on the linear layer (feat. PRIDE). In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 57–76. Springer, Heidelberg, August 2014.

[AES01] Advanced Encryption Standard (AES). National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce, November 2001.

[BBB+20] Davide Bellizia, Francesco Berti, Olivier Bronchain, Gaëtan Cassiers, Sébastien Duval, Chun Guo, Gregor Leander, Gaëtan Leurent, Itamar Levi, Charles Momin, Olivier Pereira, Thomas Peters, François-Xavier Standaert, Balazs Udvarhelyi, and Friedrich Wiemer. Spook: Sponge-based leakage-resistant authenticated encryption with a masked tweakable block cipher. *IACR Trans. Symm. Cryptol.*, 2020(S1):295–349, 2020.

[BBF+06] Anton Betten, Michael Braun, Harald Fripertinger, Adalbert Kerber, Axel Kohnert, and Alfred Wassermann. *Error-correcting linear codes: Classification*

by isometry and applications, volume 18. Springer Science & Business Media, 2006.

[Bih97] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 260–272. Springer, Heidelberg, January 1997.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 123–153. Springer, Heidelberg, August 2016.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, pages 450–466. Springer, Heidelberg, September 2007.

[BPP+17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present (full version). Cryptology ePrint Archive, Report 2017/760, 2017. https://eprint.iacr.org/2017/760.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34(3):33, July 2021.

[DPVAR00] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie proposal: Noekeon. October 2000. http://gro.noekeon.org/Noekeon-spec.pdf.

[DR01] Joan Daemen and Vincent Rijmen. The wide trail design strategy. In Bahram Honary, editor, *8th IMA International Conference on Cryptography and Coding*, volume 2260 of *LNCS*, pages 222–238. Springer, Heidelberg, December 2001.

[GLSV15] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. LS-designs: Bitslice encryption for efficient masked software implementations. In Carlos Cid and Christian Rechberger, editors, *FSE 2014*, volume 8540 of *LNCS*, pages 18–37. Springer, Heidelberg, March 2015.

[Gra07] Markus Grassl. Bounds on the minimum distance of linear codes and quantum codes. Online available at http://www.codetables.de, 2007. Accessed on 2023-08-07.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 1–17. Springer, Heidelberg, September 2009.

[Leu19] Gaëtan Leurent. Efficient linear layers for spook. Slides from the "Spook day" worfkshop, July 2019. https://www.spook.dev/assets/workshop/leurent.pdf.

[LMM91] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 17–38. Springer, Heidelberg, April 1991.

[Mat95] Mitsuru Matsui. On correlation between the order of S-boxes and the strength of DES. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 366–375. Springer, Heidelberg, May 1995.

[NIS02] NIST. Secure hash standard. FIPS 180-2, August 2002.

[Pra62] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Trans. Inf. Theory*, 8(5):5–9, 1962.

[Sha45] Claude E Shannon. A mathematical theory of cryptography. *Mathematical Theory of Cryptography*, 1945.

[TSS+03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 62–76. Springer, Heidelberg, September 2003.

[Zim96] Karl-Heinz Zimmermann. Integral hecke modules, integral generalized Reed-Muller codes, and linear codes. Technical report, Techn. Univ. Hamburg-Harburg, November 1996.