Three Input Exclusive-OR Gate Support For Boyar-Peralta's Algorithm

(Extended Version)

Anubhab Baksi[1], Vishnu Asutosh Dasu[2], Banashri Karmakar[3], Anupam Chattopadhyay[1], and Takanori Isobe[4]

[1] Nanyang Technological University, Singapore

[2] TCS Research and Innovation, Bangalore, India

[3] Indian Institute of Technology, Bhilai, India

[4] University of Hyogo, Kobe, Japan

anubhab001@e.ntu.edu.sg, vishnu.dasu1@tcs.com, banashrik@iitbhilai.ac.in, anupam@ntu.edu.sg, takanori.isobe@ai.u-hyogo.ac.jp

Abstract. The linear layer, which is basically a binary non-singular matrix, is an integral part of cipher construction in a lot of private key ciphers. As a result, optimising the linear layer for device implementation has been an important research direction for about two decades. The Boyar-Peralta's algorithm (SEA'10) is one such common algorithm, which offers significant improvement compared to the straightforward implementation. This algorithm only returns implementation with XOR2 gates, and is deterministic. Over the last couple of years, some improvements over this algorithm has been proposed, so as to make support for XOR3 gates as well as make it randomised. In this work, we take an already existing improvement (Tan and Peyrin, TCHES'20) that allows randomised execution and extend it to support three input XOR gates. This complements the other work done in this direction (Banik et al., IWSEC'19) that also supports XOR3 gates with randomised execution. Further, noting from another work (Maximov, Eprint'19), we include one additional tie-breaker condition in the original Boyar-Peralta's algorithm. Our work thus collates and extends the state-of-the-art, at the same time offers a simpler interface. We show several results that improve from the lastly best-known results.

Keywords: implementation · block cipher · linear layer

1 Introduction

With the rapid growth of lightweight cryptography in recent times, it becomes essential to reduce the cost of the cipher components. The linear layer is responsible for spreading the diffusion to the entire state in a lot of modern ciphers, thus constituting an integral part in cipher construction. Indeed, together with an SBox, it constitutes the unkeyed permutation of a cipher, which is then analysed against the common classical attacks (like differential, linear, algebraic etc.). Without loss of generality, a linear layer can be expressed as a binary non-singular matrix (e.g., `AES MixColumn` can be expressed as a 32×32 binary non-singular matrix), it can be implemented using assignment operations (software) or wiring (hardware) with XOR gates only.

While finding the naïve XOR implementation (the so-called d-XOR representation, see Definition 1), finding an optimal implementation using XOR gates is a complex problem. The Boyar-Peralta's algorithm [12] is an important step in this direction, which aims at finding efficient implementation of a given linear layer by using XOR2 gates only. The original version is

This paper combines and extends from [4, 5, 9], and the primary version with the same title is accepted in Indocrypt 2021. The first author would like to thank Sylvain Guilley (Télécom-Paris; Secure-IC) for providing the gate costs in the STM 130nm (ASIC4) library.

presented over a decade ago, but there is a renewed interest as can be seen from a number of recent follow-ups [9, 20, 23].

The main motivation for this work comes from an observation made in [9] that using higher input XOR gates may lead to reduced area in certain ASIC libraries. In particular, the authors in [9] make a randomised variation to the original Boyar-Peralta's algorithm and do a post-processing to the output to fit XOR3 gates.

Continuing in this line, we show a dynamic higher input XOR support to a randomised variation atop the original Boyar-Peralta's algorithm (this variation is taken from [23] and is referred to as RNBP). This allows us for native support for XOR3, XOR4 etc. gates, while taking care of the individual costs for each gate. This extends from the XOR3 support in [9] as this modification does not take into consideration the costs for the XOR3 and XOR2 gates, meaning it will return the same implementation no matter the costs of the XOR2 and XOR3 gates.

Contribution

In a nutshell, we present the first open-source[1] work to support higher input XOR gates that allows for efficient implementation for the linear layer. As far we know, this is the first and so-far only available project (other open-source projects like [19, 23, 24]) only consider XOR2 operation; and the source-code for [9] is not public).

Several considerations and design choices are made in our implementation. The following major changes mark our contribution:

1. We take into account all the patches/updates made to the Boyar-Peralta's algorithm [12], namely [9, 20, 23]. We implement our version of the algorithm on top of taking ideas from all of those.
   (a) It is reported in [20] that, a tie-breaker inside the original Boyar-Peralta's algorithm [12] picks only that case which maximises certain condition (see Section 3 for more details), does not (always) result in the lowest cost. It is suggested to use that case which minimises certain condition instead in [20]. We use both the maximisation and minimisation variants.
   (b) In the original Boyar-Peralta's algorithm, the tie cases are broken based on lexicography. It results in a deterministic execution, meaning the exact same representation is returned all the time. Two randomised variations are presented, in [23] and in [9]. We use the fastest implementation, called RNBP, from [23] (as the source code is public) and use the randomisation described in [9] on top of it (the corresponding source code is not public).
2. We adopt the XOR3 support in Boyar-Peralta's algorithm in [9] and provide a native interface for it in our implementation[2]. In addition, we propose a support for higher input XOR gates (which can directly work with XOR3, XOR4 etc.). The new higher input XOR support that we present is dynamic in the sense that it takes into account the exact cost for each gates. This is not the case for [9], where the same XOR3 implementation is given disregarding the cost for XOR3 gates.

With our implementation, we present several results that improve the state-of-the-art bounds with {XOR2, XOR3} gates. In total, we show the costs for five libraries, namely gate count (GC), STM 90nm (ASIC1), STM 65nm (ASIC2), TSMC 65nm (ASIC3) and STM 130nm (ASIC4), see Section 2.4 for the respective costs for the gates in the library. More details on the results can be

[1] Available at https://bitbucket.org/vdasu_edu/boyar-peralta-xor3/.

[2] The algorithm in [9], with the kind permission from the authors, is available within our implementation (the relevant source-code is written by us).

found in Section 5, here we mention a few which set the new state-of-the-art. For `AES MixColumn`, we get the least cost in b_2 (see Definition 6) for GC (59 with depth 4, down from 67 with depth 6) and for ASIC4 (258.98 GE). For the `TWOFISH` [21] and `JOLTIK-BC` [15] linear layers, we either touch or improve the benchmarks for all the five libraries.

2 Background and Prerequisite

2.1 Notions of XOR Count

Three notions for XOR count are mentioned in the literature; namely d-XOR, s-XOR, and g-XOR [16, 17, 18, 24]. Those names are shorthand notations for 'direct XOR', 'sequential XOR', and 'general XOR', respectively. As our aim is to support higher input XOR gates, we need to generalise the definitions.

In order to do that, first we present the respective definitions in Definitions 1, 2 and 3. Then we extend the definitions of by an additional parameter ϵ. Instead of the term general XOR (introduced in [24]), we use the term branch XOR (b-XOR for short) instead. Since we are generalising the pre-existing definitions, we argue it sounds better to call 'generalised branch XOR' than 'generalised general XOR'. The term 'branch' indicates that there can be branches (i.e., feed-forward paths) in this implementation.

Definition 1 (d-XOR Count). *The d-XOR count of the binary matrix $M^{m\times n}$ is defined as $d(M) = \mathrm{HW}(M) - m$, where $\mathrm{HW}(\cdot)$ denotes the Hamming weight. The corresponding implementation is referred to as the d-XOR representation.*

Definition 2 (s-XOR Count). *A binary non-singular matrix $M^{n\times n}$, can be implemented by a sequence of in-place XOR operations of the form: $x_i \leftarrow x_i \oplus x_j$ for $0 \le i, j \le n-1$. The s-XOR count is defined as the minimum number of XOR operations of this form. Any representation that conforms to this implementation is referred to a s-XOR representation.*

Definition 3 (g-XOR Count). *A given binary $M^{m\times n}$ matrix can be implemented as a sequence of equations either of the form: $a_i \leftarrow b_i \oplus c_i$ (1 XOR operation is needed), or $a_i \leftarrow b_i$ (no XOR operation is needed). The representation is called a g-XOR representation and the minimum number of XOR operations needed is referred to as the g-XOR count of M.*

The definitions for s-XOR (to s_ϵ-XOR) and g-XOR (to b_ϵ-XOR) are given subsequently in Definitions 5 and 6. To facilitate the definition for s_ϵ-XOR, we define the 'ϵ-addition matrix' in Definition 4. Note that the case for $\epsilon = 1$ is referred to as the 'addition matrix' in the literature [18].

Definition 4 (ϵ-addition Matrix). *Let $I^{n\times n}$ be the identity matrix and $E_{i,j}^{n\times n}$ be null matrix except for $E[i,j] = 1$ for some i, j over $\mathbb{F}_2$. Then $A_\epsilon = I + E_{i,j_1} + \cdots + E_{i,j_\epsilon}$ for distinct $\{i, j_1, \ldots, j_\epsilon\}$, is defined an ϵ-addition matrix where $\epsilon \ge 1$.*

Definition 5 (s_ϵ-XOR Count). *Given a cost vector $c = [c_0, c_1, \ldots, c_\epsilon]$ where $\epsilon \ge 1$ and $c_i \ge 0\ \forall i$, the s_ϵ-XOR count, of the non-singular matrix $M^{n\times n}$ over $\mathbb{F}_2$, is defined as*

$$\min\left(c_0 + c_1 e_1 + \cdots + c_\epsilon e_\epsilon\right),$$

provided M can be expressed as a product of the factor matrices from the multi-set (with the given multiplicity) in any order:

$$[P, \underbrace{A_1, \ldots, A_1}_{e_1 \text{ times}}, \ldots, \underbrace{A_\epsilon, \ldots, A_\epsilon}_{e_\epsilon \text{ times}}],$$

where $A_\epsilon^{n\times n}$'s are ϵ-addition matrices, and $P^{n\times n}$ is a permutation matrix. Here c_0 is the cost for P, and equals to 0 if P is identity.

The s_ϵ-XOR notion coincides with s-XOR when $\epsilon = 1$ and the cost vector is $[0, 1]$. Since this is the most common cost vector, it is assumed intrinsically unless mentioned otherwise. The permutation matrix P can be implemented as a wire in hardware, which effectively takes zero area, but it can take few clock cycles in software. Thus to generalize, we consider a non-negative cost for P.

Definition 6 (b_ϵ-XOR Count). *Given a cost vector $c = [c_0, c_1, \ldots, c_\epsilon]$ where $\epsilon \geq 1$ and $c_i \geq 0\ \forall i$, the b_ϵ-XOR count of the matrix $M^{m\times n}$ over $\mathbb{F}_2$ is defined as*

$$\min\left(c_0 e_0 + c_1 e_1 + \cdots + c_\epsilon e_\epsilon\right),$$

given M can be expressed by using equations of the following types (with the frequency for each type as mentioned) in any order:

$$\begin{aligned}
&t_i = t_{j_0} && \}\ e_0 \text{ times,}\\
&t_i = t_{j_0} \oplus t_{j_1} && \}\ e_1 \text{ times,}\\
&\vdots\\
&t_i = t_{j_0} \oplus t_{j_1} \oplus t_{j_2} \oplus \cdots \oplus t_{j_\epsilon} && \}\ e_\epsilon \text{ times.}
\end{aligned}$$

The notion of b_ϵ-XOR coincides with that of general XOR or the g-XOR for short [24] when $\epsilon = 1$. Similar to the case of s_ϵ, the cost for the assignment operation can be taken as zero in hardware, but it is likely not zero in software.

Example 1. Consider the binary matrix, $M^{5\times 5} = \begin{pmatrix} 1&0&0&0&0\\ 0&1&0&0&0\\ 1&1&1&0&0\\ 1&1&0&1&0\\ 1&1&0&0&1 \end{pmatrix}$.

With the modelling in [6, Section 3], we confirm that the s_1-XOR count of M is 5 with one addition matrix having multiplicity of 2. Here, $d(M) = 6, s_1(M) = 5$ and $b_1(M) = 4$ (obtained using the Boyar-Peralta's algorithm [23]. Hence for this case, $b_1(M) < s_1(M) < d(M)$.

It may be noted that the notion of b_ϵ covers that of d-XOR as well as s_ϵ, thus $b_\epsilon(M) \leq \min(d(M), s_\epsilon(M))$. Moreover, s_ϵ is undefined if the matrix is singular or rectangular. While we are not aware of use of a singular matrix, rectangular matrices are used in coding theory, which has application to cryptography.

On the other hand, s_ϵ allows to implement in-place [24]. Also, the best known result on the `AES MixColumn` takes 92 XOR2 gates (a b_1 representation by [20]), but one s_1 representation is also available at the same cost, thanks to [24]. Later, the authors of [19] manage to reduce the cost to 91 with a b_1 representation.

Further, implementations that follow the s_ϵ representation will be useful in the context of reversible computing (this includes quantum computing). There have been a few research works regarding reversible (including quantum) implementation of symmetric key ciphers recently, like [3, 14].

As for the application of the d-XOR count, one may note that this is the simplest among all the notions and the fastest to compute. Ergo, it may be useful when finding the cost for a large number of linear layers with an automated tool.

Results on `AES MixColumn`. With regard to the state-of-the-art progress on the `AES` linear layer (i.e., `MixColumn`), it can be stated that the Boyar-Peralta's algorithm [12] and its variants such as [9, 20, 23] attempt to find b_1 (i.e., returns a solution in the b_1 representation). The FSE'20 paper [24] attempts to find an s_1 representation (best known result in this category); and the CT-RSA'21 paper [19] uses a b_1 representation to get a cost of 91 (also the best known result in this category). Additionally, the authors of [9] implement the `AES MixColumn` using XOR2 and XOR3 gates, which falls into the realm of b_2 representation with the associated cost vectors $[0, 2, 3.25]$ for STM 90nm and $[0, 1.981, 3.715]$ for STM 65nm CMOS libraries, respectively (these libraries are indicated as ASIC1 and ASIC2 respectively in this work). An overview of notable works in recent times that implement `AES MixColumn`, including our own results, can be found in Table 1.

Table 1: Summary of recent `AES MixColumn` implementations

	Representation	# XOR2	# XOR3	Depth	GC
Banik, Funabiki, Isobe [9]	b_1	95	–	6	95
(Available within this work)	b_2	39	28	6	67
Tan, Peyrin [23]	b_1	94	–	9	94
Maximov [20]	b_1	92	–	6	92
Xiang, Zeng, Lin, Bao, Zhang [24]	s_1	92	–	6	92
Lin, Xiang, Zeng, Zhang [19]	b_1	91	–	7	91
Exclusively in this work	b_2	12	47	4	59

2.2 Straight Line Program (SLP)

The implementation of the linear circuits is generally shown as a sequence of operations where every step is of the form: $u \leftarrow \bigoplus_{i=1}^{\epsilon} \lambda_i v_i$ where $\lambda_i \in \{0, 1\}\ \forall i$ are constants and rest are variables. Note that, it inherently captures multi-input XOR gates. This definition is captured from [12] (it introduces the Boyar-Peralta's algorithm). Note that it coincides with the g-XOR representation given $\epsilon = 1$ (Definition 3), or with the b_ϵ-XOR representation (Definition 6).

Not clear why, but an agreed-upon terminology appears to be non-existent. The original paper that presents the Boyar-Peralta's algorithm [12] uses the term, 'linear straight line program'; the IWSEC'19 paper [9] uses the term, 'shortest linear program'; the TCHES'20 paper [23] uses the term 'short linear program' in the title. We use the term 'straight line program' (adopting from [12]) and 'SLP' as its abbreviation.

2.3 Depth

The depth of a logical circuit can be defined as the number of combinational logic gates along the longest path of the circuit. The input variables are at depth 0; and for an SLP, depth can be computed as the maximum of depths for the variables in RHS plus 1.

For example, our b_2 implementation of Figure 2 of `AES MixColumn` has the depth of 4. The variables at equal depth are shown column-wise from left to right, the left-most column has depth of 0. The variables $y_{25}, y_1, y_{15}, y_{31}, y_{30}, y_{14}$ (see Section 5 for the interpretation/details) are at depth 4, making the entire implementation of depth 4.

2.4 Logic Libraries

A total of five logic libraries are used in this paper for benchmarking the implementations. Shown in Table 2, the libraries contain XOR2 and XOR3 gates.

The gate count library simply counts the number of gates. The first two ASIC libraries are adopted from [9]. The third ASIC library is the same as the one used in [7]. The fourth ASIC library is the 130nm process from STMicroelectronics, HCMOS9GP.

Table 2: Logic libraries with gates and corresponding cost

Gate \ Library	Gate Count (GC)	STM 90nm (ASIC1)	STM 65nm (ASIC2)	TSMC 65nm (ASIC3)	STM 130nm (ASIC4)
XOR2	1	2.00 GE	1.981 GE	2.50 GE	3.33 GE
XOR3	1	3.25 GE	3.715 GE	4.20 GE	4.66 GE

3 Boyar-Peralta's Algorithm and Its Variants

Before proceeding further, we describe the basic work-flow of the Boyar-Peralta's algorithm [12] (Section 3.1). Over the years, multiple variants of this algorithm are proposed, a summary of which is given thereafter (Section 3.2).

3.1 Basic Work-flow of Boyar-Peralta's Algorithm

The original Boyar-Peralta's algorithm [12] attempts to implement b_1 with the cost vector $[0, 1]$ for the binary matrix $M^{m \times n}$. The algorithm works as follows. Initially, two vectors called the $Base$ vector of size n and $Dist$ vector of size m are created. The $Dist$ vector is initially assigned one less than the Hamming weight of each row and the $Base$ vector contains all the input variables, i.e, $x_1, x_2, \ldots, x_n$. At any given point, the $Dist$ vector for a given row represents the number of elements from the $Base$ vector that need to be combined to generate the implementation of that particular row and the $Base$ vector contains the implementations that have been generated so far. The following steps are then performed until the sum of all elements of the $Dist$ vector are 0.

1. Generate all $\binom{n}{2}$ combinations of the $Base$ vector elements and compute their sum. Create a copy of the $Dist$ vector for each combination. This will be called $DistC$ for each combination.
2. For each combination, determine whether it is possible to reduce $DistC[i]$ by 1, where $i \in [1, m]$. To put it explicitly, determine whether it is possible to implement the sum of the i^{th} row of M and the combination using $DistC[i] - 1$ elements of the $Base$ vector. If it is possible to do so, set $DistC[i]$ to $DistC[i] - 1$. If it is not possible, leave $DistC[i]$ as is.
3. Determine the most suitable combination (based on a defined heuristic) to be added to the $Base$ vector. Set the $Dist$ vector with the $DistC$ vector of the selected combination.
4. If any element $Dist[i] = 1$, this means that the i^{th} row of M can be implemented by adding two elements of the $Base$ vector. Check every pair of elements in the $Base$ vector to determine which pair when summed will be equal to the i^{th} row of M. Once this pair has been found, set $Dist[i]$ to 0.
5. Repeat until $Dist[i] = 0$ for all i.

The steps of Boyar-Peralta's algorithm can be illustrated as given in the flowchart of Figure 1.

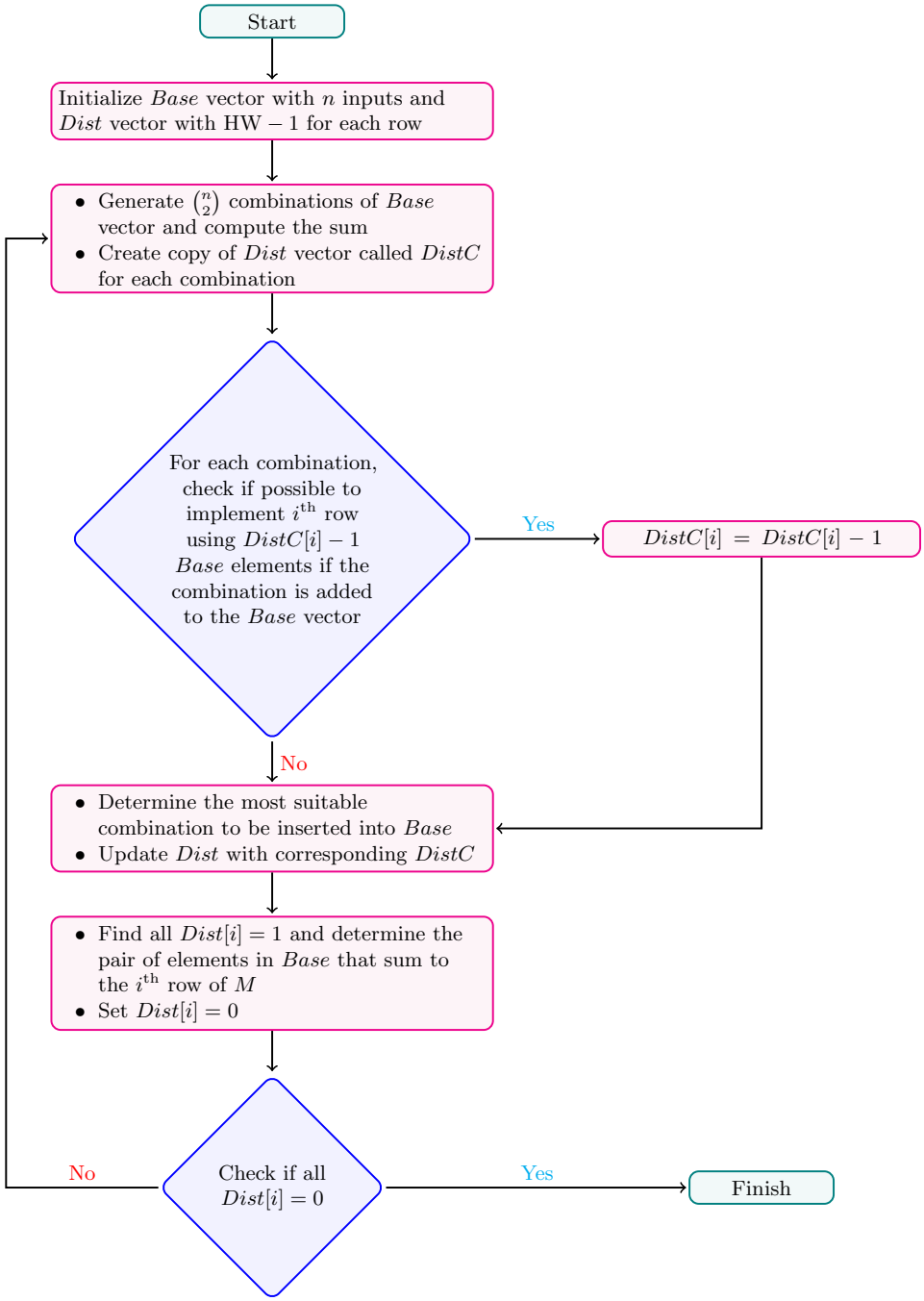

Fig. 1: Basic work-flow of Boyar-Peralta's algorithm

Remark 1. $Dist[i] + 1$ at each step of the Boyar-Peralta's algorithm contains the number of elements of the $Base$ vector which need to be summed to equal to the i^{th} row of M.

Example 2 (Working example of Boyar-Peralta's algorithm). Consider the same 5×5 binary matrix from Example 1. The step-by-step execution of the Boyar-Peralta's algorithm with the RNBP [23] heuristic used in step 3 of the algorithm is as follows. The most suitable candidate to be added to the $Base$ vector is chosen based on the L1 and squared L2 norm of the corresponding

DistC vector. Specifically, the L1 norm is minimized and ties are broken by maximizing the squared L2 norm. If multiple candidates pass the initial filter, one of them is randomly chosen.

1. Initial situation:
   - $Base = [x_0, x_1, x_2, x_3, x_4]$
   - $Dist = [0, 0, 2, 2, 2]$
   - SLP:
```
y0 = x0
y1 = x1
```
2. Iteration 1:
   - Pick all $\binom{n}{2}$ combinations of $Base$ vector and compute the corresponding $DistC$ vectors.
   - Ideal candidate: $t_0 = x_0 \oplus x_1$
   - $Base = [x_0, x_1, x_2, x_3, x_4, x_0 \oplus x_1]$
   - $Dist = [0, 0, 1, 1, 1]$
   - SLP:
```
y0 = x0
y1 = x1
t0 = x0 + x1
```
3. Iteration 2:
   - Since $Dist[2] = 1$, y_2 is implemented
   - $Base = [x_0, x_1, x_2, x_3, x_4, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2]$
   - $Dist = [0, 0, 0, 1, 1]$
   - SLP:
```
y0 = x0
y1 = x1
t0 = x0 + x1
y2 = x2 + t0
```
4. Iteration 3:
   - Since $Dist[3] = 1$, y_3 is implemented
   - $Base = [x_0, x_1, x_2, x_3, x_4, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, x_0 \oplus x_1 \oplus x_3]$
   - $Dist = [0, 0, 0, 0, 1]$
   - SLP:
```
y0 = x0
y1 = x1
t0 = x0 + x1
y2 = x2 + t0
y3 = x3 + t0
```
5. Iteration 4:
   - Since $Dist[4] = 1$, y_4 is implemented
   - $Base = [x_0, x_1, x_2, x_3, x_4, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, x_0 \oplus x_1 \oplus x_3, x_0 \oplus x_1 \oplus x_4]$
   - $Dist = [0, 0, 0, 0, 0]$, so the algorithm terminates after this step
   - SLP:
```
y0 = x0
y1 = x1
t0 = x0 + x1
y2 = x2 + t0
y3 = x3 + t0
y4 = x4 + t0
```

Table 3: Exemplary execution of Boyar-Peralta's algorithm

Candidate *Base* Elements	*DistC*	$\|DistC\|_1$	$\|DistC\|_2^2$
$t_0 = x_0 \oplus x_1$	$[0, 0, 1, 1, 1]$	3	3
$t_1 = x_0 \oplus x_2$	$[0, 0, 1, 2, 2]$	5	9
$t_2 = x_0 \oplus x_3$	$[0, 0, 2, 1, 2]$	5	9
$t_3 = x_0 \oplus x_4$	$[0, 0, 2, 2, 1]$	5	9
$t_4 = x_1 \oplus x_2$	$[0, 0, 1, 2, 2]$	5	9
$t_5 = x_1 \oplus x_3$	$[0, 0, 2, 1, 2]$	5	9
$t_6 = x_1 \oplus x_4$	$[0, 0, 2, 2, 1]$	5	9
$t_7 = x_2 \oplus x_3$	$[0, 0, 2, 2, 2]$	6	12
$t_8 = x_2 \oplus x_4$	$[0, 0, 2, 2, 2]$	6	12
$t_9 = x_3 \oplus x_4$	$[0, 0, 2, 2, 2]$	6	12

Choice of Heuristic. In Step 3 of Boyar-Peralta's algorithm, a heuristic is to be chosen to break the tie among multiple candidates which give the equal cost reduction. In the original Boyar-Peralta's algorithm [12], maximisation of Euclidean norm on *Dist* vector is taken (we skip the justification given in [12] for brevity). Therefore, for multiple *Dist* vectors which equal sum, the algorithm picks that one which maximises the Euclidean norm.

This maximisation heuristic is followed as-is in [9, 23]. However, it is argued in [20] that the minimisation of the same will work just fine. Following this, we run both the maximisation and minimisation variants in our algorithm independently of one another.

Role of `reachable()` Function. The existing literature [9, 12, 23] tend to overly explain the initial part of the Boyar-Peralta's algorithm (upto the maximisation of Euclidean norm), whereas the later part that contains the `reachable()` function seems to be overlooked in the textual description. The concept of `reachable()` is arguably more difficult. So, we present a short description here for the sake of completeness and better understanding of the algorithm.

The `reachable()` function of the Boyar-Peralta's algorithm determines whether its possible to implement the i^{th} row of a binary matrix with a new pair of *Base* vector elements using $Dist[i] - 1$ XOR2 operations. For example, consider the following from the first row of the `AES MixColumn` matrix (following the encoding used in, e.g., [9]): $y_0 \leftarrow x_7 \oplus x_8 \oplus x_{15} \oplus x_{16} \oplus x_{24}$ The initial *Base* vector contains 32 elements, i.e, $x_0, \ldots, x_{31}$; and the distance of the first row is 4 since it can be trivially implemented using 4 XOR2 gates. Given a new pair $t_0 \leftarrow x_7 \oplus x_8$ of the *Base* vector, the `reachable()` function returns true since the first row can be implemented with three XOR2 gates: $y_0 \leftarrow t_0 \oplus x_{15} \oplus x_{16} \oplus x_{24}$. However, the `reachable()` returns false for the pair $t_1 \leftarrow x_1 \oplus x_3$ since the number of XOR2 gates required to implement the row does not reduce if this pair is chosen. Once the new distance corresponding to all possible $\binom{n}{2}$ pairs of the initial *Base* vector is computed, the optimal candidates that are to be added to the augmented *Base* vector is determined using the chosen heuristic (like RNBP [23]).

3.2 Variants of Boyar-Peralta's Algorithm

In the last couple of years, three variants of the original Boyar-Peralta's algorithm are proposed in the literature [9, 20, 23]. The authors of [9] use random row and column permutations of the target matrix before feeding to the original Boyar-Peralta's algorithm. They report a b_1 cost of 95. In another direction, the authors of [23] proposed three types of randomisation, all are

internal to the algorithm and the least b_1 cost reported is 94[3]. The best implementation using the Boyar-Peralta's algorithm family is reported in [20] with a b_1 cost of 92.

4 XOR3 (b_2) Support for Boyar-Peralta's Algorithm

To the best of our knowledge, the only attempt on b_2 is reported in [9], where the authors use a post processing on the output of the algorithm that returns XOR2 implementation. This algorithm (see Section 4.1 for a concise description), with some amendments is implemented in the overall open-source package we deliver.

4.1 Modelling for XOR3 (Adopted)

The basic idea of XOR3 support in the IWSEC'19 paper by Banik, Funabiki, Isobe [9] can be thought of as a post-processing to the original Boyar-Peralta's algorithm. It can be concisely described as follows.

Start with the sequence of SLPs returned by the original Boyar-Peralta's algorithm (the matrix is fed to the algorithm after the rows and columns are given random permutations). Then look for an instance where a t variable has the fan-out of 1. For example, assume a snippet of a sequence of SLPs looks like this:

```
t4 = x0 + x6 // t4 has fan-out of 1
t20 = x1 + t4 // t20 is the only variable that uses t4
```

Then, it can be rewritten by introducing an XOR3 operation as:

```
// t4 = x0 + x6 (omitted)
t20 = x1 + x0 + x6 // t4 is substituted, t20 uses an XOR3 operation
```

Therefore, one SLP is omitted where the LHS variable has fan-out of 1. The other variable which uses the variable is now substituted with the RHS of the omitted SLP, thereby introducing an XOR3 operation. In this way, multiple (if not all) such variables with 1 fan-out can be substituted. As it can be seen, it does not take into account the cost vector for XOR2, XOR3. Thus, it is going to give the same implementation no matter the costs for XOR2 and XOR3.

One thing to note, it may not be always possible to substitute SLPs just like that. For instance, consider the following snippet:

```
t84 = t0 + t13 // t84 has fan-out of 1
t85 = x13 + t10 // t85 has fan-out of 1
y7 = t84 + t85 // Either t84 or t85 (but not both) can be substituted
```

Here, either of the t variables can be substituted, but not both (substituting both would lead to an XOR4 operation). Either of the implementation will result in equal cost, but the depth may vary. In our source-code that implements this algorithm, we explore both cases and pick the one with the least depth (tie is broken arbitrarily).

[3] It may be stated that, we are unable to reduce the cost of `AES MixColumn` from 95 XOR2 gates by using the source-code for all the three variants (RNBP, A1, A2) presented in [23], despite dedicated efforts. Apparently, the case of 94 XOR2 gates reported in [23] happens with a low probability.

4.2 Modelling for XOR3 (New)

Suppose a cost vector with costs for XOR2 and XOR3 is given. Initialization of the *Base* vector and *Dist* vector is identical to the original algorithm (described in Section 3.1). The following steps highlight the changes made:

1. Generate all $\binom{n}{2}$ pairs and $\binom{n}{3}$ triplets of the *Base* vector elements and compute the XOR. The pairs represent XOR2 combinations and the triplets represent XOR3 combinations. For each combination, assign the corresponding cost from the cost vector.
2. For each of the XOR2 combinations, determine whether it is possible to reduce $DistC[i]$ by 1. Similarly, for each of the XOR3 combinations, first check it is possible to reduce $DistC[i]$ by 2; if it is not, then check if $DistC[i]$ can be reduced by 1. If $DistC[i]$ cannot be changed, leave it as-is.
3. Based on the defined heuristic, determine the most suitable combination to be included to the *Base* vector. Unlike the original Boyar-Peralta's algorithm, the heuristic would account for the distance vector, $DistC$; and the cost of the combination.
4. If $Dist[i] = 1$ or $Dist[i] = 2$, then the i^{th} row of M can be implemented by adding two or three elements of the *Base* vector respectively, i.e., either XOR2 or XOR3 operations. Check every pair/triplet of the *Base* vector to determine the elements which sum to $M[i]$. Once these elements have been found, set $Dist[i]$ to 0, and include $M[i]$ to the *Base* vector.
5. Repeat until $Dist[i] = 0$ for all i.

Remark 2. The Boyar-Peralta's algorithm assumes that the cost of XOR2 gates is 1. Therefore, to incorporate XOR3 support, XOR3 cost needs to be taken relative to XOR2 (so that the cost for XOR2 remains at 1).

Remark 3. Because of the way the `EasyMove()` function in Boyar-Peralta's algorithm is implemented, an additional hard check (that allows any XOR3 gate only if its cost is less than or equal to 2× XOR2 gate) would possibly be required. Otherwise, if the Hamming weight of a row is 2, the algorithm would always pick an XOR3 gate even if it costs more than 2× XOR2 gate. In our current implementation, a warning is given if XOR3 is greater than 2× XOR2.

Remark 4. In a similar way, it is possible to extend the support for XOR4 (and even higher input XOR gates). This is an interesting research direction, but we skip it here for brevity. Also, support for higher input XOR gates would make the program taking considerably longer time.

As the source code for [23] is available online[4], we decide to implement our approach (which is described in Section 4.2) on top of it. We choose the RNBP variant due to its efficiency over the other variants proposed in [23].

4.3 Other Aspects Considered

In addition to the incorporation of two types of XOR3 support, the following amendments are incorporated in our source-code:

- We make continuous numbering for the temporary variables. This appears not to be the case for the previous works [9, 12, 23].

[4] At https://github.com/thomaspeyrin/XORreduce.

- Due to the way the Boyar-Peralta's algorithm is implemented, it skips the rows of the given matrix if the Hamming weight is 1 (as the XOR2 implementation is trivial in this case). While the justification is correct, it leads to a wrong SLP sequence in this case. We fix this issue.
- As already noted, we make the following enhancements:
  - Make use of the randomisation inside the algorithm (namely, RNBP) which is proposed in [23], as well as the row-column permutation of the matrix before feeding it to the algorithm which is proposed in [9].
  - Instead of only the tie-breaker based on maximisation of the Euclidean norm (which is the case for [9, 12, 23]); we also implement the same but with minimisation, following [20].
  - XOR3 support which is proposed in [9] is available as a native interface (with our implementation of the algorithm, as the source-code for [9] is not publicly available). Aside from that, we implement the minimisation tie-breaker. Also we check for depth for all possible implementations with the same cost (see Section 4.1 for more details).
- We provide an easy-to-use Python interface to generate SLPs and the entire package is available as an open-source project. The user is notified when a least cost is obtained and all the relevant results are stored in separate files. The maximum and minimum tie-breakers are internally supported with two threads in the program.

5 Results

In this part, we present a summary of the experimental results. Here x's and y's are the Boolean variables which respectively indicate the input variables and the output variables. An arbitrary number of temporary variables, t's, are created on-the-fly (those are required since this is not an in-place algorithm) to produce the SLP. So, the RHS can contain all variables while the LHS can only have the t and y variables. Given the binary matrix $M^{m\times n}$, $X^{n\times 1}$ = vector of x's, $Y^{m\times 1}$ = vector of y's, then it holds that $Y = MX$. In all the examples that follow, we only take square matrices, i.e., $m = n$. Least costs with respect to the five libraries (described in Section 2.4) and depth (described in Section 2.3) are given hereafter.

5.1 16×16 Matrices

In general, the execution for the binary 16×16 matrices is quite fast in our implementation. Table 4 shows consolidated results for few well-known ciphers. All the results reported can be obtained by our implementation, though some are already reported in [9]. The highlighted entries mark the least cost in the respective category, which are reported for the first time in the literature.

Table 4: Implementations of few 16×16 matrices in b_2

Matrix	GC	ASIC1	ASIC2	ASIC3	ASIC4
`JOLTIK-BC` [15]	28 (6, 22)	83.0 (9, 20)	91.14 (16, 16)	106.5 (9, 20)	122.50 (6, 22)
`MIDORI` [8]	16 (0, 16)	45.0 (16, 4)	46.56 (16, 4)	56.8 (16, 4)	71.92 (16, 4)
`PRINCE` M_0, M_1 [11]	16 (0, 16)	45.0 (16, 4)	46.56 (16, 4)	56.8 (16, 4)	71.92 (16, 4)
`PRIDE` $L_0 - L_3$ [1]	16 (0, 16)	45.0 (16, 4)	46.56 (16, 4)	56.8 (16, 4)	71.92 (16, 4)
`QARMA-64` [2]	16 (0, 16)	45.0 (16, 4)	46.56 (16, 4)	56.8 (16, 4)	71.92 (16, 4)
`SMALLSCALE-AES` [13]	24 (0, 24)	78.0 (0, 24)	85.93 (19, 13)	100.8 (0, 24)	111.84 (0, 24)

Number of (XOR2, XOR3) gates are given within parenthesis

The algorithm for XOR3 support in [9] (Section 4.1) only allows for a fixed implementation disregarding the costs for XOR2 and XOR3 gates. In contrast, the new idea we present here takes

into account the relative costs and returns various implementations. One such example is given, which is the case of the JOLTIK-BC linear layer in Codes 1.1 and 1.2 each as a sequence of SLPs (both are indicated in Table 4). The former implementation has the least cost so far for ASIC1 (83.0 GE, with 9 XOR2 and 20 XOR3), and has the depth of 5; the latter gives the least cost so far for GC (28) and ASIC4 (122.50 GE, with 6 XOR2 and 22 XOR3), and has the depth of 7.

Code 1.1: JOLTIK-BC linear layer in b_2 (83.0 GE in ASIC1) in SLP format

```
t0 = x8 + x13 + x12
y3 = x5 + x3 + t0
t2 = x0 + x4 + x1
y15 = x9 + x15 + t2
t4 = x9 + t0
t5 = x14 + x2 + t0
y5 = x11 + y3 + t5
t7 = x5 + t2
t8 = x2 + x10 + x12
y12 = t7 + t8
t10 = x4 + x2 + t4
y4 = x10 + t10
t12 = x7 + t4
y7 = x13 + x1 + t12
y9 = x15 + t5 + t12
t15 = x6 + x13 + x10
y13 = x3 + x11 + t15
y8 = t5 + y12 + t15
t18 = x14 + x0 + t4
y0 = x6 + t18
t20 = x11 + t12
y2 = x12 + t10 + t20
y11 = t7 + y7 + t20
y14 = x8 + t18 + t20
t24 = y3 + y12 + t10
y10 = y15 + t24
t26 = x15 + t4 + t15
y6 = x1 + t24 + t26
y1 = y7 + t26
```

Code 1.2: JOLTIK-BC linear layer in b_2 (28 GC, 122.50 GE in ASIC4) in SLP format

```
t0 = x5 + x4 + x0
y11 = x13 + x11 + t0
t2 = x8 + x9 + x12
y7 = x1 + x7 + t2
t4 = x1 + t0
t5 = x13 + t2
t6 = x10 + x6 + t0
y13 = x3 + y11 + t6
t8 = x5 + x9
y15 = x15 + t4 + t8
y3 = x3 + t5 + t8
t11 = x14 + x9 + x2
y9 = x15 + x7 + t11
t13 = x14 + x6
y0 = x0 + t5 + t13
y8 = x8 + t4 + t13
t16 = x14 + y0 + y8
y14 = y11 + y7 + t16
t18 = t0 + t2 + t11
y12 = t6 + y8 + t18
y5 = y11 + y3 + t18
y2 = x5 + y14 + t18
t22 = y15 + t13 + t16
y6 = y3 + t22
t24 = t6 + t8
y1 = x7 + y15 + t24
y10 = t2 + y6 + t24
y4 = t11 + y0 + t24
```

5.2 32×32 Matrices

Similar to the 16×16 matrices, we now show the summarised results for the 32×32 binary matrices in Table 5, while the results reported for the first time are highlighted (the rest are the state-of-the-art results and reported in [9]). It is to be noted that, we achieve improvement for GC for all the matrices we experiment with.

In terms of the AES MixColumn matrix (with encoding compatible to that of [23]), our best result is of 59 GC (12 XOR2 and 47 XOR3 gates), depth 4; which is given in Code 1.3. This improves from the 67 GC, depth 6 implementation of [9] (all the implementations obtained have a depth of 6). Note that the same implementation also gives the least known cost for ASIC4 (258.98 GE, which is an improvement from [9]). A graphical representation for this implementation grouping variables at same depth in the same column is given in Figure 2. Further, an improved

Table 5: Implementations of few 32×32 matrices in b_2

Matrix	GC	ASIC1	ASIC2	ASIC3	ASIC4
`AES`	59 (12, 47)	169.0 (39, 28)	181.28 (39, 28)	215.1 (39, 28)	258.98 (12, 47)
`ANUBIS` [10]	62 (11, 51)	185.0 (60, 20)	193.16 (60, 20)	234.0 (60, 20)	274.29 (11, 51)
`CLEFIA` M_0 [22]	62 (13, 49)	185.0 (60, 20)	193.16 (60, 20)	234.0 (60, 20)	271.63 (13, 49)
`CLEFIA` M_1 [22]	65 (3, 62)	193.0 (38, 36)	209.00 (38, 36)	246.2 (38, 36)	294.30 (38, 36)
`TWOFISH` [21]	73 (17, 56)	215.5 (20, 54)	240.23 (20, 54)	276.8 (20, 54)	317.57 (17, 56)

Number of (XOR2, XOR3) gates are given within parenthesis

implementation of `TWOFISH` [21] linear layer is given in Code 1.4; which incurs a GC cost of 73, ASIC4 cost of 317.57 GE, and depth of 9.

Code 1.3: `AES MixColumn` in b_2 with 59 GC/depth 4 in SLP format

```
t0 = x8 + x16
t1 = x24 + x0
t2 = x28 + x24 + x16
t3 = x12 + x8 + x0
t4 = x22 + x14 + x7
t5 = x23 + x6 + x30
t6 = x20 + x27 + x3
y19 = x11 + t2 + t6
t8 = x11 + x4 + x19
y3 = x27 + t3 + t8
t10 = x13 + x20 + x28
y4 = x5 + t3 + t10
t12 = x12 + x4 + x29
y20 = x21 + t2 + t12
t14 = x18 + x26 + x11
y2 = x10 + x3 + t14
y10 = x2 + x19 + t14
t17 = x18 + x10 + x27
y18 = x11 + y10 + t17
y26 = x3 + x2 + t17
t20 = x22 + x5 + x29
y21 = x13 + x30 + t20
y13 = x14 + x21 + t20
t23 = x18 + x1 + x25
y9 = x10 + x17 + t23
y17 = x9 + x26 + t23
t26 = x6 + x21 + x29
y29 = x22 + y21 + t26
y5 = x13 + x14 + t26
t29 = x24 + x16 + x31
y23 = x7 + x15 + t29
y22 = x14 + t5 + t29
t32 = x2 + y9 + y17
y25 = x10 + x1 + t32
y1 = x26 + x25 + t32
t35 = x8 + x15 + x0
y6 = x30 + t4 + t35
y7 = x23 + x31 + t35
t38 = x23 + t1 + y23
y15 = t35 + t38
y31 = t29 + t38
t41 = x17 + x16 + t1
y16 = x25 + t0 + t41
y8 = x9 + t41
t44 = x1 + x0 + t0
y0 = x9 + t1 + t44
y24 = x25 + t44
t47 = t1 + y6 + y7
y30 = t5 + t47
t49 = x4 + x21 + t10
y12 = t0 + t49
t51 = x19 + t0 + t6
y11 = x12 + t51
t53 = x20 + x5 + t1
y28 = t12 + t53
t55 = x28 + x3 + t1
y27 = t8 + t55
t57 = t0 + y23 + y22
y14 = t4 + t57
```

6 Conclusion

With the renewed interest in the Boyar-Peralta's algorithm [12] in the last couple of years, our work combines existing ideas about the algorithm atop our own idea of incorporating XOR3 support in it. We take an open-source implementation of the algorithm (provided by the authors of [23]), make several changes to reflect the state-of-the-art observations [9,20], and finally deliver a complete package as an easy-to-use and versatile open-source project (that contains our algorithm for XOR3 support).

Our work achieves the best known results in terms of a logic library comprising of {XOR2, XOR3} gates, several of which are reported for the first time (the rest results that are reported here are tied with [9]). For instance, we present an implementation of the `AES MixColumn` matrix with 59 gate count/4 depth/258.98 GE in STM 130nm process (ASIC4). We are optimistic, these results can be improved further with more runs of our implementation.

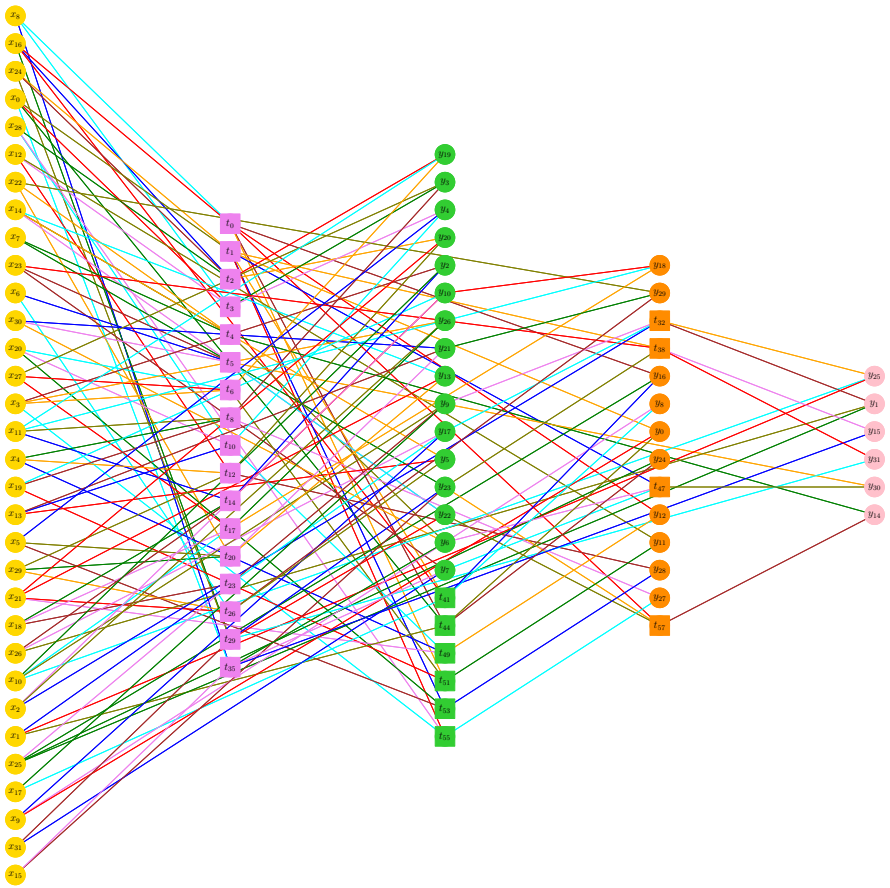

Fig. 2: `AES` linear layer (`MixColumn`) in b_2 with 59 GC/depth 4 in graphical form

In the future scope, a number of works can be undertaken. First, we may consider the XNOR gates in the library. Higher input XOR gates (XOR4 and beyond) can be incorporated. It may be of interest to optimise the depth for the implementation, as far we know there is no dedicated work in the literature for studying this metric. The cost for inverse of the matrices for the given libraries may be an interesting direction to study as well. One may also be interested in finding a reversible implementation together with XOR3 support.

References

1. Albrecht, M.R., Driessen, B., Kavun, E.B., Leander, G., Paar, C., Yalçin, T.: Block ciphers - focus on the linear layer (feat. PRIDE). In: Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I. pp. 57–76 (2014), https://doi.org/10.1007/978-3-662-44371-2_4 12
2. Avanzi, R.: The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. IACR Trans. Symmetric Cryptol. 2017(1), 4–44 (2017), https://doi.org/10.13154/tosc.v2017.i1.4-44 12
3. Baksi, A., Jang, K., Song, G., Seo, H., Xiang, Z.: Quantum implementation and resource estimates for rectangle and knot. Cryptology ePrint Archive, Report 2021/982 (2021), https://ia.cr/2021/982 4
4. Baksi, A., Karmakar, B., Dasu, V.A.: Poster: Optimizing device implementation of linear layers with automated tools. In: Zhou, J., Ahmed, C.M., Batina, L., Chattopadhyay, S., Gadyatskaya, O., Jin, C., Lin, J., Losiouk, E.,

Code 1.4: `TWOFISH` linear layer in b_2 (73 GC, 317.57 GE in ASIC4) in SLP format

```
t0 = x8 + x9 + x17
t1 = x0 + x25 + x1
t2 = x24 + x8 + x16
t3 = x19 + x27 + t2
t4 = x31 + x23 + x15
t5 = x26 + x2 + t1
y16 = x10 + t2 + t5
t7 = x12 + x20 + x0
t8 = x18 + x17 + y16
y24 = x25 + x10 + t8
t10 = x30 + x14 + t2
t11 = x13 + x29 + x21
t12 = x22 + x6 + t10
y6 = x15 + t12
t14 = x28 + x4 + x16
t15 = x16 + x7 + t4
y31 = x17 + t1 + t15
y15 = x1 + t0 + t15
y23 = t0 + t2 + y31
t19 = x9 + x1 + t8
y8 = x26 + x25 + t19
t21 = x11 + t3 + t11
y3 = x12 + x3 + t21
t23 = x24 + t7 + t14
t24 = x5 + y3 + t23
t25 = x9 + t12 + t23
t26 = x0 + x23 + y6
y22 = t15 + t26
y14 = x24 + t26
t29 = x5 + t11
t30 = x27 + t8 + t14
t31 = x25 + t0 + t4
y5 = x14 + t29 + t31
y7 = x7 + t31
t34 = x11 + t19 + t23
y2 = x4 + t34
t36 = x5 + t1 + t25
y20 = x22 + x29 + t36
t38 = x13 + y15 + t25
y4 = x6 + y23 + t38
t40 = x3 + t0 + t5
t41 = x27 + x11 + t40
y17 = x24 + t41
t43 = x26 + x18 + t3
y25 = t40 + t43
t45 = x22 + x31 + y5
y13 = y15 + t31 + t45
t47 = x3 + t30
y26 = x20 + t3 + t47
y18 = x12 + t0 + t47
t50 = y6 + y31
y29 = t10 + t29 + t50
y21 = x25 + t45 + t50
t53 = t5 + y8
y0 = x17 + t1 + t53
t55 = x24 + x21
y19 = x20 + t24 + t55
t57 = x11 + t43 + t53
y1 = t2 + t57
t59 = x13 + t2 + t24
y27 = x28 + t59
t61 = x8 + x15 + t26
y30 = x7 + t61
t63 = x29 + t14 + t24
y11 = x12 + t63
t65 = t0 + t36 + t55
y28 = x14 + t65
t67 = x12 + y16 + t34
y9 = y24 + y26 + t67
t69 = x19 + t1 + y2
y10 = x0 + t14 + t69
t71 = x30 + x21 + t38
y12 = x9 + t15 + t71
```


Luo, B., Majumdar, S., Maniatakos, M., Mashima, D., Meng, W., Picek, S., Shimaoka, M., Su, C., Wang, C. (eds.) Applied Cryptography and Network Security Workshops. pp. 500–504. Springer International Publishing, Cham (2021) 1

5. Baksi, A., Karmakar, B., Dasu, V.A., Saha, D., Chattopadhyay, A.: Further insights on implementation of the linear layer. In: SILC Workshop-Security and Implementation of Lightweight Cryptography (2021) 1
6. Baksi, A., Karmakar, B., Dasu, V.A., Saha, D., Chattopadhyay, A.: Further insights on implementation of the linear layer. SILC Workshop – Security and Implementation of Lightweight Cryptography (2021), https://www.esat.kuleuven.be/cosic/events/silc2020/wp-content/uploads/sites/4/2020/10/Submission1.pdf 4
7. Baksi, A., Pudi, V., Mandal, S., Chattopadhyay, A.: Lightweight ASIC implementation of AEGIS-128 pp. 251–256 (2018), https://doi.org/10.1109/ISVLSI.2018.00054 6
8. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A block cipher for low energy (extended version). Cryptology ePrint Archive, Report 2015/1142 (2015), https://eprint.iacr.org/2015/1142 12
9. Banik, S., Funabiki, Y., Isobe, T.: More results on shortest linear programs. Cryptology ePrint Archive, Report 2019/856 (2019), https://eprint.iacr.org/2019/856 1, 2, 5, 6, 9, 10, 11, 12, 13, 14
10. Barreto, P.S.L.M., Rijmen, V.: The anubis block cipher (2000), Submission to NESSIE project. Available at https://www.cosic.esat.kuleuven.be/nessie/workshop/submissions/anubis.zip 14
11. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In: Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. pp. 208–225 (2012), https://doi.org/10.1007/978-3-642-34961-4_14 12
12. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In: Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings. pp. 178–189 (2010), https://doi.org/10.1007/978-3-642-13193-6_16 1, 2, 5, 6, 9, 11, 12, 14
13. Cid, C., Murphy, S., Robshaw, M.J.B.: Small scale variants of the AES. In: Gilbert, H., Handschuh, H. (eds.) Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23,

2005, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3557, pp. 145–162. Springer (2005), https://doi.org/10.1007/11502760_10 12
14. Dasu, V.A., Baksi, A., Sarkar, S., Chattopadhyay, A.: LIGHTER-R: optimized reversible circuit implementation for sboxes. In: 32nd IEEE International System-on-Chip Conference, SOCC 2019, Singapore, September 3-6, 2019. pp. 260–265 (2019), https://doi.org/10.1109/SOCC46988.2019.1570548320 4
15. Jean, J., Nikolić, I., Peyrin, T.: Joltik v1.3 (2015), Submission to the CAESAR competition, http://www1.spms.ntu.edu.sg/~syllab/Joltik 3, 12
16. Jean, J., Peyrin, T., Sim, S.M., Tourteaux, J.: Optimizing implementations of lightweight building blocks. IACR Trans. Symmetric Cryptol. 2017(4), 130–168 (2017), https://doi.org/10.13154/tosc.v2017.i4.130-168 3
17. Khoo, K., Peyrin, T., Poschmann, A.Y., Yap, H.: FOAM: searching for hardware-optimal SPN structures and components with a fair comparison. In: Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings. pp. 433–450 (2014), https://doi.org/10.1007/978-3-662-44709-3_24 3
18. Kölsch, L.: Xor-counts and lightweight multiplication with fixed elements in binary finite fields. In: Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I. pp. 285–312 (2019), https://doi.org/10.1007/978-3-030-17653-2_10 3
19. Lin, D., Xiang, Z., Zeng, X., Zhang, S.: A framework to optimize implementations of matrices. In: Paterson, K.G. (ed.) Topics in Cryptology - CT-RSA 2021 - Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17-20, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12704, pp. 609–632. Springer (2021), https://doi.org/10.1007/978-3-030-75539-3_25 2, 4, 5
20. Maximov, A.: Aes mixcolumn with 92 xor gates. Cryptology ePrint Archive, Report 2019/833 (2019), https://eprint.iacr.org/2019/833 2, 4, 5, 9, 10, 12, 14
21. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N.: Twofish: A 128-bit block cipher (1998), https://www.schneier.com/academic/paperfiles/paper-twofish-paper.pdf 3, 14
22. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., Iwata, T.: The 128-bit blockcipher clefia (extended abstract). In: Biryukov, A. (ed.) Fast Software Encryption. pp. 181–195. Springer Berlin Heidelberg, Berlin, Heidelberg (2007) 14
23. Tan, Q.Q., Peyrin, T.: Improved heuristics for short linear programs. Cryptology ePrint Archive, Report 2019/847 (2019), https://eprint.iacr.org/2019/847 2, 4, 5, 7, 9, 10, 11, 12, 13, 14
24. Xiang, Z., Zeng, X., Lin, D., Bao, Z., Zhang, S.: Optimizing implementations of linear layers. Cryptology ePrint Archive, Report 2020/903 (2020), https://eprint.iacr.org/2020/903 2, 3, 4, 5