IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2022, No. 2, pp. 351–378. DOI:10.46586/tosc.v2022.i2.351-378

More Inputs Makes Difference: Implementations of Linear Layers Using Gates with More Than Two Inputs

Qun Liu[1,2], Weijia Wang[1,2], Ling Sun[1,2], Yanhong Fan[1,2], Lixuan Wu[1,2] and Meiqin Wang[1,2,3(✉)]

[1] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Jinan, China
[2] School of Cyber Science and Technology, Shandong University, Qingdao, China
[3] Quan Cheng Shandong Laboratory, Jinan, China
{qunliu,fanyh,lixuanwu}@mail.sdu.edu.cn, {wjwang,lingsun,mqwang}@sdu.edu.cn

Abstract. Lightweight cryptography ensures cryptography applications to devices with limited resources. Low-area implementations of linear layers usually play an essential role in lightweight cryptography. The previous works have provided plenty of methods to generate low-area implementations using 2-input xor gates for various linear layers. However, it is still challenging to search for smaller implementations using two or more inputs xor gates. This paper, inspired by Banik *et al.*, proposes a novel approach to construct a quantity of lower area implementations with $(n+1)$-input gates based on the given implementations with n-input gates. Based on the novel algorithm, we present the corresponding search algorithms for $n = 2$ and $n = 3$, which means that we can efficiently convert an implementation with 2-input xor gates and 3-input xor gates to lower-area implementations with 3-input xor gates and 4-input xor gates, respectively.

We improve the previous implementations of linear layers for many block ciphers according to the area with these search algorithms. For example, we achieve a better implementation with 4-input xor gates for AES MixColumns, which only requires 243 GE in the STM 130 nm library, while the previous public result is 258.9 GE. Besides, we obtain better implementations for all 5500 lightweight matrices proposed by Li *et al.* at FSE 2019, and the area for them is decreased by about 21% on average.

Keywords: Lightweight Cryptography · Linear Layers · Low Area · AES

1 Introduction

In recent years, lightweight cryptography has been a significant trend in many fields, such as the Internet of Things (IoTs) and Radio-Frequency IDentification tags (RFID). Lightweight cryptography means a low-cost implementation, where the cost covers the circuit area, latency, power consumption, and so on. It extends cryptography applications to devices with limited resources. Security has been a core area of concern for researchers, as various limitations have led to new security threats.

Generally speaking, research on lightweight cryptography falls in two directions. The first direction focuses on designing new ciphers that are assumed to be efficient implementations, such as PRESENT [BKL+07], LED [GPP11], MIDORI [BBI+15], and SAND [CFS+22]. The second direction tries to optimize the implementations of given ciphers, which has also drawn much attention. For example, the Advanced Encryption Standard (AES) [DR20] has been widely used in practice. Its round function has been

Received: 2022-03-01 Accepted: 2022-05-01 Published: 2022-06-10

frequently used in designing other cryptographic primitives (e.g., AEGIS [WP13] and Rocca [SLN+21]). Therefore, an efficient implementation will directly reduce the cost of deploying AES for primitives using its round function.

In practice, we can build circuit implementations of linear layers using a variety of heuristics (e.g., [Paa97, BP10, BMP13, KLSW17, LSL+19, TP20, XZL+20, BFI21, LXZZ21, LWF+22] for an incomplete list) originating in Paar's work [Paa97]. Most of those previous works only consider the circuits using 2-input gates. However, most standard cell libraries of CMOS logic processes have dedicated gates that support 3-input xor gates or 4-input xor gates ([BPMC18, RMTA20, BFI21, BDK+21]). Meanwhile, it should be noted that using gates with more than two inputs may give rise to some more efficient circuits. We first consider the most popular criteria for the lightweight implementation gate equivalents (GE).[1] In this paper, we use two ASIC libraries (see Table 1), adopted from [BDK+21].

Table 1: Logic libraries with gates and the corresponding costs.

Gate Library	XOR2	XOR3	XOR4
1: STM 90 nm	2 GE	3.25 GE	5 GE
2: STM 130 nm	3.33 GE	4.66 GE	5.99 GE

We give an example with a matrix M_1,

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix},$$

the inputs $\vec{x} = (x_0, x_1, x_2, x_3)^T$, and the outputs $\vec{y} = (y_0, y_1)^T$. The optimal solution with the minimum number of XOR operations to compute $\vec{y} = M_1\vec{x}$ can be performed by the procedure described by Figure 1-left, requiring three gates. In STM 130 nm library, the circuit needs 9.99 GE and cannot be improved by 2-input xor gates. However, if we construct the circuit with 3-input gates (see Figure 1-right), it only needs 9.32 GE.

$$\begin{aligned} t_4 &= x_1 \oplus x_2 \\ y_0 &= t_4 \oplus x_0 \\ y_1 &= t_4 \oplus x_3 \end{aligned} \qquad\qquad \begin{aligned} y_0 &= x_0 \oplus x_1 \oplus x_2 \\ y_1 &= x_1 \oplus x_2 \oplus x_3 \end{aligned}$$

Figure 1: Two implementations of $(y_0, y_1)^T = M_1 \times (x_0, x_1, x_2, x_3)^T$.

Despite the potential advantage of the circuit using gates with more than two inputs, it is still challenging to design an approach to find suitable circuits. Baksi *et al.* [BDK+21] directly searched for the circuits by adopting the BP algorithm proposed in [BP10] with 3-input xor gates. It is the first heuristic method that takes them into account and performs well in STM 130 nm library. Besides, this approach may significantly increase the searching space due to the larger amount of gate inputs, leading to a long time to run until a proper circuit is found. In [BFI21], Banik *et al.* proposed another strategy that attempts to convert a circuit with 2-input gates into another one with 3-input gates. This strategy can benefit from the known heuristic algorithms that produce circuits with 2-input gates. We call it the BFI algorithm and give a brief introduction. The algorithm starts with the BP algorithm to generate many circuits with 2-input gates, and then transforms them into ones with 3-input gates. This intelligent strategy reduces the search space and obtains many circuits for linear layers with fewer areas than before. Nevertheless, we note that

[1] The unit of gate size is Gate Equivalent (GE), where one GE equals the area of a 2-input NAND gate. The cost of other gates in terms of GE is a normalized ratio between their area and one NAND gate area.

firstly, it is still unknown whether it can be generalized to gates with more than 3 inputs, and secondly, the one (with 2-input gates) to one (with 3-input gates) transformation may reduce too much the search space. It is a bit of waste that a suitable circuit is only transformed into one circuit with 3-input gates. Therefore, this is an exciting research direction.

1.1 Our Contributions

In this paper, we follow the line of the work of Banik *et al.* and propose two algorithms to provide more generalized and efficient algorithms to reduce the circuit areas of linear layers. Then, we instantiate them to optimize the existing matrices with 3/4-input xor gates.

New algorithms to optimize the circuit areas with 3/4-input xor gates. This paper proposes two novel algorithms that can construct a quantity of lower area implementations with $(n+1)$-input gates based on the given implementations with n-input gates. The first method is the transform algorithm. It can convert a circuit using gates with no more than n inputs into a lower area circuit using gates with no more than $n+1$ inputs. As a transform framework, we can utilize it to convert circuits using gates with more inputs. The second method is the graph extending algorithm. It can produce massive equivalent circuits with low area and depth for a given circuit. The algorithm utilizes more information hidden in the given circuit and can be applied after any existing algorithms to optimize the results further.

Based on the novel algorithms, we instantiate the corresponding search algorithms EGT2 for $n = 2$ and EGT3 for $n = 3$, which means that we can efficiently transform an implementation with 2-input xor gates and 3-input xor gates to lower-area implementations with 3-input xor gates and 4-input xor gates, respectively.

Application to many linear layers of block ciphers. We apply EGT2 and EGT3 to several linear layers from the literature, including matrices already used in different ciphers [DR20, CMR05, JNP15, Ava17, BBI+15, BCG+12, ADK+14, Ava17, BJK+16, AIK+00]. With the help of these search algorithms, we improve the previous implementations of linear layers for many block ciphers according to the area. The results are listed in Table 2 and Table 3, where XZLBZ is a heuristic algorithm proposed in [XZL+20]. For the thirteen linear layers in the tables below, we optimize eight matrices in circuit areas and obtain four with the same circuit areas. Notably, for AES MixColumns, we achieve a circuit with 243 GE, better than the previous best result (258.9 GE) reported in [BDK+21].

We also apply our algorithms to 5500 lightweight matrices proposed by Li *et al.* in [LSL+19] and obtain better circuits for all the matrices than all known state-of-the-art results. Figure 2 shows the comparison between the GE concerning different algorithms. On average, for each matrix, the circuit area is decreased by about 21%.

Additionally, we synthesize different implementations of AES MixColumns in hardware (see Table 4). The results show that our implementation achieves a better area and reduces power at the cost of slight and reasonable growth of latency.

1.2 Organization

In Section 2, we give some basic notations and definitions. Then, we propose the transform algorithm in Section 3. In Section 4, we propose the graph extending algorithm and give some examples. Next, in Section 5, we combine two algorithms and instantiate EGT2 and EGT3 algorithms. Finally, we conclude and propose future research directions in Section 6.

Table 2: The results of implementation cost of matrices in the library named STM 90 nm. The results consist of the circuit area (GE) and the gates. We use "(m)" to represent m 2-input xor gates, use "(m, p)" to represent m 2-input xor gates and p 3-input xor gates, and use "(m, p, q)" to represent m 2-input xor gates and p 3-input xor gates and q 4-input xor gates.

Matrix	XZLBZ[a]	[BDK+21][b]	[BFI21][b]	XZLBZ+BFI[b]	**XZLBZ+EGT2**[b]	**XZLBZ+EGT3**[c]
AES [DR20]	184 (92)	176.7 (12, 47)	169.0 (39, 28)	169.7 (41, 27)	**167.5 (35, 30)**	**166.5 (31, 26, 4)**
ANUBIS [BR00]	198 (99)	187.7 (11, 51)	185.0 (60, 20)	**177.2 (35, 33)**	**177.2 (35, 33)**	**175.5 (28, 26, 7)**
CLEFIA M_0 [SSA+07]	196 (98)	185.2 (13, 49)	185.0 (60, 20)	180.7 (40, 31)	**178.5 (34, 34)**	**178.0 (32, 32, 2)**
CLEFIA M_1 [SSA+07]	206 (103)	207.5 (3, 62)	193.0 (38, 36)	190.2 (48, 29)	**186.5 (38, 34)**	**185.7 (35, 31, 3)**
FOX MU4 [JV04]	272 (136)	-	**231.7 (46, 43)**	241.7 (64, 35)	241.7 (64, 35)	239.0 (53, 24, 11)
JOLTIK [JNP15]	88 (44)	83.0 (9, 20)	84.0 (16, 16)	83.5 (19, 14)	**82.0 (15, 16)**	**81.7 (14, 15, 1)**
MIDORI [BBI+15]	48 (24)	52.0 (0, 16)	**45.0 (16, 4)**	48.0 (24, 0)	**45.0 (16, 4)**	**45.0 (16, 4, 0)**
PRINCE M_0, M_1 [BCG+12]	48 (24)	52.0 (0, 16)	**45.0 (16, 4)**	46.5 (20, 2)	**45.0 (16, 4)**	**45.0 (16, 4, 0)**
PRIDE $L_0 - L_3$ [ADK+14]	48 (24)	52.0 (0, 16)	**45.0 (16, 4)**	48.0 (24, 0)	**45.0 (16, 4)**	**45.0 (16, 4, 0)**
QARMA128 [Ava17]	96 (48)	-	90.7 (34, 7)	90.7 (34, 7)	**90.0 (32, 8)**	**90.0 (32, 8, 0)**
QARMA64 [Ava17]	48 (24)	52.0 (0, 16)	**45.0 (16, 4)**	**45.0 (16, 4)**	**45.0 (16, 4)**	**45.0 (16, 4)**
SMALLSCALE AES [CMR05]	86 (43)	**78.0 (0, 24)**	80.2 (19, 13)	80.2 (19, 13)	79.5 (17, 14)	79.5 (17, 14, 0)
TWOFISH [SKW+98]	222 (111)	215.5 (20, 54)	222.5 (43, 42)	203.0 (56, 28)	**201.0 (55, 28)**	**199.7 (50, 23, 5)**

[a] Using 2-input xor gates.
[b] Using 2/3-input xor gates.
[c] Using 2/3/4-input xor gates.

Table 3: The results of implementation cost of matrices in the library named STM 130 nm. The results consist of the circuit area (GE) and the gates. We use "(m)" to represent m 2-input xor gates, use "(m, p)" to represent m 2-input xor gates and p 3-input xor gates, and use "(m, p, q)" to represent m 2-input xor gates and p 3-input xor gates and q 4-input xor gates.

Matrix	XZLBZ[a]	[BDK+21][b]	[BFI21][b]	XZLBZ+BFI[b]	**XZLBZ+EGT2**[b]	**XZLBZ+EGT3**[c]
AES [DR20]	306.3 (92)	258.9 (12, 47)	260.3 (39, 28)	259.0 (26, 37)	**255.0 (29, 34)**	**243.0 (22, 21, 12)**
ANUBIS [BR00]	329.6 (99)	274.2 (11, 51)	293.0 (60, 20)	**270.3 (35, 33)**	**270.3 (35, 33)**	**253.6 (21, 24, 12)**
CLEFIA M_0 [SSA+07]	326.3 (98)	271.63 (13, 49)	293.0 (60, 20)	276.3 (34, 35)	**270.9 (31, 36)**	**258.9 (23, 16, 18)**
CLEFIA M_1 [SSA+07]	342.9 (103)	298.9 (3, 62)	294.3 (38, 36)	292.9 (39, 35)	**283.6 (32, 38)**	**270.2 (20, 27, 13)**
FOX MU4 [JV04]	452.8 (136)	-	**353.5 (46, 43)**	374.2 (48, 46)	372.2 (46, 47)	**347.5 (32, 26, 20)**
JOLTIK [JNP15]	146.5 (44)	**122.5 (6, 22)**	127.8 (16, 16)	126.5 (10, 20)	123.8 (12, 18)	**115.8 (9, 12, 5)**
MIDORI [BBI+15]	79.9 (24)	74.5 (0, 16)	**71.9 (16, 4)**	**71.9 (16, 4)**	**71.9 (16, 4)**	**71.9 (16, 4, 0)**
PRINCE M_0, M_1 [BCG+12]	79.9 (24)	74.5 (0, 16)	**71.9 (16, 4)**	**71.9 (16, 4)**	**71.9 (16, 4)**	**71.9 (16, 4, 0)**
PRIDE $L_0 - L_3$ [ADK+14]	79.9 (24)	74.5 (0, 16)	**71.9 (16, 4)**	**71.9 (16, 4)**	**71.9 (16, 4)**	**71.9 (16, 4, 0)**
QARMA128 [Ava17]	159.8 (48)	-	145.8 (34, 7)	145.8 (34, 7)	**144.5 (28, 11)**	**144.5 (28, 11, 0)**
QARMA64 [Ava17]	79.9 (24)	74.5 (0, 16)	**71.9 (16, 4)**	**71.9 (16, 4)**	**71.9 (16, 4)**	**71.9 (16, 4, 0)**
SMALLSCALE AES [CMR05]	143.1 (43)	**111.8 (0, 24)**	123.8 (19, 13)	123.8 (19, 13)	121.8 (17, 14)	118.4 (5, 9, 10)
TWOFISH [SKW+98]	369.6 (111)	317.5 (17, 56)	338.9 (43, 42)	312.9 (31, 45)	**306.9 (25, 48)**	**293.5 (13, 28, 20)**

[a] Using 2-input xor gates.
[b] Using 2/3-input xor gates.
[c] Using 2/3/4-input xor gates.

Table 4: The results of AES MixColumns in the UMC 55 nm library. "Ours" is from our algorithm. "Syn" is from the synthesizer Synopsys Design Compiler version R-2020.09-SP4.

Type	**Area** (GE)	**Latency** (us)	**Power** (uW)
1. [LXZZ21]	227.5	0.52	17.5
2. **Ours**	**220.0**	0.65	16.0
3. [LWF+22]	257.5	**0.28**	15.9
4. Syn	251.0	0.37	**15.2**

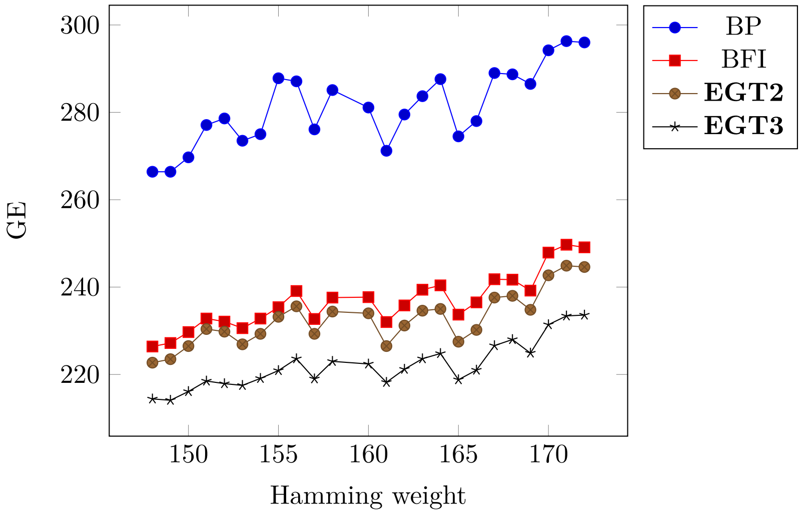

Figure 2: The results of lightweight matrices with the Hamming weight from 148 to 172 for STM 130 nm library, where the Hamming weight counts the number of 1's contained in the matrix.

2 Preliminaries

2.1 Notations

Let $\mathbb{F}_2$ be the finite field with two elements 0 and 1 and $\mathbb{F}_2^n$ be the vector space of all n-dimensional vectors over $\mathbb{F}_2$. $M_{m\times n}$ denotes an $m \times n$ matrix over $\mathbb{F}_2$. $wt(M)$ denotes the Hamming weight of a matrix M over $M_{m\times n}$, which counts the number of 1's contained in M.

2.2 g_ϵ-XOR Metric

For any linear layer of a cipher associated to an $m \times n$ binary matrix M, given inputs $\vec{x} = (x_0, x_1, ..., x_{n-1})^T$, the outputs $\vec{y} = (y_0, y_1, ..., y_{m-1})^T$ of the linear layer can be computed by $\vec{y} = M\vec{x}$, and y_i $(0 \le i \le m-1)$ can be computed by

$$y_i = a_{i0}x_0 \oplus a_{i1}x_1 \oplus ... \oplus a_{i(n-1)}x_{n-1}, \tag{1}$$

where each coefficient a_{ij} is the entry of M at i-th row and j-th column.

We first recall some metrics to compute $M\vec{x}$ with less number of 2-input xor gates for a matrix M over $M_{m\times n}$.

Definition 1 (d-XOR [KPPY14])**.** The d-XOR is defined as $wt(M) - m$, where $wt(M)$ is the Hamming weight of M, i.e., the number of 1's in M.

Definition 2 (s-XOR [JPST17])**.** It is always possible to perform a sequence of XOR gates $x_i = x_i \oplus x_j$ with $0 \le i, j \le n-1$, such that the inputs are updated to the outputs. The s-XOR count of M is defined as the minimal number of updating operations.

Definition 3 (g-XOR [XZL+20])**.** The circuit of M can be viewed as a sequence of XOR gates $x_i = x_{j_1} \oplus x_{j_2}$ where $0 < x_{j_1}, x_{j_2} < i$. The g-XOR count is defined as the minimal number of operations $x_i = x_{j_1} \oplus x_{j_2}$ that compute the m outputs completely.

Since the d-XOR metric is intuitive and easy to compute (i.e., the number of 1's in M), it has been adopted to design new lightweight diffusion layers. The s-XOR and g-XOR metrics are used in evaluating matrices for further optimization. The difference is that the g-XOR can generate new values while the s-XOR continuously renews original values. For example, in the procedure of computing $x_0 \oplus x_1$, the s-XOR performs $x_0 = x_0 \oplus x_1$ or $x_1 = x_0 \oplus x_1$, while the g-XOR can generate t_2 with $t_2 = x_0 \oplus x_1$.

Next, we use a new metric for optimization. We define the ϵ-operation ($\epsilon \in \mathbb{N}$) as an operation containing ϵ continuous 2-input xor gates. 1-operation represents a 2-input xor gate, 2-operation represents a 3-input xor gate, 3-operation represents a 4-input xor gate, and so on. Different operations may have different costs (i.e., GE in hardware). λ_ϵ is defined as the cost of the ϵ-operation. Then, we expand the definition of g-XOR. Actually, the definition is similar to [BDK+21].

Definition 4 (g_ϵ-XOR)**.** Given the cost λ_i $(1 \leq i \leq \epsilon)$ of every operation, the g_ϵ-XOR metric is defined as

$$\min(\lambda_1 e_1 + \lambda_2 e_2 + \ldots + \lambda_\epsilon e_\epsilon), \tag{2}$$

where e_i counts the number of the i-operation.

If $\lambda_1 = 1$, g_1-XOR is the g-XOR. The g_ϵ-XOR metric can use 1-operation, 2-operation,. . ., ϵ-operation to optimize matrices. The circuit with 1-operation, 2-operation,. . ., ϵ-operation are called the circuit with the g_ϵ-XOR metric. For convenience, we use XOR2, XOR3, and XOR4 to represent the 1-operation, 2-operation, and 3-operation, respectively.

Another metric is the circuit depth. The critical path of a circuit is defined as the path between an input and output involving the maximum number of gates. The circuit depth is the number of gates involved in the critical path.

2.3 Directed Acyclic Graph

A graph is formed by nodes and by edges connecting pairs of nodes. In the case of a directed graph, each edge has an orientation from one node to another. A Directed Acyclic Graph (DAG) is a directed graph that has no cycles. *In-degree* of a node is defined as the number of edges that end at this node, and *out-degree* of a node is defined as the number of edges whose origin is the node. We use $in(u)$ and $out(u)$ to represent the in-degree and out-degree of u, respectively. Moreover, the input set $I(u)$ and the output set $O(u)$ are used to save the nodes relevant to u. If there exists an edge from u to v, we will put v into $O(u)$ and put u into $I(v)$. We have

$$|I(v)| = in(v), |O(u)| = out(u).$$

Definition 5 (**Reachability Relation**)**.** The reachability relation can be formalized as a partial order $\preceq$ on the nodes of the DAG. In this partial order, two nodes u and v are ordered as $u \preceq v$ exactly when a directed path exists from u to v in the DAG.

Definition 6 (**Reachability Set**)**.** Given a directed acyclic graph G, the reachability set R_u of the node u $(u \in G)$ is defined as the set in which each node v satisfies $v \in G$ and $u \preceq v$. The reachability set R_G of the graph G is defined as the set containing all the R_u.

The definition of the reachability set shows that if one node $v \in R_u$, the path from u to v must exist. A path consists of multiple consecutive edges. Then, we introduce the topological ordering, which is used to sort the nodes.

Definition 7 (**Topological Ordering**)**.** The topological ordering T_G of a directed acyclic graph G is an ordering of its nodes into a sequence. For every edge, the start node of the edge occurs earlier in the sequence than the ending node.

3 Transform Algorithm

In this section, we introduce the generalized algorithm for converting a circuit with gates up to n inputs into a circuit with gates up to $n+1$ inputs. Reducing the number of XOR2 gates is helpful for linear layers. The circuit area can be reduced by finding the minimum number of XOR2 gates. However, there is a gap between the above circuit and the smallest circuit area. For example, the number of XOR2 gates for the matrix used in AES MixColumns is 92, proposed by Xiang *et al.* in [XZL$^+$20]. Banik *et al.* proposed a new circuit with 39 XOR2 gates and 28 XOR3 gates. In the library named STM 130 nm, the area of the new circuit is 260.3 GE. In comparison, the area of the 92-gate circuit is 306.3 GE.

To adapt to different situations, we limit the types of gates. If we use g_ϵ-XOR metric, only i-input ($i \leq \epsilon + 1$) xor gates can be used. For example, the g_2-XOR metric means that only 2/3-input xor gates can be used. Transforming circuits with gates from n inputs into $n+1$ inputs means that the metric used in the circuits is changed from g_{n-1}-XOR metric to g_n-XOR metric. The cost of the circuit is changed from

$$\lambda_1 e_1 + \lambda_2 e_2 + ... + \lambda_{n-1} e_{n-1}$$

to

$$\lambda_1 e'_1 + \lambda_2 e'_2 + ... + \lambda_{n-1} e'_{n-1} + \lambda_n e'_n,$$

where e_i and e'_i are the number of i-operations before and after the transformation.

Notably, for the AES MixColumns in the library named STM 130 nm, we can decrease the circuit area to 255 GE with the g_2-XOR metric (see Table 5) and to 243 GE with the g_3-XOR metric (see Table 6).

Table 5: An implementation of AES MixColumns with 255 GE for STM 130 nm library. It uses 29 XOR2 gates and 34 XOR3 gates. $t_0, t_1, \ldots, t_{31}$ are the input values. The output values are $y_0, y_1, \ldots, y_{31}$.

No.	Operation	No.	Operation	No.	Operation
1	$t_{32} = t_7 \oplus t_{15}$	22	$t_{67} = t_{33} \oplus t_{40} \oplus t_{38}$	43	$t_{105} = t_{35} \oplus t_4 \oplus t_{84}$ $//y_{20}$
2	$t_{33} = t_{31} \oplus t_{15}$	23	$t_{69} = t_{37} \oplus t_{14} \oplus t_6$	44	$t_{107} = t_{84} \oplus t_{126}$ $//y_{28}$
3	$t_{35} = t_{11} \oplus t_3 \oplus t_{32}$	24	$t_{71} = t_{33} \oplus t_8 \oplus t_{16}$	45	$t_{108} = t_{14} \oplus t_{43} \oplus t_{47}$ $//y_{30}$
4	$t_{36} = t_{17} \oplus t_9$	25	$t_{72} = t_{33} \oplus t_{14} \oplus t_{41}$ $//y_7$	46	$t_{110} = t_{39} \oplus t_{71} \oplus t_9$
5	$t_{37} = t_5 \oplus t_{21}$	26	$t_{73} = t_{27} \oplus t_{35} \oplus t_{39}$	47	$t_{111} = t_{96} \oplus t_{89} \oplus t_{45}$ $//y_{18}$
6	$t_{38} = t_{20} \oplus t_{12}$	27	$t_{75} = t_{33} \oplus t_{35} \oplus t_{45}$	48	$t_{112} = t_{13} \oplus t_{38} \oplus t_{52}$ $//y_{13}$
7	$t_{39} = t_{31} \oplus t_{23}$	28	$t_{76} = t_{16} \oplus t_{48}$	49	$t_{113} = t_{13} \oplus t_{38} \oplus t_{79}$ $//y_{21}$
8	$t_{40} = t_{11} \oplus t_{27}$	29	$t_{79} = t_{47} \oplus t_{46}$	50	$t_{114} = t_{79} \oplus t_{56}$ $//y_5$
9	$t_{41} = t_{23} \oplus t_6$	30	$t_{82} = t_{36} \oplus t_2 \oplus t_{44}$ $//y_{10}$	51	$t_{115} = t_0 \oplus t_{71} \oplus t_{32}$ $//y_{24}$
10	$t_{43} = t_{22} \oplus t_6$	31	$t_{83} = t_{33} \oplus t_{36} \oplus t_{48}$	52	$t_{116} = t_{110} \oplus t_{96}$ $//y_9$
11	$t_{44} = t_{18} \oplus t_{26}$	32	$t_{84} = t_{19} \oplus t_{73} \oplus t_{46}$	53	$t_{117} = t_{69} \oplus t_{125}$ $//y_{14}$
12	$t_{45} = t_{18} \oplus t_2$	33	$t_{88} = t_{19} \oplus t_{62} \oplus t_{32}$ $//y_3$	54	$t_{118} = t_{72} \oplus t_{43} \oplus t_{32}$ $//y_{15}$
13	$t_{46} = t_{12} \oplus t_{28}$	34	$t_{89} = t_{63} \oplus t_{44} \oplus t_9$ $//y_2$	55	$t_{119} = t_{43} \oplus t_{92} \oplus t_{39}$ $//y_{23}$
14	$t_{47} = t_5 \oplus t_{29}$	35	$t_{91} = t_0 \oplus t_{39} \oplus t_{48}$ $//y_{16}$	56	$t_{120} = t_{25} \oplus t_{115} \oplus t_{76}$
15	$t_{48} = t_8 \oplus t_{24}$	36	$t_{92} = t_{32} \oplus t_{41} \oplus t_{30}$ $//y_{31}$	57	$t_{121} = t_{96} \oplus t_{120}$ $//y_{25}$
16	$t_{51} = t_{37} \oplus t_{13}$	37	$t_{95} = t_{44} \oplus t_{73} \oplus t_{32}$ $//y_{19}$	58	$t_{122} = t_{83} \oplus t_{120}$ $//y_1$
17	$t_{52} = t_{29} \oplus t_{51}$	38	$t_{96} = t_{25} \oplus t_{36} \oplus t_1$	59	$t_{123} = t_{71} \oplus t_{91}$ $//y_8$
18	$t_{56} = t_4 \oplus t_{28} \oplus t_{51}$ $//y_{29}$	39	$t_{97} = t_{63} \oplus t_{25} \oplus t_{45}$ $//y_{26}$	60	$t_{124} = t_{116} \oplus t_{83}$ $//y_{17}$
19	$t_{59} = t_{35} \oplus t_{38} \oplus t_{28}$ $//y_4$	40	$t_{99} = t_{76} \oplus t_{32}$ $//y_0$	61	$t_{125} = t_{101} \oplus t_{43} \oplus t_{52}$ $//y_6$
20	$t_{62} = t_{10} \oplus t_{40} \oplus t_2$	41	$t_{101} = t_{69} \oplus t_{47} \oplus t_{30}$ $//y_{22}$	62	$t_{126} = t_{67} \oplus t_{105}$ $//y_{12}$
21	$t_{63} = t_{10} \oplus t_1$	42	$t_{102} = t_{19} \oplus t_{44} \oplus t_{75}$ $//y_{27}$	63	$t_{127} = t_{88} \oplus t_{75} \oplus t_{39}$ $//y_{11}$

3.1 Relationship between Circuit and DAG

We treat the circuit of a matrix as a directed acyclic graph. Given an $m \times n$ binary matrix M, the inputs $\vec{x} = (x_0, x_1, ..., x_{n-1})^T$, and the outputs $\vec{y} = (y_0, y_1, ..., y_{m-1})^T$, we call

Table 6: An implementation of AES MixColumns with 243 GE for STM 130 nm library. It uses 22 XOR2 gates, 21 XOR3 gates, and 12 XOR4 gates. $t_0, t_1, \dots, t_{31}$ are the input values. The output values are $y_0, y_1, \dots, y_{31}$.

No.	Operation	No.	Operation	No.	Operation
1	$t_{32} = t_{31} \oplus t_{23}$	20	$t_{69} = t_{11} \oplus t_{36} \oplus t_{32} \oplus t_{45}$ //y_{19}	38	$t_{108} = t_{56} \oplus t_{40} \oplus t_{29}$ //y_{29}
2	$t_{33} = t_{32} \oplus t_{19}$	21	$t_{71} = t_{38} \oplus t_{12} \oplus t_{40}$ //y_{20}	39	$t_{110} = t_{50} \oplus t_{56} \oplus t_{78}$ //y_{30}
3	$t_{34} = t_{31} \oplus t_{15}$	22	$t_{73} = t_9 \oplus t_{17}$	40	$t_{111} = t_{79} \oplus t_{43} \oplus t_9$ //y_{17}
4	$t_{35} = t_{12} \oplus t_4$	23	$t_{74} = t_{20} \oplus t_{36} \oplus t_{39} \oplus t_{35}$ //y_{28}	41	$t_{112} = t_{79} \oplus t_{39} \oplus t_{102}$ //y_{24}
5	$t_{36} = t_{27} \oplus t_3$	24	$t_{75} = t_{37} \oplus t_8$	42	$t_{113} = t_{37} \oplus t_{32} \oplus t_{34} \oplus t_{16}$ //y_8
6	$t_{37} = t_0 \oplus t_{24}$	25	$t_{76} = t_{28} \oplus t_{20} \oplus t_{56} \oplus t_{21}$ //y_{21}	43	$t_{114} = t_{47} \oplus t_{32} \oplus t_{93}$ //y_{31}
7	$t_{38} = t_{27} \oplus t_{33}$	26	$t_{77} = t_6 \oplus t_{50} \oplus t_{21} \oplus t_{29}$ //y_{22}	44	$t_{115} = t_{47} \oplus t_{77} \oplus t_{56}$ //y_6
8	$t_{39} = t_{31} \oplus t_7$	27	$t_{78} = t_{30} \oplus t_{47} \oplus t_{41}$ //y_{14}	45	$t_{116} = t_{81} \oplus t_{73} \oplus t_{10}$ //y_2
9	$t_{40} = t_{28} \oplus t_4$	28	$t_{79} = t_{32} \oplus t_{16} \oplus t_{24}$	46	$t_{117} = t_{76} \oplus t_{57} \oplus t_{40} \oplus t_{29}$ //y_{13}
10	$t_{41} = t_{13} \oplus t_{21}$	29	$t_{80} = t_{67} \oplus t_{20}$ //y_{12}	47	$t_{118} = t_{43} \oplus t_{17} \oplus t_{75} \oplus t_{113}$ //y_9
11	$t_{42} = t_2 \oplus t_{18}$	30	$t_{81} = t_{45} \oplus t_1 \oplus t_{17}$	48	$t_{119} = t_{114} \oplus t_{39} \oplus t_{50}$ //y_7
12	$t_{43} = t_1 \oplus t_{25}$	31	$t_{86} = t_{37} \oplus t_{39} \oplus t_{73} \oplus t_1$ //y_{25}	49	$t_{120} = t_{38} \oplus t_{69} \oplus t_{39} \oplus t_{42}$ //y_{27}
13	$t_{45} = t_{26} \oplus t_{18}$	32	$t_{88} = t_{26} \oplus t_{36} \oplus t_{10} \oplus t_{34}$	50	$t_{121} = t_{88} \oplus t_{33} \oplus t_{45}$ //y_{11}
14	$t_{47} = t_6 \oplus t_{22}$	33	$t_{90} = t_{26} \oplus t_{73} \oplus t_{42}$ //y_{10}	51	$t_{122} = t_{88} \oplus t_{120}$ //y_3
15	$t_{50} = t_{30} \oplus t_{14}$	34	$t_{93} = t_{34} \oplus t_{51}$ //y_{23}	52	$t_{123} = t_{81} \oplus t_{96}$ //y_{18}
16	$t_{51} = t_{30} \oplus t_7 \oplus t_{22}$	35	$t_{95} = t_{32} \oplus t_{50} \oplus t_{51}$ //y_{15}	53	$t_{125} = t_{79} \oplus t_{86} \oplus t_{17} \oplus t_{118}$ //y_1
17	$t_{56} = t_5 \oplus t_{41} \oplus t_{29}$	36	$t_{96} = t_{43} \oplus t_{10} \oplus t_{42}$ //y_{26}	54	$t_{126} = t_{37} \oplus t_{112} \oplus t_{34}$ //y_0
18	$t_{57} = t_{35} \oplus t_{41} \oplus t_{29}$ //y_5	37	$t_{102} = t_{32} \oplus t_{75}$ //y_{16}	55	$t_{127} = t_{67} \oplus t_{38} \oplus t_{74}$ //y_4
19	$t_{67} = t_{11} \oplus t_{33} \oplus t_{34} \oplus t_{40}$				

x_i $(0 \leq i \leq n-1)$ the *unit node* and y_j $(0 \leq j \leq m-1)$ the *target node*. Searching for circuits means finding a DAG from all the unit nodes to all the target nodes. Besides, the depth of a graph is defined as the number of edges involved in its critical path. According to Liu *et al.*, every unit node has the in-degree 0, and every *non-unit* node has the in-degree n $(n \geq 2)$ [LWF+22]. The following property shows the features of nodes in DAG.

Property 1. The circuit with g_ϵ-XOR metric can be converted into a DAG, in which every unit node has in-degree 0 and every non-unit node has in-degree n $(n \leq \epsilon + 1)$ and represents an $(n+1)$-input xor gate (i.e., n-operation).

3.2 Transforming DAG

Given the available operations, 1-operation, 2-operation, …, ϵ-operation, and their cost λ_1, λ_2, …, λ_ϵ, we can obtain the cost of a DAG:

$$\lambda_1 e_1 + \lambda_2 e_2 + ... + \lambda_\epsilon e_\epsilon,$$

where e_i $(i \leq \epsilon)$ counts the number of the nodes with in-degree $i+1$. The core idea of the transform algorithm is how to reduce the cost by removing nodes in the DAG.

Our first concern is what happens when we remove nodes from a DAG. Removing u means that we delete the edges from u to $O(u)$ and from $I(u)$ to u and add the edges from every node in $I(u)$ to every node in $O(u)$. Meanwhile, the in-degree of every node in $O(u)$ increases.

Next, we discuss which nodes can be removed to reduce the cost of the DAG. Suppose that we have a direct acyclic graph G with g_ϵ-XOR metric. We define S_u as the reduced cost by removing u from G. S_u can be computed by considering the change of in-degree of the node in $O(u)$ and related to hardware libraries. Thus, $S_u > 0$ means that we can benefit from removing u.

A general approach is to compute the reduced cost S_u of every node in G and choose the maximum cost. The transformation will stop if every S_u is smaller than 0. However, given two nodes u and v, it is difficult to determine which is better. Different gates and libraries lead to different costs, and different metrics also influence the comparison. For the case with the fixed library and metric, we can usually find a standard to compare them. In Section 5, we give the comparisons with g_2-XOR and g_3-XOR metrics.

Notably, not all the nodes can be removed. We can only use the i-operation ($i \leq \epsilon$), which means that the in-degree of a node u is not greater than $\epsilon + 1$. Thus, we have the following proposition.

Proposition 1. *Suppose that the circuit is with the g_ϵ-XOR metric and $in(u) = j$. Only when the in-degree k of every node in $O(u)$ is not greater than $\epsilon + 2 - j$, can we remove u.*

Proof. Removing u means that we delete the edges from u to $O(u)$ and from $I(u)$ to u and add the edges from every node in $I(u)$ to every node in $O(u)$. After removing u, the in-degree of the node in $O(u)$ will increase by $j - 1$. In the g_ϵ-XOR metric, the in-degree of every node is not greater than $\epsilon + 1$. Thus, we have

$$k + j - 1 \leq \epsilon + 1.$$

This means

$$k \leq \epsilon + 2 - j.$$

□

Therefore, we can propose the generalized transform algorithm.

1. Suppose that we need to transform the graph G from $g_{\epsilon-1}$-XOR metric into g_ϵ-XOR metric. Initialize the set A_G, in which every node u meets that the reduced cost $S_u > 0$.
2. Compute the reduced cost of S_u for every node $u \in A_G$ and choose the node v with the maximum value S_v and remove v from A_G. If $S_v > 0$, go to Step 3. Otherwise, stop the procedures.
3. Check whether v can be removed. If v cannot be removed, go to Step 2. Otherwise, remove v and corresponding edges, and let $I(v)$ point to $O(v)$. Then, go to Step 2.

Because many operations are related to the specific libraries and gates, we instantiate two algorithms, EGT2 and EGT3, based on the following graph extending algorithm in two libraries, STM 90 nm and STM 130 nm. These two instances show that our algorithms perform well.

4 Graph Extending Algorithm

In this section, we propose the graph extending algorithm, which is a local optimization algorithm. There are many local techniques based on specific reduction rules (see [TP20, LXZZ21]). However, these rules cannot cover all the cases. We find some cases in which no reduction rules are effective. Therefore, we first show an example and propose our algorithm formally. The new algorithm converts a circuit into many circuits and fully utilizes the circuit's information. Note that new circuits may have fewer gates and a smaller depth.

We take the circuits with XOR2 as an example. The graph extending algorithm also holds for other operations (e.g., XOR3 and XOR4). For a matrix M over $M_{m\times n}$, a circuit of M can be seen as a sequence of l XOR2 gates $t_i = t_j \oplus t_k$ where $i = n, n+1, ..., n+l-1$ and $j, k < i$. We use an operation $\boldsymbol{t_{i,j,k}}$ instead of $t_i = t_j \oplus t_k$ for convenience. We say that the *implementation* of t_i is (t_j, t_k) and t_j and t_k are the *predecessor* nodes of t_i in the DAG. The circuit is represented as $seq = t_{n,j_0,k_0}, t_{n+1,j_1,k_1}, ..., t_{n+l-1,j_{l-1},k_{l-1}}$. Next, we introduce the issue briefly.

4.1 An Issue in Reduction Rules

Although Lin *et al.* tried to derive all the potential reduction rules in [LXZZ21], we still find other cases where some nodes can be removed while they are not included in any rules. Given a matrix M_P,

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix},$$

and the circuit,

$$t_{8,0,1}, t_{9,2,3}, t_{10,4,5}, t_{11,6,7}, t_{\underline{12},8,9}, t_{13,9,10}, t_{\underline{14},4,11}, t_{\underline{15},5,11}, t_{\underline{16},12,13}, t_{\underline{17},13,14},$$

in which $t_0, t_1, \dots t_7$ are the input values. $t_{\underline{12},8,9}$ implies that t_{12} is the output of the circuit. Removing that 2-input xor gate will be counter-productive because we will have lost the output value t_{12}. The operation tuple $(t_{12,8,9}, t_{13,9,10}, t_{16,12,13})$ satisfies the rules in [LXZZ21] as $t_{16} = t_8 \oplus t_9$. However, t_{12} is the output value and t_{13} is used in $t_{16,12,13}$ and $t_{17,13,14}$. This means that we cannot remove them. Thus, the circuit needs **ten** 2-input xor gates.

If we rearrange the implementation, the new circuit will be

$$t_{8,0,1}, t_{9,2,3}, t_{10,4,5}, t_{11,6,7}, t_{\underline{12},8,9}, t_{13,9,10}, t_{\underline{14},4,11}, t_{\underline{15},5,11}, \boldsymbol{t_{\underline{16},8,10}}, \boldsymbol{t_{\underline{17},9,15}}. \tag{3}$$

The implementations of t_{16} and t_{17} are changed. We observe that t_{13} is not used in any operations and t_{13} is not the output value, which is *redundant*. Thus, we can remove $t_{13,9,10}$, and the circuit only needs **nine** 2-input xor gates.

The example shows that there is still room for further improvements, and it is not enough to consider whether a node can be removed by checking the out-degree of the node. We need to explore more features of circuits. Our graph extending algorithm can optimize a given circuit and generate many equivalent circuits. The algorithm can be used in any heuristics to optimize the generated circuits. The application to multi-input gates can be seen in the next section.

4.2 Single Graph and Extended Graph

The reachability set and the topological ordering have been introduced. Our graph extending algorithm uses them to optimize a given DAG. We give some necessary explanations.

The definition of the reachability set of u shows which nodes are generated by u. Note that if we have $u \preceq v$ and $v \preceq w$, $u \preceq w$ holds. This property can help us to find the reachability set quickly. We initialize two temporary sets $temp_1$ and $temp_2$. For each node $u \in G$, we execute the following steps.

1. Let $R_u = \phi$, $temp_1 = \phi$, and $temp_2 = \phi$. Then, put $O(u)$ into $temp_1$.
2. If $temp_1 = \phi$, return R_u. Otherwise, go to Step 3.
3. For each node $v \in temp_1$, we put v into R_u, put $O(v)$ into $temp_2$, and remove v from $temp_1$. When $temp_1 = \phi$, we let $temp_1 = temp_2$ and $temp_2 = \phi$. Then, go to Step 2.

The procedures will be performed iteratively for each node in G and obtain the reachability set R_G. We use GetReachabilitySet() to calculate the reachability set of a graph.

Another definition is the topological ordering. The ordering implies the reachability relation. If one node a occurs earlier than b in the topological ordering, we can infer that the relationship $b \preceq a$ does not hold. The problem of finding a topological ordering can be solved in linear time by Kahn's algorithm [Kah62]. The strategy is as follows:

1. Suppose that graph G contains n nodes. The topological ordering T is ϕ, and S contains the nodes with in-degree 0.
2. If $S \neq \phi$, go to Step 3. Otherwise, we stop the search procedures. If T contains n nodes, return T. If T contains m nodes ($m < n$), return *error*.
3. For node $v \in S$, we remove v and corresponding edges from G and let the in-degree of nodes in $O(v)$ decrease by 1. Next, we recheck all the nodes in G and put the nodes with in-degree 0 into S. Then, go to Step 2.

We use TopologicalOrdering() to represent the strategy. In Step 2, we have two possible outputs. If the algorithm returns T, we can get the topological ordering. However, if the algorithm returns *error*, there exist *cycles* in G. The cycle is defined as a path from one node to this node. The node in the cycle cannot occur in T since its in-degree is always greater than 0. If a cycle exists in the graph, it will be the wrong circuit for the linear layer. Because each node has only one implementation in the graph, the nodes in the cycle cannot be calculated by unit nodes.

We introduce two types of graphs used in the graph extending algorithm. The definitions of the *single graph* and *extended graph* are shown as follows.

Definition 8. The *single graph* is a directed graph so that each non-unit node has only an implementation. The *extended graph* is the directed graph so that each node can have more than one implementation.

In the single graph, each non-unit node has one implementation. We can add new implementations to the nodes in the single graph to generate another DAG instance, the extended graph, in which the in-degree of every non-unit can be more than 2.

4.3 Graph Extending Algorithm

In [LXZZ21], the authors check all the reduction rules and remove some nodes which are only used once. Our algorithm says that the nodes only have the out-degree 1. If the out-degree is greater than 1, their rules do not work. However, as is shown in Subsection 4.1, there still exist redundant nodes hiding in the graph. Our algorithm is performed as follows:

1. Generate the extended graph from a single graph.
2. Split the extended graph into many single graphs.
3. Remove redundant nodes from every single graph.
4. Delete wrong graphs.

Totally, the goal of our algorithm is to generate many equivalent graphs and optimize them. The circuit of a linear layer can be treated as a single graph. We generate the extended graph by adding different implementations. Then, the extended graph can be split into many single graphs. After removing redundant nodes, we can choose the best one from proper graphs. We take the XOR2 as an example. Other gates can also be used. The following section will use XOR3 to generate the extended graph. We explain every step in detail, introduce complete procedures, and provide an example of the matrix M_P used in Subsection 4.1.

Generate the extended graph with XOR2. From Property 1, we know that the unit node has the in-degree 0, and the non-unit node has the in-degree 2 and represents one XOR2 gate. However, in previous work, each node had only one implementation, and other implementations were ignored. Suppose that we have unit nodes $\{x_0, x_1, x_2\}$ and the sequence,

$$
\begin{aligned}
t_0 &= x_0 \oplus x_1 \\
t_1 &= x_1 \oplus x_2. \\
t_2 &= t_0 \oplus x_2
\end{aligned}
$$

t_2 can also be generated by $t_2 = t_1 \oplus x_0$. If $out(t_0)$ is 0, we can remove t_0 from the circuit. Note that $out(t_0)$ and $out(t_1)$ are changed in the above steps. Thus, the transformation helps consider other implementations in the single graph.

Algorithm 1 gives a method to generate the extended graph. First, for each non-unit node u, we try to find different implementations of u where some nodes will not be utilized. For example, we will not use a node to generate its predecessor nodes, i.e., if $u = a \oplus b$, we do not use u to generate a or b. It may lead to cycles in the graph. The nodes that can generate u are called the *available nodes*. We use the *available set* $\mathcal{A}_u$ to save the available nodes of u. For a node u' ($u' \neq u$), either $u' \in R_u$ or $u' \in A_u$ holds. *op* in the algorithm decides which operation can be used. This paper only uses XOR2 and XOR3. $op = 2$ means using XOR2 to generate the extended graph. $op = 3$ means that we use XOR3. More operations can be used in the algorithm, and we omit them for the sake of brevity. In this section, we only consider the XOR2 gates. This means that we will try all the combinations of two nodes in A_u to generate u. If we find a new implementation $u = p \oplus q$, we will add edges from p to u and from q to u. Thus, the in-degree of every non-unit is a multiple of 2, which guarantees that the extended graph can be split. Finally, we get the extended graph G_e.

The matrix M_P in 4.1 is taken as an example. Suppose that G_s has the sequence,

$$t_{8,0,1}, t_{9,2,3}, t_{10,4,5}, t_{11,6,7}, t_{\underline{12},8,9}, t_{13,9,10}, t_{\underline{14},4,11}, t_{\underline{15},5,11}, t_{\underline{16},12,13}, t_{\underline{17},13,14}.$$

We can use GetReachabilitySet(G_s) to obtain the reachability set (see Table 7). Then, we use GenerateExtendedGraph(G_s, 2) to obtain the extended graph G_e,

$$
\begin{aligned}
&t_{8,0,1}, t_{9,2,3}, \{t_{10,4,5}, t_{10,14,15}\}, t_{11,6,7}, t_{\underline{12},8,9}, t_{13,9,10}, \{t_{\underline{14},4,11}, t_{\underline{14},10,15}\}, \\
&\{t_{\underline{15},5,11}, t_{\underline{15},10,14}\}, \{t_{\underline{16},12,13}, t_{\underline{16},8,10}\}, \{t_{\underline{17},13,14}, t_{\underline{17},9,15}\}.
\end{aligned}
\tag{4}
$$

If a node has different implementations, we use "{}" to represent them. For example, $\{t_{10,4,5}, t_{10,14,15}\}$ means that t_{10} can be generated by (t_4, t_5) or (t_{14}, t_{15}). We find that five nodes have different implementations:

$$t_{10}, t_{14}, t_{15}, t_{16}, t_{17}.$$

Table 7: The reachability sets.

Nodes	reachability sets
t_0, t_1	$\{t_8, t_{12}, t_{16}\}$
t_2, t_3	$\{t_9, t_{12}, t_{13}, t_{16}, t_{17}\}$
t_4	$\{t_{10}, t_{13}, t_{14}, t_{16}, t_{17}\}$
t_5	$\{t_{10}, t_{13}, t_{15}, t_{16}, t_{17}\}$
t_6, t_7	$\{t_{11}, t_{14}, t_{15}, t_{17}\}$
t_8	$\{t_{12}, t_{16}\}$
t_9	$\{t_{12}, t_{13}, t_{16}, t_{17}\}$
t_{10}	$\{t_{13}, t_{16}, t_{17}\}$
t_{11}	$\{t_{14}, t_{15}, t_{17}\}$
t_{12}	$\{t_{16}\}$
t_{13}	$\{t_{16}, t_{17}\}$
t_{14}	$\{t_{17}\}$
t_{15}, t_{16}, t_{17}	ϕ

Split the extended graph. In the extended graph G_e, some nodes have many implementations, i.e., every implementation can generate the corresponding node. Thus,

Algorithm 1 GenerateExtendedGraph()

```
Input: A single graph $G_s$ and the operation $op$ (2 or 3)
Output: An extended graph $G_e$
  $R_{G_s}$ = GetReachabilitySet($G_s$)
  if TopologicalOrdering($G_s$) = $error$ then                ▷ Checking the cycles
      return $error$
  end if
  $T$ = TopologicalOrdering($G_s$)
  $G_e \leftarrow G_s$
  for $i$ from 1 to $\mid T \mid -1$ do      ▷ Checking whether two nodes has the same value
      $u = T[i]$
      for $j$ from $i+1$ to $\mid T \mid$ do
          $v = T[j]$
          if $u = v$ then
              Remove $v$ and let the origin of each edge whose origin is $v$ be $u$
          end if
      end for
  end for
  for each $u \in G_s$ do                     ▷ Generating the extended graph $G_c$
      if $u$ is not unit node then
          $\mathcal{A} \leftarrow \phi$                       ▷ The available set of all the nodes
          for each $v \in G_s/\{u\}$ do
              if $v$ not in $R_u$ then
                  $\mathcal{A} \leftarrow \mathcal{A} \cup \{v\}$
              end if
          end for
          if ($\mid \mathcal{A} \mid< 2$ and $op = 2$ ) or ($\mid \mathcal{A} \mid< 3$ and $op = 3$ ) then
              Continue
          end if
          if $op = 2$ then
              for $w, v \in \mathcal{A}(w \neq v)$ do                       ▷ Using XOR2
                  if $u = w \oplus v$ and $u$ has not the implementation $(w, v)$ then
                      Add $(w, v)$ for $u$ in $G_e$          ▷ Adding a new implementation for $u$
                  end if
              end for
          end if
          if $op = 3$ then
              for $w, v, p \in \mathcal{A}(w \neq v \neq p)$ do                 ▷ Using XOR3
                  if $u = w \oplus v \oplus p$ and $u$ has not the implementation $(w, v, p)$ then
                      Add $(w, v, p)$ for $u$ in $G_e$       ▷ Adding a new implementation for $u$
                  end if
              end for
          end if
      end if
  end for
  return $G_e$
```

a large number of single graphs can be generated by using the property. However, the complexity increases exponentially as the number of implementations of nodes increases. Theoretically, it may exceed the existing computing power in generating single graphs from the G_e. Therefore, we provide two methods to split the extended graph G_e.

We define n_i as the number of implementations of each node t_i in G_e. Suppose that there are k nodes in G_e. The number N of single graphs can be computed by:

$$N = \prod_{i=0}^{k-1} n_i. \tag{5}$$

Then, we define the limitation N' as the maximum number of single graphs. If $N \leq N'$, we split the extended graph using the *complete split* method. If $N > N'$, we use the *partial split* method.

1. The complete split method. It traverses all the implementations of each node in G_s and generates corresponding single graphs:

$$G_{s_0}, G_{s_1}, \ldots, G_{s_{N-1}}.$$

2. The partial split method. Suppose that we add m additional implementations in G_e. We set $G_{s_0} = G_s$. We replace the corresponding original implementation in G_s and generate a new single graph for each additional implementation. Finally, we will generate $m + 1$ single graphs:

$$G_{s_0}, G_{s_1}, \ldots, G_{s_m}.$$

We use a function SplitExtendedGraph() to execute the procedures. It chooses a suitable method to split the extended graph.

Then, we still take Equation (4) as an example. The total number N is 32. We use the complete split method to split G_e and generate 32 single graphs:

$$\begin{array}{ll}
G_{s_0}: & t_{8,0,1}, t_{9,2,3}, \boldsymbol{t}_{\mathbf{10,4,5}}, t_{11,6,7}, t_{\underline{12},8,9}, t_{13,9,10}, \boldsymbol{t}_{\underline{\mathbf{14}}\mathbf{,4,11}}, \boldsymbol{t}_{\underline{\mathbf{15}}\mathbf{,5,11}}, \boldsymbol{t}_{\underline{\mathbf{16}}\mathbf{,12,13}}, \boldsymbol{t}_{\underline{\mathbf{17}}\mathbf{,13,14}}, \\
G_{s_1}: & t_{8,0,1}, t_{9,2,3}, t_{11,6,7}, t_{\underline{12},8,9}, t_{13,9,10}, \boldsymbol{t}_{\underline{\mathbf{14}}\mathbf{,4,11}}, \boldsymbol{t}_{\underline{\mathbf{15}}\mathbf{,5,11}}, \boldsymbol{t}_{\mathbf{10,14,15}}, \boldsymbol{t}_{\underline{\mathbf{16}}\mathbf{,12,13}}, \boldsymbol{t}_{\underline{\mathbf{17}}\mathbf{,13,14}}, \\
G_{s_2}: & t_{8,0,1}, t_{9,2,3}, \boldsymbol{t}_{\mathbf{10,4,5}}, t_{11,6,7}, t_{\underline{12},8,9}, t_{13,9,10}, \boldsymbol{t}_{\underline{\mathbf{15}}\mathbf{,5,11}}, \boldsymbol{t}_{\underline{\mathbf{14}}\mathbf{,10,15}}, \boldsymbol{t}_{\underline{\mathbf{16}}\mathbf{,12,13}}, \boldsymbol{t}_{\underline{\mathbf{17}}\mathbf{,13,14}}, \\
\ldots & \ldots \\
G_{s_{31}}: & t_{8,0,1}, t_{9,2,3}, t_{11,6,7}, t_{\underline{12},8,9}, t_{13,9,10}, \boldsymbol{t}_{\mathbf{10,14,15}}, \boldsymbol{t}_{\underline{\mathbf{14}}\mathbf{,10,15}}, \boldsymbol{t}_{\underline{\mathbf{15}}\mathbf{,10,14}}, \boldsymbol{t}_{\underline{\mathbf{16}}\mathbf{,8,10}}, \boldsymbol{t}_{\underline{\mathbf{17}}\mathbf{,9,15}}.
\end{array}$$

Remove redundant nodes. After splitting the extended graph, the nodes in different single graphs may have different in-degrees and out-degrees. The out-degree 0 means that the node is not used to generate other nodes. Thus, we use the following property.

Property 2. Given a DAG, if $out(u) = 0$, u must be the target node or the redundant node.

If a non-unit node with the out-degree 0 is not the target node, we call it the redundant node. Removing the redundant nodes from the graph can decrease the number XOR2 gates. In the example of M_P, t_{13} in Equation (3) is the redundant node and can be removed. We give the procedures to remove redundant nodes as follows.

1. Suppose that we need to remove redundant nodes in G. We set a variable *success* $= 1$. It implies whether one node is removed.

2. If *success* $= 0$, return G. Otherwise, go to Step 3.

3. We set $\mathit{success} = 0$ and check every non-input node in G. If v with $\mathit{out}(v) = 0$ is not the target node, we remove v and the corresponding edges from G, and set $\mathit{success} = 1$. Next, the out-degree of each node in $I(v)$ decreases by 1. Then, go to Step 2.

We use the function RemovingRedundantNodes() to remove redundant nodes. Note that there is a loop in the function. When we remove one node, we set $\mathit{success} = 1$ and recheck the graph because the out-degree of each node in $I(v)$ changes. New redundant nodes may occur.

In the example of 32 single graphs, each graph includes 10 non-unit nodes. We apply the function RemovingRedundantNodes() to all single graphs. 18 graphs still have 10 non-unit nodes, 12 graphs have 9 non-unit nodes, and 2 graphs have 8 non-unit nodes.

Delete wrong graphs. The wrong graph is an incorrect circuit for the corresponding matrix, which usually contains the cycles in the graph. Unit nodes cannot generate the nodes in the cycle.

Table 8: The reduced graphs.

Reduced graph 1	$t_{8,0,1}, t_{9,2,3}, \boldsymbol{t}_{\mathbf{10,4,5}}, t_{11,6,7}, t_{\underline{12},8,9}, \boldsymbol{t}_{\underline{\mathbf{14}},\mathbf{4,11}}, \boldsymbol{t}_{\underline{\mathbf{15}},\mathbf{5,11}}, \boldsymbol{t}_{\underline{\mathbf{16}},\mathbf{8,10}}, \boldsymbol{t}_{\underline{\mathbf{17}},\mathbf{9,15}},$
Reduced graph 2	$t_{8,0,1}, t_{9,2,3}, t_{11,6,7}, t_{\underline{12},8,9}, \boldsymbol{t}_{\underline{\mathbf{14}},\mathbf{4,11}}, \boldsymbol{t}_{\underline{\mathbf{15}},\mathbf{5,11}}, \boldsymbol{t}_{\mathbf{10,14,15}}, \boldsymbol{t}_{\underline{\mathbf{16}},\mathbf{8,10}}, \boldsymbol{t}_{\underline{\mathbf{17}},9,15},$
Reduced graph 3	$t_{8,0,1}, t_{9,2,3}, \boldsymbol{t}_{\mathbf{10,4,5}}, t_{11,6,7}, t_{\underline{12},8,9}, \boldsymbol{t}_{\underline{\mathbf{15}},\mathbf{5,11}}, \boldsymbol{t}_{\underline{\mathbf{14}},\mathbf{10,15}}, \boldsymbol{t}_{\underline{\mathbf{16}},\mathbf{8,10}}, \boldsymbol{t}_{\underline{\mathbf{17}},\mathbf{9,15}},$
Reduced graph 4	$t_{8,0,1}, t_{9,2,3}, \boldsymbol{t}_{\mathbf{10,4,5}}, t_{11,6,7}, t_{\underline{12},8,9}, \boldsymbol{t}_{\underline{\mathbf{14}},\mathbf{4,11}}, \boldsymbol{t}_{\underline{\mathbf{15}},\mathbf{10,14}}, \boldsymbol{t}_{\underline{\mathbf{16}},\mathbf{8,10}}, \boldsymbol{t}_{\underline{\mathbf{17}},\mathbf{9,15}}.$

Although we try to avoid this case (e.g., the available set), the cycle may occur in single graphs. We use the function TopologicalOrdering() to execute the procedure. If TopologicalOrdering() returns *error*, cycles must exist in the graph, and we delete the graph. After the step, we call the left graphs the *reduced graphs*. They are the results of our graph extending algorithm. We can choose the best one from all the reduced graphs or further optimize them with more gates.

In the example of M_P, 16 graphs are finally left, in which 12 graphs have 10 non-unit nodes, and 4 graphs have 9 non-unit nodes. We show the graphs with 9 non-unit nodes in Table 8.

Now our graph extending algorithm is finished. After the graph extending algorithm, we obtain different reduced graphs. Some of them have less cost in hardware. If we plan to optimize a single graph further, the reduced graph will provide more precise information. In the above example, if we take the number of 2-input xor gates and the depth into account, we choose the 1-*st* and 3-*rd* reduced graphs with depth 3 from Table 8. They are the best circuits after our graph extending algorithm. The complete algorithm of our graph extending algorithm is shown in Algorithm 2. We can use other operations (e.g., XOR3) to generate the extended graph. We will discuss it in the next section.

5 Applications

In this section, we instantiate the transform algorithms. With the help of the graph extending algorithm, we propose two algorithms to optimize the given circuit using XOR3 and XOR4 gates, respectively. We start from a circuit with XOR2 gates. For the g_ϵ-XOR metric with $\epsilon \geq 4$, we can follow similar procedures, and thus we do not discuss them in this section. The source codes are available at https://github.com/QunLiu-sdu/Using-Gates-with-More-Than-Two-Inputs.

Algorithm 2 ExtendGraph2()

```
Input: A single graph $G_s$
Output: The set $\mathcal{G}_2$ containing all the reduced graphs
  $G_e$ = GenerateExtendedGraph($G_s$, 2)            ▷ The extended graph
  $\mathcal{G}_2$ = SplitExtendedGraph($G_e$)        ▷ Generating the single graphs
  for each $G_r \in \mathcal{G}_2$ do                ▷ Removing additional nodes
      $G_r$ = RemovingRedundantNodes($G_r$)
  end for
  for each $G_r \in \mathcal{G}_2$ do                ▷ Deleting wrong graphs
      if TopologicalOrdering($G_r$) = error then
          $\mathcal{G}_2 \leftarrow \mathcal{G}_2/\{G_r\}$
      end if
  end for
  return $\mathcal{G}_2$
```

5.1 Transforming Gates from 2 Inputs into 3 Inputs

We first focus on the g_2-XOR metric starting the circuits with XOR2 gates, i.e., we try to convert gates from 2 inputs into 3 inputs. In the problem, we can use 1-operation and 2-operation. λ_1 and λ_2 represent the corresponding cost. Our goal is to find

$$\min(\lambda_1 e_1 + \lambda_2 e_2). \tag{6}$$

If $v = a \oplus b \oplus c$, we say that v has implementation (a, b, c). We consider a case where the out-degree of one node is 1, which is discussed in [BFI21]. If v_i $(out(v_i) = 1)$ represents an XOR2 gate $v_i = v_x \oplus v_y$, we merge them to an XOR3 gate, $v_{i,j} = v_x \oplus v_y \oplus v_z$. Note that if v_i is the output value, we will lose the output signal of the circuit after the merge procedure.

According to our transform algorithm, more nodes can be removed. Suppose that we have the circuit:

$$u = a \oplus b$$
$$v = u \oplus c$$
$$w = u \oplus d$$

and $out(u)$ is 2. The circuit area is $3\lambda_1$. If $3\lambda_1 - 2\lambda_2 > 0$ holds, we can remove u and let $I(u)$ point to $O(u)$ by merging two 3-input xor gates. The new circuit is

$$v = a \oplus b \oplus c$$
$$w = a \oplus b \oplus d$$

and the area of the new circuit is $2\lambda_2 < 3\lambda_1$.

We can remove more nodes based on the above cases to reduce the cost. Thus, we give the following proposition.

Proposition 2. *Let N be the maximum value such that $(N+1)\lambda_1 - N\lambda_2 > 0$ holds. Given a circuit with XOR2 gates, it can reduce the cost by removing the nodes with out-degree n $(n \le N)$.*

Proof. Suppose that $out(u)$ is n $(n \le N)$. There are $n + 1$ related XOR2 gates. One of them is used to generate u. Others are used to generate the nodes in $O(u)$. Thus, removing u will delete $(n + 1)$ XOR2 gates and add n XOR3 gates. If $(n + 1)\lambda_1 - n\lambda_2 > 0$, the cost of the circuit is reduced. □

Proposition 2 shows which nodes can be removed in g_2-XOR metric. For convenience, the maximum value N is called the *upper bound*. If one node has the out-degree of

n ($n \leq N$), we can remove it by deleting $n + 1$ XOR2 gates and adding n XOR3 gates. Another problem is if the nodes with different out-degrees can be removed, how to determine the priority. We provide a proposition to solve the problem.

Proposition 3. *Suppose that the upper bound is N. If $out(u) = m$ and $out(v) = n$ ($n < m \leq N$), removing v will reduce more cost than u. That is, $S_v > S_u$.*

Proof. We can remove u by deleting $m + 1$ XOR2 gates and adding m XOR3 gates. The saved cost S_u is $(m + 1)\lambda_1 - m\lambda_2$. We can also remove v by deleting $n + 1$ XOR2 gates and adding n XOR3 gates. The saved cost S_v is $(n + 1)\lambda_1 - n\lambda_2$. We have

$$S_v - S_u = (m - n)(\lambda_2 - \lambda_1) > 0 \tag{7}$$

Thus, $S_v > S_u$ holds. □

We propose our algorithm using graph extending algorithm called EGT2 (see Algorithm 3). ExtendGraph2() is used to obtain the set $\mathcal{G}_2$ that contains many reduced graphs. For each graph G_r in $\mathcal{G}_2$, we execute the following procedures:

1. Compute the upper bound N. We use $\mathcal{U}$ to save the nodes that can not be removed. $u \in \mathcal{U}$ means that u is the target node or u represents the XOR3 gate.
2. Set $n = 1$. Then, Step 3 is recursively executed. Each time we finish Step 3, we set $n = n + 1$. If $n \leq N$, we continue to execute Step 3. Otherwise, we stop the procedures and put the new graph G_r into $\mathcal{G}_3$.
3. Check the nodes in the topological ordering. $O(u) \cap \mathcal{U} \neq \phi$ means that at least one node in $O(u)$ has been optimized by the XOR3 gate and cannot be merged again. If the out-degree of one node is n and the node is not the target node, we will remove u by adding n XOR3 gates and deleting $n + 1$ XOR2 gates based on Proposition 2. Then, we put the nodes in $O(u)$ into $\mathcal{U}$.

We explain why $O(u) \cap \mathcal{U} = \phi$ in Step 3 is necessary. Suppose that we have the circuit:

$$t_{u,a,b}, t_{v,c,d}, t_{w,u,v}, t_{y,p,u},$$

in which $out(v) = 1$ and $out(u) = 2$. When $n = 1$, we can remove v and have the new circuit:

$$t_{u,a,b}, \boldsymbol{t^3_{w,u,c,d}}, t_{y,p,u}.$$

Next, we have $\mathcal{U} = \mathcal{U} \cup \{v\}$. Then, we set $n = 2$. However, we cannot remove u since $O(u) = \{w, y\}$ and w represents an XOR3 gate. Thus, only when $O(u) \cap \mathcal{U} = \phi$ holds, can u be removed.

Because of the topological ordering property, in Step 3, the next node that needs to be checked can always be generated by the checked nodes. If we combine all the checked nodes to a new graph, a path must exist from unit nodes to every non-unit node, and no nodes can be removed with the current upper bound n. Every non-unit node u in the new graph may have three states:

- $out(u)$ is greater than the current upper bound n. We cannot remove u in the current state based on the Proposition 2.
- $out(u)$ is m ($m < n$). Based on the Proposition 3, we have checked the u in the previous procedures. If u has been left, we must have $O(u) \cap \mathcal{U} \neq \phi$ or $u \cap \mathcal{U} \neq \phi$.
- $out(u)$ is n. We have checked u in Step 3 and cannot remove u.

In addition, according to Proposition 3, we first check the nodes with out-degree 1. When we have checked all the nodes, Step 3 is finished, and we set $n = n + 1$. If we match the condition when we check one node, we will remove the node, delete corresponding edges, and add corresponding edges to the graph. We use n XOR3 gates to replace $n + 1$

Algorithm 3 EGT2()

```
Input: A single graph $G_s$
Output: A set $\mathcal{G}_3$ containing all the reduced graphs with 2/3-input xor gates
  $\mathcal{G}_2$ = ExtendGraph2($G_s$)          ▷ Containing the reduced graphs with XOR2 gates
  $N \leftarrow 0$                                ▷ The upper bound
  while $((N+1)+1)\lambda_1 - (N+1)\lambda_2 > 0$ do
      $N \leftarrow N+1$
  end while
  for each $G_r \in \mathcal{G}_2$ do             ▷ Removing nodes
      $\mathcal{T}$ = TopologicalOrdering($G_r$)
      The set $\mathcal{U}$ containing all the target nodes in $G_r$
      $n \leftarrow 1$
      while $n \leq N$ do
          for each node $u$ in $\mathcal{T}$ do
              if $u \notin \mathcal{U}$, $out(u) = n$, and $O(u) \cap \mathcal{U} = \phi$ then
                  We remove $u$, delete corresponding edges, and add $n$ operations in $G_r$
                  Put the nodes in $O(u)$ into $\mathcal{U}$
              end if
          end for
          $n \leftarrow n+1$
      end while
      $\mathcal{G}_3 \leftarrow \mathcal{G}_3 \cup \{G_r\}$
  end for
  return $\mathcal{G}_3$
```

XOR2 gates. The saved cost is S_v is $(n+1)\lambda_1 - n\lambda_2$. We set the upper bound $N = 0$ if $2\lambda_1 < \lambda_2$ because the cost of two XOR2 gates is less than one XOR3 gate. Not all the libraries can apply our algorithm. We only take the libraries with $2\lambda_1 > \lambda_2$ into account.

We also take the matrix in Subsection 4.1 as an example. Using the BFI algorithm, the reduced circuit area is $6 \times 3.33 + 2 \times 4.66 = 29.3$ GE (six 2-input xor gates and two 3-input xor gates). The initial sequence is shown in Subsection 4.1. We first run Algorithm 2 and get the $\mathcal{G}_2$. From $\mathcal{G}_2$, we choose one graph G_r with the circuit:

$$t_{8,0,1}, t_{9,2,3}, t_{10,4,5}, t_{11,6,7}, t_{\underline{12},8,9}, t_{\underline{14},4,11}, t_{\underline{15},5,11}, t_{\underline{16},8,10}, t_{\underline{17},9,15}.$$

Then, we get the upper bound $N = 2$ $((2+1) \times 3.33 - 2 \times 4.66 > 0)$. We set $n = 1$ and find that t_{10} can be removed. The left circuit is:

$$t_{8,0,1}, t_{9,2,3}, t_{11,6,7}, t_{\underline{12},8,9}, t_{\underline{14},4,11}, t_{\underline{15},5,11}, \boldsymbol{t^{3}_{\underline{16},4,5,8}}, t_{\underline{17},9,15}.$$

Next, we set $n = 1 + 1 = 2$ and find that t_9 and t_{11} can be removed. The left circuit is:

$$t_{8,0,1}, \boldsymbol{t^{3}_{\underline{12},2,3,8}}, \boldsymbol{t^{3}_{\underline{14},4,6,7}}, \boldsymbol{t^{3}_{\underline{15},5,6,7}}, t^{3}_{\underline{16},4,5,8}, \boldsymbol{t^{3}_{\underline{17},2,3,15}}. \tag{8}$$

The new circuit area is only 26.6 GE (one XOR2 gate and five XOR3 gates).

5.2 Transforming Gates from 3 Inputs into 4 Inputs

Then, we focus on the g_3-XOR metric starting the circuits with 2/3-input xor gates, i.e., we try to convert gates from 2/3 inputs into 4 inputs. The operations we can use are 1-operation, 2-operation, and 3-operation. The corresponding costs are λ_1, λ_2, and λ_3. Our goal is to find

$$\min(\lambda_1 e_1 + \lambda_2 e_2 + \lambda_3 e_3). \tag{9}$$

For a given circuit with XOR2 gates, we use Algorithm 3 to convert the initial graph into many graphs with 2/3-input xor gates and save them in $\mathcal{G}_3$. For each G_r in $\mathcal{G}_3$, we use a new function ExtendGraph3() to generate the extended graph G_e with 2/3-input xor gates. The new function is similar to ExtendGraph2() and use GenerateExtendedGraph$(G_r, 3)$ instead of GenerateExtendedGraph$(G_r, 2)$. Then, we split G_e into many single graphs. For each new single graph, we optimize it with XOR4 gates.

Similar to the above section, we discuss which nodes can be removed. In our algorithm, $t^4_{u,a,b,c,d}$ represents an XOR4 gate. We propose three *circuit types*, which can be transformed into 4-input xor gates.

Type 1. If $t_{u,p,q}$ and $t^3_{p,a,b,c}$ are contained in the circuit, we can obtain $t^4_{u,q,a,b,c}$ by removing p. We say that p matches Type 1 (see Figure 4-left).

Type 2. If $t^3_{u,p,q,w}$ and $t_{p,a,b,}$ are contained in the circuit, we can obtain $t^4_{u,a,b,q,w}$ by removing p. We say that p matches Type 2 (see Figure 4-middle).

Type 3. If $t_{u,p,q}$, $t_{p,a,b}$, and $t_{q,c,d}$ are contained in the circuit, we can obtain $t^4_{u,a,b,c,d}$ by removing p and q. p and q are called the nodes in Type 3 (see Figure 4-right).

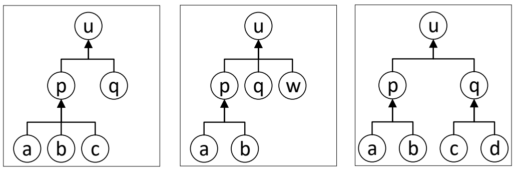

Figure 3: The different circuit types.

We take Type 1 and Type 2 into account because the nodes in Type 3 can be transformed into Type 2. The following observations can help us to simplify the the analysis procedures.

Observation 1. *If p matches Type 1, $in(p) = 3$. If p matches Type 2, $in(p) = 2$. Thus, if p matches one type, it never matches another type.*

Observation 2. *Suppose that we have a node p and its output set $O(p)$. For any node $q \in O(p)$, if p matches one type, other nodes in $I(q)$ will never match another type.*

The above types only consider the case where the out-degree of a node is 1. Next, we discuss the case in which the out-degree is greater than 1. Suppose that p matches Type 1 or Type 2 and $O(p) = \{u_1, u_2, \dots u_n\}$ $(n \geq 2)$. If p matches Type 1, we can use n XOR4 gates instead of an XOR3 gate and n XOR2 gates. Figure 4-left shows the case where $out(p)$ is 2. If p matches Type 2, there exist two cases:

- u_i $(1 \leq i \leq n)$ has the in-degree 2;
- u_i $(1 \leq i \leq n)$ has the in-degree 3.

For the first case, we use an XOR3 gate instead of an XOR2 gate additionally (see Figure 4-middle). For the second case, we use an XOR4 gate instead of an XOR3 gate additionally (see Figure 4-right). To decide which nodes can be removed, we can obtain following proposition by extending Proposition 2.

Proposition 4. *Let N_1 be the maximum value such that $\lambda_2 + N_1(\lambda_1 - \lambda_3) > 0$ holds and N_2 be the maximum value such that $\lambda_1 + \lambda_2 - \lambda_3 - (N_2 - 1) \cdot min((\lambda_2 - \lambda_1), (\lambda_3 - \lambda_2)) > 0$ holds. It can reduce the cost if*

- *the node matches Type 1 and has the out-degree m $(m \leq N_1)$, or*
- *the node matches Type 2 and has the out-degree n $(n \leq N_2)$.*

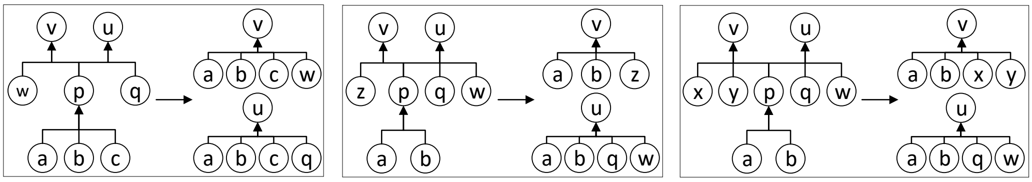

Figure 4: The cases that the *out*(p) is 2.

Proof. Suppose that p matches Type 1 and $out(p)$ is m ($m \leq N_1$). There are m related XOR2 gates and a related XOR3 gate. The XOR3 gate is used to generate p. Other gates are used to generate new nodes by p. Thus, we can remove p by adding m XOR4 gates and deleting m XOR2 gates and an XOR3 gate. If $\lambda_2 + m(\lambda_1 - \lambda_3) > 0$, the circuit area decreases.

Suppose that p matches Type 2 and $out(p)$ is n ($n \leq N_2$). If $n = 1$, We can use an XOR4 gate instead of an XOR2 gate and an XOR3 gate. Then, the circuit area decreases by $\lambda_1 + \lambda_2 - \lambda_3 > 0$. If p is also used to generate new node u, there are two different cases. If $in(u) = 2$ ($u \in O(p)$), we will use an XOR3 gate instead of an XOR2 gate. If $in(u) = 3$ ($u \in O(p)$), we will use an XOR4 gate instead of an XOR3 gate. Thus, we choose $\min((\lambda_2 - \lambda_1), (\lambda_3 - \lambda_2))$. The remained proof is similar to Proposition 2. We do not repeat it. □

Another question is if the nodes with different out-degrees can be removed, how to determine the priority. Proposition 3 provides a solution. If the nodes with different out-degrees can be removed, we discuss how to determine the priority of each node. Suppose that we have two nodes u and v, and $out(u) = m$, $out(v) = n$ ($m < n$). We discuss this on a case-by-case basis.

- If u and v are not the same types, we just deal with them in order. Based on Observation (2), we have $O(u) \cap O(v) = \phi$. The operations between u and v are independent.
- If two nodes u and v match the same type, we will have $S_u > S_v$ because of $m < n$. The proof is similar to Proposition 3.

Thus, we can search for two types separately. We provide an algorithm called EGT3 (see Algorithm 4). The procedures are as follows.

1. Run Algorithm 3 to obtain $\mathcal{G}_3$ containing different graphs with 2/3-input xor gates.
2. For each graph G_r in $\mathcal{G}_3$, run the function GenerateExtendedGraph$(G_r, 3)$ to obtain the extended graph G_e, split it into different single graphs, and put all the single graphs into $\mathcal{G}_4$.
3. Compute N_1 and N_2. Let $n_1 = 1$ and $n_2 = 1$. We use $\mathcal{U}$ to save the nodes that can not be optimized. $u \in \mathcal{U}$ means that u is the target node or u represents the XOR4 gate.
4. If $n_1 > N_1$ and $n_2 > N_2$, we stop the procedures. Otherwise, go to Step 5.
5. Check the nodes in the topological ordering. If u matches Type 1, $out(u) = n_1$ ($n_1 \leq N_1$), $O(u) \cap \mathcal{U} = \phi$, and $u \notin \mathcal{U}$, we can remove u by adding n_1 XOR4 gates and deleting an XOR2 gate and n_1 XOR3 gates. Then, we let $n_1 = n_1 + 1$ and go to Step 6.
6. Check the nodes in the topological ordering. If u matches Type 2, $out(u) = n_2$ ($n_2 \leq N_2$), $O(u) \cap \mathcal{U} = \phi$, and $u \notin \mathcal{U}$, we can remove u. We check $O(u)$ and for each node $u_i \in O(u)$,
   - add an XOR3 gate and delete an XOR2 gate (u_i is the first case in Type 2);

– add an XOR4 gate and delete an XOR3 gate (u_i is the second case in Type 2).

Then, we let $n_2 = n_2 + 1$ and go to Step 4.

Note that we will stop the algorithm when $n_1 > N_1$ and $n_2 > N_2$. For example, if $n_1 < N_1$ and $n_2 > N_2$, we can optimize the nodes which match Type 1. We can search the types sequentially based on Observation (2). When we finish one search, we set $n_1 = n_1 + 1$ and $n_2 = n_2 + 1$. Through the algorithm, we can further optimize the circuit with XOR4 gates.

We still take the circuit in Subsection 4.1 as an example. We have $N_1 = 1$ and $N_2 = 2$. Finally, we can find the circuit with 24.6 GE. The circuit requires four XOR3 gates and one XOR4 gate. Note that only the target nodes are left. Other nodes have been removed. Thus, we cannot find a better circuit with less area. The sequence is:

$$t^4_{\underline{12},0,1,2,3}, t^3_{\underline{14},4,6,7}, t^3_{\underline{15},5,6,7}, t^3_{\underline{17},2,3,15}, t^3_{\underline{16},12,14,17}. \tag{10}$$

5.3 Experiments

We apply our algorithms to many lightweight matrices. The circuits are generated by the algorithm proposed by Xiang *et al.* in [XZL+20], which is called the XZLBZ algorithm. The metrics are g_2-XOR and g_3-XOR.

5.3.1 XZLBZ Algorithm

In FSE 2020, Xiang *et al.* proposed a new heuristic based on the matrix decomposition [XZL+20]. Given an invertible matrix, the author first decomposed it as a product of elementary matrices. The original problem is converted into the problem of finding a better decomposition. Note that XZLBZ algorithm uses s-XOR. It updates original values and cannot be used in our algorithm directly. We use a method converting the s-XOR results into g-XOR results. Given a sequence of s-XOR operations, we execute the following procedures:

1. Suppose that the input values are $\{x_0, x_1, \ldots x_{m-1}\}$. Put each input value into a set S in order. That is to say, we have $S[i] = x_i$.
2. We set a variable $n = 0$ and a set $C = \phi$. For each operation $x_i = x_i \oplus x_j$, we transform it in Step 3.
3. We generate a new value $t_n = x_i$, and set $n = n + 1$. The new operation op is $t_n = S[i] \oplus S[j]$. Then, we set $S[i] = t_n$ and put op into C.
4. Finally, C contains all the new sequence.

We give an example to illustrate the method. Suppose that the input values are $\{x_0, x_1, x_2, x_3\}$ and the initial sequence is

$$\begin{aligned} x_0 &= x_0 \oplus x_1 \\ x_0 &= x_0 \oplus x_2. \\ x_0 &= x_0 \oplus x_3 \end{aligned}$$

After transforming the operations, we have the new sequence,

$$\begin{aligned} \boldsymbol{t_0} &= x_0 \oplus x_1 \\ \boldsymbol{t_1} &= \boldsymbol{t_0} \oplus x_2. \\ \boldsymbol{t_2} &= \boldsymbol{t_1} \oplus x_3 \end{aligned}$$

Algorithm 4 EGT3()

```
Input: A single graph $G_s$
Output: A set $\mathcal{G}_4$ containing all the reduced graphs with 2/3/4-input gates
  $\mathcal{G}_4 \leftarrow \phi$
  $\mathcal{G}_3 \leftarrow \phi$
  $\mathcal{G}_2 = \text{EGT2}(G_s)$
  for each graph $G_r$ in $\mathcal{G}_2$ do
      $\mathcal{G}' = \text{EGT3}(G_r)$
      for each graph $G$ in $\mathcal{G}'$ do
          $\mathcal{G}_3 \leftarrow \mathcal{G}_3 \cup \{G\}$
      end for
  end for
  $N_1 \leftarrow 0$
  $N_2 \leftarrow 0$
  while $\lambda_2 + (N_1+1)(\lambda_1 - \lambda_3) > 0$ do
      $N_1 \leftarrow N_1 + 1$
  end while
  while $\lambda_1 + \lambda_2 - \lambda_3 - N_2 \cdot \min((\lambda_2 - \lambda_1), (\lambda_3 - \lambda_2)) > 0$ do
      $N_2 \leftarrow N_2 + 1$
  end while
  for each $G_r \in \mathcal{G}_3$ do                                   ▷ Removing nodes
      $\mathcal{T} = \text{TopologicalOrdering}(G_r)$
      The set $\mathcal{U}$ containing all the target nodes in $G_r$
      $n_1, n_2 \leftarrow 1$
      while $n_1 \le N_1$ or $n_2 \le N_2$ do
          for each node $u$ in $\mathcal{T}$ do
              if $n_1 \le N_1$, $u$ matches Type 1, $O(u) \cap \mathcal{U} = \phi$, and $u \notin \mathcal{U}$ then
                  Remove $u$, delete corresponding edges, and add edges from $I(u)$ to $O(u)$
                  Put the nodes in $O(u)$ into $\mathcal{U}$
              end if
              if $n_2 \le N_2$, $u$ matches Type 2, $O(u) \cap \mathcal{U} = \phi$, and $u \notin \mathcal{U}$ then
                  Remove $u$, delete corresponding edges, and add edges from $I(u)$ to $O(u)$
                  Put the nodes in $O(u)$ into $\mathcal{U}$
              end if
          end for
          $n_1 \leftarrow n_1 + 1$
          $n_2 \leftarrow n_2 + 1$
      end while
      $\mathcal{G}_4 \leftarrow \mathcal{G}_4 \cup \{G_4\}$
  end for
  return $\mathcal{G}_4$
```

5.3.2 Applying Our Algorithms to Many Proposed Matrices

In this section, we apply EGT2 and EGT3 to several linear layers from the literature, including matrices used in many ciphers [DR20, CMR05, JNP15, Ava17, BBI$^+$15, BCG$^+$12, ADK$^+$14, Ava17, BJK$^+$16, AIK$^+$00]. The results in ASIC1 are shown in Table 2 and the results in ASIC2 are shown in Table 3. We list the results in [XZL$^+$20], [BDK$^+$21], and [BFI21]. Note that [XZL$^+$20] only searches for circuits with the g_1-XOR metric. Then, we run the XZLBZ algorithm and optimize the circuits using the BFI, EGT2, and EGT3 algorithms.

It is worthy to say that the circuit with the minimum number of 2-input xor gates may have the worse performance with g_i-XOR metric ($i > 1$). For example, in Table 3, we use an circuit for AES MixColumns with 96 XOR2 gates instead of 92 gates. The area of the 96-gate circuit can be reduced to 255 GE by EGT2, while the 92-gate circuit cannot.

As the procedures of splitting the matrix into elementary matrices and generating s-XOR sequences are randomized, recalling the computations at different times would lead to different results. Hence, it is difficult to determine how long we wait to achieve the best solution. We execute the XZLBZ algorithm in a limited and reasonable time (e.g., 24 hours) to collect many implementations and select the best one among them. This strategy is quite similar to many previous search approaches in, e.g., [KLSW17, TP20, XZL$^+$20, BFI21].

In our experiments, many previous results can be optimized. Notably, for the matrix used in AES MixColumns, we achieve the circuit with 255 GE in ASIC2, while the best previous result is 258.9 GE [BDK$^+$21] in the same library. Moreover, we can decrease the circuit area to 243 GE with the help of XOR4 gates. This shows the effectiveness of our strategy.

Table 9: The results of implementation cost of matrices from [LSL$^+$19] for STM 90 nm library.

HW	Num	best of BFI	**best of EGT2**	BP (GE)	BP+BFI (GE)	**BP+EGT2** (GE)	**BP+EGT3** (GE)
148	18	11/18	**18/18**	160.0	145.0	**144.5**	**144.2**
149	48	23/48	**48/48**	160.0	145.3	**144.6**	**144.3**
150	72	27/72	**72/72**	162.0	147.0	**146.3**	**145.9**
151	48	25/48	**48/48**	166.4	149.8	**149.3**	**148.6**
152	60	43/60	**60/60**	167.3	149.9	**149.6**	**148.9**
153	72	33/72	**72/72**	164.3	148.2	**147.6**	**147.1**
154	84	45/84	**84/84**	165.1	149.3	**148.8**	**148.2**
155	24	20/24	**24/24**	172.8	153.2	**153.1**	**151.9**
156	72	28/72	**72/72**	172.4	154.4	**153.8**	**152.8**
157	96	34/96	**96/96**	165.8	149.6	**148.9**	**148.3**
158	156	76/156	**156/156**	171.2	153.4	**152.9**	**152.0**
160	210	93/210	**210/210**	168.8	152.6	**151.7**	**150.7**
161	144	37/144	**144/144**	162.9	148.1	**146.8**	**146.4**
162	204	78/204	**204/204**	167.9	151.4	**150.4**	**149.7**
163	192	78/192	**192/192**	170.4	153.8	**152.6**	**151.7**
164	300	116/300	**300/300**	172.7	155.0	**154.0**	**153.2**
165	312	84/312	**312/312**	164.8	149.4	**148.0**	**147.5**
166	324	63/324	**324/324**	167.0	151.2	**149.8**	**149.2**
167	336	121/336	**336/336**	173.6	155.9	**155.0**	**154.1**
168	600	314/600	**600/600**	173.4	155.7	**155.2**	**154.4**
169	380	146/384	**384/384**	172.1	154.1	**153.3**	**152.6**
170	504	204/504	**504/504**	176.7	159.1	**158.2**	**157.4**
171	528	172/528	**528/528**	177.9	160.4	**159.5**	**158.6**
172	762	283/762	**762/762**	177.8	160.1	**159.4**	**158.5**

5.3.3 Applying Our Algorithms to More Lightweight Matrices

In FSE 2019, Li *et al.* proposed 5500 lightweight matrices [LSL$^+$19] and applied the BP algorithm to optimize them. We also apply our algorithms to these matrices for comparison. We run the BP, BFI, EGT2, and EGT3 in two libraries for each matrix and then compare the obtained circuit areas. The results are listed in Table 9 and Table 10. “HW” is the Hamming weight of matrices. “num” is the number of the matrices with the same

Table 10: The results of implementation cost of matrices from [LSL+19] for STM 130 nm library.

HW	Num	best of BFI	**best of EGT2**	BP (GE)	BP+BFI (GE)	**BP+EGT2** (GE)	**BP+EGT3** (GE)
148	18	0/18	**18/18**	266.4	226.4	**222.7**	**214.4**
149	48	0/48	**48/48**	266.4	227.2	**223.5**	**214.1**
150	72	0/72	**72/72**	269.7	229.7	**226.5**	**216.1**
151	48	0/48	**48/48**	277.1	232.8	**230.4**	**218.5**
152	60	6/60	**60/60**	278.6	232.1	**229.8**	**217.9**
153	72	0/72	**72/72**	273.5	230.6	**226.9**	**217.5**
154	84	0/84	**84/84**	275.0	232.8	**229.3**	**219.1**
155	24	0/24	**24/24**	287.8	235.4	**233.2**	**220.9**
156	72	0/72	**72/72**	287.1	239.1	**235.6**	**223.6**
157	96	0/96	**96/96**	276.1	232.7	**229.3**	**219.0**
158	156	6/156	**156/156**	285.1	237.6	**234.4**	**223.0**
160	210	12/210	**210/210**	281.1	237.7	**234.0**	**222.4**
161	144	0/144	**144/144**	271.2	232.0	**226.5**	**218.2**
162	204	10/204	**204/204**	279.5	235.8	**231.2**	**221.2**
163	192	1/192	**192/192**	283.7	239.4	**234.6**	**223.6**
164	300	6/300	**300/300**	287.6	240.4	**235.0**	**224.8**
165	312	0/312	**312/312**	274.5	233.7	**227.5**	**218.8**
166	324	0/324	**324/324**	278.0	236.5	**230.2**	**221.0**
167	336	0/336	**336/336**	289.0	241.8	**237.6**	**226.6**
168	600	32/600	**600/600**	288.7	241.7	**238.0**	**228.0**
169	384	0/384	**384/384**	286.5	239.2	**234.8**	**224.9**
170	504	0/504	**504/504**	294.2	247.9	**242.7**	**231.4**
171	528	2/528	**528/528**	296.3	249.7	**244.9**	**233.4**
172	762	3/762	**762/762**	296.0	249.1	**244.6**	**233.6**

Hamming weight. "best of BFI" represents the proportion that BFI algorithm can obtain the best results and "best of EGT2" represents the proportion that our EGT2 can obtain the best results. "BP", "BP+BFI", "BP+EGT2", and "BP+EGT3" represent the circuit area of a matrix on average. Note that the BP algorithm uses g_1-XOR metric, both BFI algorithm and EGT2 use the g_2-XOR metric, and EGT3 uses the g_3-XOR metric.

From the results, there exists significant room for improvement. On the one hand, the method using XOR3 gates can optimize all the circuits generated by the BP algorithm. On the other hand, our algorithms have better performance. In ASIC1, BFI can obtain about 35% matrices with the best results. In ASIC2, the proportion decreases to 1.3%. EGT2 and EGT3 can always obtain the best results.

5.3.4 Hardware Implementations

Our algorithms aim at searching for low-area circuits. It is insufficient for other hardware metrics (e.g., latency and power consumption) to simply conduct the estimation based on the every single gate. These criteria are closely related to the standard cell library [BFI21, BDK+21]. In this respect, we synthesize the implementations with UMC 55 nm library (λ_1 = 2.5 GE, λ_2 = 4.5 GE). The logic synthesis is performed with Synopsys Design Compiler version R-2020.09-SP4 (using the compile_ultra and compile_ultra -no_autoungroup commands), and simulation is done in Mentor Graphics ModelSim SE v10.2c. All the results are shown in Table 4. We give some explanations. There are four different circuits to implement AES MixColumns, which are from different tools based on different metrics.

Type 1 is from [LXZZ21] and is the best result with g_1-metric. It only needs 91 XOR2 gates with depth 7.

Type 2 is the circuit found by our algorithms based on **Type 1**. We always focus on the area and generate an optimized circuit with 61 XOR2 gates and 15 XOR3 gates.

Type 3 is from [LWF+22] and is one of the best results with respect to the minimum depth (another best result is from [BFI21]). It needs 103 XOR2 gates with depth 3. We provide it to show the comparison in latency.

Type 4 is from the synthesizer starting from **Type 1**. The synthesizer takes a trade-off between multiple metrics. We provide it to show the comparison with the previous synthesizers.

For the AES MixColumns, our implementation achieves the best area and the reduction of energy consumption (1.49 uW) at the cost of slight and reasonable growth of latency (0.13 ns). The goal of our algorithms is to achieve implementations with the smallest area, and our results show that considering gates with more input bits offers more possibilities.

6 Conclusion

In this paper, we propose the transform algorithm and the graph extending algorithm, and then combine the two algorithms to instantiate EGT2 and EGT3 with g_2-XOR and g_3-XOR metrics, respectively. The two instantiated algorithms can be used to further optimize the circuit area of matrices in hardware. Our methods contribute to

- better circuits of AES MixColumns with 243 GE (with g_3-XOR in ASIC2), which is the best result than ever before;
- better circuits of several linear layers from the literature;
- better circuits for 100% of matrices proposed in [LSL+19].

Though our algorithms can reduce the circuit area easily with local optimization, it is still important to perform the procedures with the global optimization based on heuristics like [BDK+21]. In addition, our research provides a new tool for the construction of lightweight matrices. There should exist some matrices more compatible with our algorithms and thus have better circuits with g_ϵ-XOR metric, which we leave as future work.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. The research leading to these results has received funding from the National Natural Science Foundation of China (Grant No. 62032014, Grant No. 62002201, Grant No. 62002204), the National Key Research and Development Program of China (Grant No. 2018YFA0704702, Grant No. 2021YFA1000600), the Major Basic Research Project of Natural Science Foundation of Shandong Province, China (Grant No. ZR202010220025), and the Program of Qilu Young Scholars (Grant No. 61580082063088) of Shandong University.

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, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2014.

[AIK+00] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Camellia: A 128-bit block cipher suitable for multiple platforms - design and analysis. In Douglas R. Stinson and Stafford E. Tavares, editors, *Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000, Waterloo, Ontario, Canada, August 14-15, 2000, Proceedings*, volume 2012 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2000.

[Ava17] Roberto Avanzi. 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.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In Xiaoyun Wang and Kazue Sako, editors, *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*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[BDK+21] Anubhab Baksi, Vishnu Asutosh Dasu, Banashri Karmakar, Anupam Chattopadhyay, and Takanori Isobe. Three input exclusive-or gate support for boyar-peralta's algorithm. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 2021.

[BFI21] Subhadeep Banik, Yuki Funabiki, and Takanori Isobe. Further results on efficient implementations of block cipher linear layers. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 104-A(1):213–225, 2021.

[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, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 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, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BMP13] Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *J. Cryptol.*, 26(2):280–312, 2013.

[BP10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental*

Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2010.

[BPMC18] Anubhab Baksi, Vikramkumar Pudi, Swagata Mandal, and Anupam Chattopadhyay. Lightweight asic implementation of aegis-128. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 251–256, 2018.

[BR00] Paulo S.L.M. Barreto and Vincent Rijmen. The anubis block cipher. *First Open NESSIE Workshop*, 2000.

[CFS$^+$22] Shiyao Chen, Yanhong Fan, Ling Sun, Yong Fu, Haibo Zhou, Yongqing Li, Meiqin Wang, Weijia Wang, and Chun Guo. SAND: an AND-RX feistel lightweight block cipher supporting s-box-based security evaluations. *Des. Codes Cryptogr.*, 90(1):155–198, 2022.

[CMR05] Carlos Cid, Sean Murphy, and Matthew J. B. Robshaw. Small scale variants of the AES. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 2005.

[DR20] Joan Daemen and Vincent Rijmen. *The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition.* Information Security and Cryptography. Springer, 2020.

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[JNP15] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Joltik v1. 3. *CAESAR Round*, 2, 2015.

[JPST17] Jérémy Jean, Thomas Peyrin, Siang Meng Sim, and Jade Tourteaux. Optimizing implementations of lightweight building blocks. *IACR Trans. Symmetric Cryptol.*, 2017(4):130–168, 2017.

[JV04] Pascal Junod and Serge Vaudenay. FOX : A new family of block ciphers. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2004.

[Kah62] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[KLSW17] Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. Shorter linear straight-line programs for MDS matrices. *IACR Trans. Symmetric Cryptol.*, 2017(4):188–211, 2017.

[KPPY14] Khoongming Khoo, Thomas Peyrin, Axel York Poschmann, and Huihui Yap. FOAM: searching for hardware-optimal SPN structures and components with a fair comparison. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 433–450. Springer, 2014.

[LSL$^+$19] Shun Li, Siwei Sun, Chaoyun Li, Zihao Wei, and Lei Hu. Constructing low-latency involutory MDS matrices with lightweight circuits. *IACR Trans. Symmetric Cryptol.*, 2019(1):84–117, 2019.

[LWF$^+$22] Qun Liu, Weijia Wang, Yanhong Fan, Lixuan Wu, Ling Sun, and Meiqin Wang. Towards low-latency implementation of linear layers. *IACR Transactions on Symmetric Cryptology*, pages 158–182, 2022.

[LXZZ21] Da Lin, Zejun Xiang, Xiangyong Zeng, and Shasha Zhang. A framework to optimize implementations of matrices. In Kenneth G. Paterson, editor, *Topics in Cryptology - CT-RSA 2021 - Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17-20, 2021, Proceedings*, volume 12704 of *Lecture Notes in Computer Science*, pages 609–632. Springer, 2021.

[Paa97] Christof Paar. Optimized arithmetic for reed-solomon encoders. In *Proceedings of IEEE International Symposium on Information Theory*, page 250. IEEE, 1997.

[RMTA20] Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. New low-area designs for the aes forward, inverse and combined s-boxes. *IEEE Transactions on Computers*, 69(12):1757–1773, 2020.

[SKW$^+$98] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Two sh: A 128-bit block cipher. *AES submission*, 1998.

[SLN$^+$21] Kosei Sakamoto, Fukang Liu, Yuto Nakano, Shinsaku Kiyomoto, and Takanori Isobe. Rocca: An efficient aes-based encryption scheme for beyond 5g. *IACR Transactions on Symmetric Cryptology*, 2021(2):1–30, Jun. 2021.

[SSA$^+$07] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA (extended abstract). In Alex Biryukov, editor, *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2007.

[TP20] Quan Quan Tan and Thomas Peyrin. Improved heuristics for short linear programs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 203–230, 2020.

[WP13] Hongjun Wu and Bart Preneel. AEGIS: A fast authenticated encryption algorithm. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2013.

[XZL$^+$20] Zejun Xiang, Xiangyong Zeng, Da Lin, Zhenzhen Bao, and Shasha Zhang. Optimizing implementations of linear layers. *IACR Trans. Symmetric Cryptol.*, 2020(2):120–145, 2020.