Improved Heuristics for Short Linear Programs

Quan Quan Tan and Thomas Peyrin

Nanyang Technological University, Singapore
quanquan001@e.ntu.edu.sg, thomas.peyrin@ntu.edu.sg

Abstract. In this article, we propose new heuristics for minimising the amount of XOR gates required to compute a system of linear equations in GF(2). We first revisit the well known Boyar-Peralta strategy and argue that a proper randomisation process during the selection phases can lead to great improvements. We then propose new selection criteria and explain their rationale. Our new methods outperform state-of-the-art algorithms such as Paar or Boyar-Peralta (or open synthesis tools such as Yosys) when tested on random matrices with various densities. They can be applied to matrices of reasonable sizes (up to about 32×32). Notably, we provide a new implementation record for the matrix underlying the MixColumns function of the AES block cipher, requiring only 94 XORs.

Keywords: XOR gate · gate count · linear systems · diffusion matrices.

1 Introduction

Lightweight cryptography has received a lot of attention in the past decade and with the increasing miniaturisation of electronic chips and devices, this trend will probably continue further. In a general sense, the study of lightweight cryptography refers to finding cryptographic primitives circuits that are efficient in constrained environments. The definition of efficiency obviously differs depending on the use-case: throughput, area, energy, power, latency are classical dimensions of the problem that have been considered so far. Trade-offs are often possible and desirable for a cryptographic primitive to adapt well to various situations. Yet, circuit area is a crucial criterion for lightweight cryptography in hardware, as smaller and cheaper devices means stronger area constraints.

Many lightweight ciphers have been proposed [BKL+07, CDK09, GPPR12, BCG+12, BJK+16, BPP+17] and they all share the particularity that a lot of efforts has been spent on minimising the circuit area; for example by using a state as small as possible, or by minimising the number of logical gates (especially the expensive ones such as XOR) required to implement the primitive. For many of them, a small or even the smallest implementation of their subcomponents is provided by construction, but it is interesting more generally to study how one can minimise the area cost of any function (circuit depth can also be an important criterion).

Tools providing a good circuit that is efficient in area and/or depth already exist, but this task being very complex they all have limitations: they either provide optimal or close to optimal circuits for small problem sizes, or they provide non-optimal solutions for bigger function domains. Examples of the former are SAT-based tools [Sto16], or LIGHTER [JPST17]. For small circuits, LIGHTER is an automated tool that uses a meet-in-the-middle approach to find efficient or even optimal circuits (under some assumptions). However, for larger circuits, the search space of the circuits grows exponentially large and this requires completely different heuristics. One can cite for example synthesis tools such as Yosys [Wol] and ABC [BM10] that offers circuit optimisation for large domain size.

Generally, a cipher can be broken down into two distinct layers: the linear layer and the non-linear layer (often composed of the application of Sboxes or modular additions). The tools mentioned previously can handle both linear or non-linear layers and many papers in the literature have proposed new techniques to implement the non-linear layer efficiently. This is especially true in particular for the AES S-Box. In [Can05], Canright used a technique involving the subfield arithmetic that eventually led to a small and compact AES S-box circuit.

While the non-linear layer, which generally costs a substantial part of the total encryption circuit, has been well studied, a lot of attention has also been given to the case of linear circuits, which can be seen as a potentially simpler sub-case. As the linear layer is usually implemented with many (relatively costly) XOR gates, improvements on this part can lead to non-negligible savings on the final circuit.

Although the problem of finding optimal linear circuits has been shown to be NP-hard [BMP08, BMP13], there are existing heuristics that could give quite good approximations. Paar's algorithms [PR97], for instance, provide good estimations for matrices, even for large sizes. In recent years, a series of algorithms have been published solely focusing on reducing the number of XORs required for linear circuits. In 2017, [KLSW17] showed that by using the Boyar-Peralta (BP) algorithm [BP10, BMP13], many linear circuits can be more efficiently handled when the size of the matrix is not too large. Many research have built on the structure of the BP algorithm in order to further reduce the complexity of linear circuits [VSP18, RTA18, ME19]. As a benchmark to compare the various heuristics, a prominent circuit to study is obviously the AES diffusion matrix, whose XOR gate cost has been reduced down to 97 XORs after several successive improvements [KLSW17].

Our Contributions. In this paper, we provide new heuristics to improve the XOR gates count in linear circuits, which can be used to optimise the implementation of diffusion matrices or any linear function with or without full rank. Firstly, we show that the randomisation of the BP algorithm [BP10, BMP13] can lead to improvements over the state-of-the-art. One particularly interesting case is that by inducing randomisation, we can obtain an implementation of the AES MDS matrix requiring only 95 XORs, a new record. The benefits of randomisation became increasingly more significant as the size and density of the matrix increase.

Then, we also propose two new algorithms, so-called A1 and A2, that aim at improving the XOR count even further. The main idea of the algorithms is to attempt to reach some of the outputs first, provided they are not too costly and at the same time, trying to minimise the cost required for the rest of the outputs. As for all known heuristics except Paar's algorithm, the size of the linear layers that A1 and A2 can handle is limited (32×32 matrices of low density such as the AES matrix will usually take around 30 to 40 minutes for a single run on a small computation cluster). We have implemented these algorithms on a set of many random square matrices (of sizes 15×15 to 20×20 with the densities ranging from 0.1 to 0.9) which is a subset from the benchmark set used in [VSP18]. We compared the results obtained with various state-of-the-art algorithms such as Paar1, BP, and a randomised version of Shortest-Dist-First [RTA18]. A distinctive trend we observed is that when the density, ρ, of the matrix is small, $\rho \approx 0.1 - 0.2$, all the above-mentioned algorithms yielded about the same results (the problem is generally much easier to solve as the search space is very limited). When $\rho \approx 0.4 - 0.6$, we start to see that A1 and A2 algorithms are performing better. More importantly, as the size of the matrix increases, the criteria used in A1 and A2 provide even stronger gains over other existing heuristics, which indicates that A1 and A2 are indeed fundamentally improving over the state-of-the-art. In order to further improve the circuits' size, we have also incorporated Yosys synthesis tool [Wol] in the picture, as well as some simple local optimisation techniques we propose in order to reduce the XOR count. In some cases, we even managed to reduce the circuit

depth as well. The benchmark results are shown in Table 4 and Table 5. We have also implemented the algorithms on matrices from [DL18] and the results are shown in Table 6 and Table 7. All in all, even though these matrices came with an optimised circuit by construction, we surprisingly managed to improve 7 of them (out of 24).

Finally, as the ultimate benchmark, we have run both A1 and A2 algorithms on the AES diffusion matrix and it produces a circuit requiring only 94 XORs, again a new record. This circuit is shown in Appendix E.

Organisation of the paper. In Section 2, we provide a brief overview of the mathematical background required for the understanding of this paper. In Section 3, we explain with some small examples the various state-of-the-art heuristics that are used to find good linear circuits for linear matrices (Paar's algorithms, BP algorithm and its variants, as well as algorithms proposed by Masoleh, Taha and Ashmawy in [RTA18]). In Section 4, we will provide our heuristics, mainly A1 and A2 algorithms, as well as a randomised version of the BP algorithm, mainly to illustrate the importance of randomisation while finding optimised circuits. Also, we will provide a short description of the local optimisation techniques in order to further post-reduce the number of XOR gates, starting from a circuit provided by the various global optimisation algorithms. In Section 5, we will provide the results after comparing the heuristics. These results are mainly on two different groups of matrices - a subset of the benchmark set in [VSP18] (which is a set of random matrices with varying densities of size 15×15 to 20×20) and matrices of size 16×16 and 32×32 from [DL18]. Finally, we conclude in Section 6 and propose possible future research directions.

2 Preliminaries

A linear Boolean function is a circuit consisting n input signals $\{x_0, x_1, ..., x_{n-1}\}$ and m output signals $\{y_0, y_1, ..., y_{m-1}\}$. Following the notation in [BP10], the output signals are also called targets. A circuit that is constructed only by the XOR gates (which can be represented by $\oplus$), is considered as linear. To describe a linear circuit, we will be using a series of t_i variables to represent the intermediate values. In this article, we will not put any restrictions on the number of such values t_i, but studying optimised linear circuits with bounded memory is also a relevant research direction.

Generally, most cryptographic applications use matrices with elements from the finite field, $\mathbb{F}_{2^k}$ in their calculations. The finite field is defined by some irreducible polynomial of degree k with coefficients from $\mathbb{F}_2$. To do multiplication with a fixed element from $\mathbb{F}_{2^k}$, we can represent the fixed element as a $k \times k$ matrix, K, with elements from $\mathbb{F}_2$ and the input element to be multiplied with as a vector $x = (x_0, x_2, ..., x_{k-1})^T$ with each $x_i \in \mathbb{F}_2$ for $i \in \{0, .., k-1\}$. A simple matrix multiplication with modulo 2 operation can be used to obtain the desired result: $K \cdot x$ for the purpose of multiplication. Therefore, a matrix M, of size $n \times n$ with fixed elements from $\mathbb{F}_{2^k}$, can also be interpreted as a matrix of size $nk \times nk$ with elements from $\mathbb{F}_2$. Therefore, we will now discuss only matrices with elements from $\mathbb{F}_2$.

A naive implementation of matrix (blindly computing the XORs directly given in the matrix representation) is usually very inefficient as there will be a lot of repeated calculations. This is especially true as the density of a matrix increases. Currently, the possibility of finding a circuit with least XOR gates relies on the fact that we can reuse intermediate variables t_i that were calculated once, for other computation steps. This strategy also forms the basis for all the algorithms that we discuss in this paper.

In [BP10], Boyar and Peralta coined the term "cancellation-free programs" which refers to programs that do not "cancel" out common terms; all cancellation-free programs have circuits that follow the rule: if the computation $t_i = u \oplus w$ is in the circuit where $u = x_{u_1} + x_{u_2} + ... + x_{u_k}$ and $v = x_{v_1} + x_{v_2} + ... + x_{v_l}$, then $\{x_{u_1}...x_{u_k}\} \cap \{x_{v_1}...x_{v_l}\} = \emptyset$.

For instance, Paar's algorithm produces a cancellation-free program whereas BP algorithm does not. Currently, it seems that by considering non-cancellation-free circuits, the XOR count can be improved. One such case can be observed in the example matrix given in Section 3.

3 State-of-the-art Heuristics

3.1 Paar's Algorithm [PR97]

In [PR97], Paar proposed two global optimisation algorithms, we will refer to them as Paar1 and Paar2 (i.e. Algorithm 1 and its improvement respectively in [PR97]). Both Paar1 and Paar2 take a binary matrix as input and attempt to find a more efficient circuit than its naive implementation. In essence, Paar1 first searches and records the frequency for all possible pairing of the input variables, $x_i \oplus x_j$ for $i, j = 0, ..., n-1, i \neq j$ occurring in the matrix. The pair with the highest frequency will then be computed and placed back into the matrix as a new variable, e.g. t_0. In the next round, all possible occurrences of $u \oplus v$ where $u, v \in \{x_0, x_1, ..., x_{n-1}, t_0\}$ will be considered. This process will repeat until all possible pairs of variables occur at most once in the matrix. Then, the remaining gates will just be computed naively. In Table 1, we take the example of a matrix M and apply Paar1 algorithm on it.

Paar2 is an improvement over Paar1, as when it considers all possible pairs with the highest frequency, it will then conduct a tree search for all such pairs. We note that this exhaustive search may not work very well as the dimension of the matrices increases.

Note that Paar1 and Paar2 produce cancellation-free circuits. However, the advantage of Paar1 is in its time complexity: it can be easily implemented for large matrices and obtains a circuit with a very reasonable computational effort.

3.2 BP algorithm [BP10]

The BP algorithm adopted a rather different approach to the problem. While the algorithm is much slower than Paar's algorithms and inapplicable for matrices of large size due to the exhaustive search nature for the calculation of the distance vector, it generally produces more efficient circuits. Notably, in [KLSW17], the BP algorithm managed to reduce the AES diffusion matrix circuit to 97 XORs. The general idea of BP is to reach a common path among as many targets as possible. Thus, the criteria for next gate in each step is to choose a gate that reduces the distance[1] to the highest number of targets. We describe here this heuristic:

1. Place all variables $\{x_0, x_1, ..., x_{n-1}\}$ into a set, *Base*. The rows in the matrix are denoted as set of targets $\{y_0, y_1, ..., y_{m-1}\}$ that we want to compute
2. Initialise an m-integer vector *Dist* which keeps track of the distances of each target from *Base*
3. Repeat until all the targets are added into the *Base*:
    (a) perform XOR on every unique pair of elements in *Base* and re-evaluate *Dist* vector
    (b) Select the pair candidate that minimises $\sum_{i=0}^{m-1} Dist[i]$ and add it into *Base*
    (c) In case of tie, choose the pair candidate that maximises the Euclidean norm of *Dist* (Norm criterion)

[1] The distance of a target is defined as the minimal number of XOR operations required to perform on a subset of elements in *Base* to reach the target

Table 1: Computation sequence when applying Paar1 algorithm to matrix M. x_i refers to the input signals $\forall i \in \{0, ..., 13\}$, y_j refers to the targets $\forall j \in \{0, ..., 6\}$ and t_k refers to the temporary intermediate gates $\forall k \in \mathbb{N}$

Iter.	**Pair with highest freq.**	**Freq.**
1	$t_0 = x_4 + x_{13}$	4
2	$t_1 = x_8 + x_9$	4
3	$t_2 = x_{10} + t_1$	4
4	$t_3 = x_3 + t_0$	3
5	$t_4 = x_5 + x_6$	3
6	$t_5 = t_2 + t_4$	3
7	$t_6 = x_1 + x_2$	2
8	$t_7 = x_7 + t_5$	2
-	$t_8 = x_0 + x_{13}$	1
-	$y_0 = t_6 + t_8$	1

Iter.	**Pair with highest freq.**	**Freq.**
-	$y_1 = t_3 + t_6$	1
-	$t_{11} = x_2 + x_5$	1
-	$y_2 = t_3 + t_{11}$	1
-	$y_3 = t_3 + t_5$	1
-	$y_4 = t_0 + t_7$	1
-	$y_5 = x_{13} + t_7$	1
-	$t_{16} = x_7 + x_{11}$	1
-	$t_{17} = x_{12} + x_{13}$	1
-	$t_{18} = t_2 + t_{16}$	1
-	$y_6 = t_{17} + t_{18}$	1

$$M = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Remark 1. In [BP10], the authors also mentioned that if there is any target with a distance value of 1 at some point (i.e. there exists two distinct *Base* elements, B_i and B_j such that the target y_k is equal to $y_k = B_i \oplus B_j$), then we can directly choose this gate to be added into *Base* as it will not increase the size of the resulting circuit (one gate will have to be spent to reach y_k anyway, so using $B_i \oplus B_j$ is necessarily among the best possible choices).

In Table 2, we provide the circuit produced by the BP algorithm when using the same input matrix M that was used for Table 1. This circuit is slightly shorter as compared to the one in Table 1 as it allows for cancellation of variables. This can be observed by looking at the construction for the intermediate variable t_{17} ($t_{17} = t_4 \oplus y_5$, where $y_5 = x_3 \oplus (x_7 \oplus (t_5 \oplus t_4)))$ and thus, a cancellation of the value t_4). This led to a reduction of the number of gates required to obtain y_6.

In the rest of the article, we will say that a target is far (resp. near) if the corresponding value in the *Dist* vector is large (resp. small) relative to the other values in *Dist*.

3.3 BP algorithm for dense matrices [VSP18]

Visconti, Schiavo and Peralta proposed in [VSP18] a new heuristic for matrices that are dense. The gist of the heuristic is to approach the Boolean complement of the matrix that it wants to compute first. It will attempt to find the all-one vector along with the targets in the complementary matrix. Eventually, an additional XOR is used between the all-one vector and each of the complementary target to get the original target. They applied their strategy on a benchmark set, which is a set consisting of random matrices of different sizes

Table 2: Computation sequence when applying BP algorithm to matrix M. B refers to the initialized base elements, x_i refers to the input signals $\forall i \in \{0, ..., 13\}$, y_j refers to the targets $\forall j \in \{0, ..., 6\}$ and t_k refers to the temporary intermediate gates $\forall k \in \mathbb{N}$

Iter.	New base element	New ***Dist*** vector	Iter.	New base element	New ***Dist*** vector
0	$B = \{x_0, ..., x_{13}\}$	[3 4 4 7 7 6 6]	10	$y_2 = x_5 + t_7$	[3 0 0 0 2 2 4]
1	$t_0 = x_4 + x_{13}$	[3 3 3 6 6 6 6]	11	$t_{10} = x_7 + t_5$	[3 0 0 0 1 1 4]
2	$t_1 = x_8 + x_9$	[3 3 3 5 5 5 5]	12	$y_4 = t_0 + t_{10}$	[3 0 0 0 0 1 4]
3	$t_2 = x_{10} + t_1$	[3 3 3 4 4 4 4]	13	$y_5 = x_{13} + t_{10}$	[3 0 0 0 0 0 4]
4	$t_3 = x_3 + t_0$	[3 2 2 3 4 4 4]	14	$t_{13} = x_0 + x_1$	[2 0 0 0 0 0 3]
5	$t_4 = x_5 + x_6$	[3 2 2 2 3 3 4]	15	$t_{14} = x_2 + x_{13}$	[1 0 0 0 0 0 3]
6	$t_5 = t_2 + t_4$	[3 2 2 1 2 2 4]	16	$y_0 = t_{13} + t_{14}$	[0 0 0 0 0 0 3]
7	$y_3 = t_3 + t_5$	[3 2 2 0 2 2 4]	17	$t_{16} = x_{11} + x_{12}$	[0 0 0 0 0 0 2]
8	$t_7 = x_2 + t_3$	[3 1 1 0 2 2 4]	18	$t_{17} = t_4 + y_5$	[0 0 0 0 0 0 1]
9	$y_1 = x_1 + t_7$	[3 0 1 0 2 2 4]	19	$y_6 = t_{16} + t_{17}$	[0 0 0 0 0 0 0]

and density. Their results indicate that this new heuristic will be well suited for matrices that have a high density.

3.4 Masoleh, Taha and Ashmawy's algorithms [RTA18]

In [RTA18], Masoleh, Taha and Ashmawy have proposed three alternatives to the BP algorithm: Improved-BP, Shortest-Dist-First and Focused-Search.

In Improved-BP, they start from the BP algorithm, but used a different tie-breaker. If there is a tie in one iteration of BP algorithm, instead of relying on the Euclidean-norm of the distances, the Improved-BP algorithm performs a local exhaustive search by testing all the best results in the next iteration to determine the best pair.

In Shortest-Dist-First (SDF), the algorithm selects the best pair by choosing a gate that maximises the number of pre-emptive gates (pre-emptive gates refer to targets that are at distance one away, thus, gates that we can directly add into the *Base* according to Remark 1). In the case that none can be found, it will look for a gate that maximises the number of targets at $Dist[i] = 2$ and so on; i.e. they try to select a gate such that it reduces the distance of as many "nearest" targets as possible.

In Focused-Search, the algorithm combines the criteria in SDF and the tie-breaking method from their Improved-BP algorithm. In [RTA18], they have implemented on some matrices and showed that this algorithm outperformed the other heuristics (BP, Improved-BP, SDF). However, we would like to emphasize that the matrices that the authors have tested on are very small: 20×8, 8×10 and 8×8.

4 New Heuristics

In this section, we propose new algorithms for finding optimised linear circuits. Our implementations can be downloaded from the following address: https://github.com/thomaspeyrin/XORreduce.

4.1 Random-Normal-BP (RNBP)

In [BP10], Boyar and Peralta have experimented on several tie-breaking techniques for their algorithm. Instead of having the *Norm* as the tie-breaker, they have also tried *Random*. Under *Random*, it considers the sum of the distance to targets and whenever there is a possible new base vector with the same sum with the current selection, with probability $\frac{1}{2}$, it will apply the *Norm* criterion to determine if we should swap or remain with the selected choice. As this results in a non-deterministic algorithm, the algorithm is repeated for a total of three times and the best circuit is then selected. However, this results in an uneven spread of probabilities among all the "equally good" candidates for the next gate according to the criteria. For instance, if there are 4 equally good candidates, the first, second, third and fourth candidates will have probabilities of $\frac{1}{8}$, $\frac{1}{8}$, $\frac{1}{4}$ and $\frac{1}{2}$ of being chosen respectively.

We propose first a simple variation of the BP algorithm: *Randomised-Normal-BP* (RNBP)[2]. After several runs of the BP algorithm, we noticed a high number of ties even after the *Norm* tie-breaker has been used. Since those possible new bases passed the criteria with the *Norm* tie-breaker, we treat these bases to be equally good. Unlike the "Random" used in [BP10], we decided to add a simple randomisation process to choose the next base, giving each possible candidate after the *Norm* tie-breaker an equal probability of being selected. This algorithm shows the effectiveness of a randomised algorithm. The results obtained experimentally show that by randomising the choices and using the algorithm multiple times, we often produce a circuit that has a lower XOR count than itself without randomisation. This produces a better measurement of how good the criterion is as it transcends the order of computation of all possible choices. In the case of AES, RNBP is able to yield a 95 XOR circuit for the MDS matrix, beating the current record by 2 XORs [KLSW17]. In Section 5, we can see that RNBP is faring better than BP.

For reference, an algorithmic description of our RNBP strategy can be found in Appendix A.1.

4.2 A1 & A2 algorithms

In this section, we propose two new non-deterministic algorithms, called A1 and A2, that are also based on the general BP strategy. Similarly to the Shortest-Dist-First (SDF) algorithm [RTA18], our rationale is to aim at reaching the targets that are the nearest (targets with the minimum *Dist* value that is non-zero). However, the main difference with SDF is that A1 and A2 are less restrictive: we do not look for a new base element that can reduce the distance of as many nearest targets as possible, instead, we just choose a gate that can minimise at least one of the nearest targets. We called this step the filtering step. We want to have this filtration to filter out gates that only minimise targets that are further away. After the filtration, we will use the criteria with *Norm* tie-breaker in BP subsequently to rank the gates that pass the filter in order to reach a common path among all the targets. For example, suppose if at step k we have a $Dist = [3\ 4\ 5\ 6\ 7]$ and we have three possible gates t_{k_a}, t_{k_b} and t_{k_c} that will result in the following changes to *Dist*:

$$t_{k_a} : Dist = [2\ 3\ 5\ 6\ 7]$$
$$t_{k_b} : Dist = [2\ 4\ 5\ 6\ 7]$$
$$t_{k_c} : Dist = [3\ 3\ 4\ 5\ 6]$$

respectively, then t_{k_a} is favoured over t_{k_b} and t_{k_c} as it reduces the nearest target, and between the t_{k_a} and t_{k_b}, it reduces the distance to the highest number of targets.

[2]Following the terminology from [RTA18], Normal-BP as the BP algorithm with the "Norm" tie-breaking is used, we added a "Randomised" process.

Let us provide our motivation for trying this strategy. In general, targets that are further away tend to contribute more to the reduction of the sum of distance due to their high Hamming Weight and thus, they have a higher probability to contain similar terms with other high Hamming Weight targets. This causes the BP's criteria of using the sum of distances to favour a move that reduces the distance to targets with high Hamming Weight. Hence, the common path is constructed to reach the higher Hamming Weight targets while not paying much attention to the ones with lower Hamming Weight. However, we want the targets that are the nearest to be taken into account as well when computing the common path. Thus, by doing this arrangement of criteria, we can avoid situations where targets with large Hamming Weight are dominating the common path.

For algorithm A1, the selection for the next gate is performed as follows:

1. **Filtering.** It must reduce the distance of at least one of the nearest targets
2. **Selection.** Among the ones that pass the filter, select the one which minimises the sum of distances
3. **Tie-breaker.** The Euclidean Norm will be used as a tie-breaker for the above criteria
4. **Randomisation.** If the tie-breaker does not resolve the tie, with equal probability, we will randomly pick one of the remaining candidates after the tie-breaking process

In algorithm A2, we skip step 3 (tie-breaking) from algorithm A1. This provides more randomisation when compared to algorithm A1.

An illustration using the same matrix example as in Table 1 is shown in Table 3. This example is crafted to illustrate the strength of A1 and A2 as compared to the BP algorithm. The matrix M is crafted in a way that the targets share many common input signals. Ideally, we would like to approach the one that is the nearest among the set of all targets (that is, y_1 in the case of M) and approach the rest of the targets by making those small gaps. The differences in selecting their next new base element can be seen from the very first iteration for BP and A1 in Table 2 and Table 3 respectively. The former tends to be greedier (in terms of distance reduction) in choosing the gate with inputs $\{x_4, x_{13}\}$ which reduces the distance vector by 4. However, the targets that gained a reduction are the ones with the highest Hamming Weights too. The latter chose to reach the nearest target first (while keeping the overall distance in check) which resulted in the choice of the gate with inputs $\{x_2, x_{13}\}$. We can see that for this particular matrix M, A1's approach managed to achieve a smaller circuit.

For reference, an algorithmic description of our A1 and A2 heuristics can be found in Appendix A.2 and A.3.

4.3 Local Optimisation

We describe in this section some local optimisation techniques (LocalOpt) aiming at reducing XOR count further after a circuit of the matrix has been formed. Unlike BP algorithm where it finds the common path by choosing a gate that minimises the sum of distances, A1 and A2 (and also SDF) choose the one that is the nearest first, which often creates a 'walk' around the targets instead of many branches from the common path. Thus, many gates are only used once. This enables us to do some swapping operations with the resulting circuit from global optimisation without involving too many variables. After some swapping, there may be some repeated gates. Thus, by clearing these repeated gates, we are able to reduce the XOR count further.

While our aim is to reduce the XOR count, the depth might also be improved by LocalOpt for some circuits. Since the time taken to do the LocalOpt is insignificant when

Table 3: Computations sequence when applying A1 algorithm to matrix M. B refers to the initialized base elements, x_i refers to the input signals $\forall i \in \{0, ..., 13\}$, y_j refers to the targets $\forall j \in \{0, ..., 6\}$ and t_k refers to the temporary intermediate gates $\forall k \in \mathbb{N}$

Iter.	**New base element**	**New dist.**	**Iter.**	**New base element**	**New dist.**
0	$B = \{x_0, ..., x_{13}\}$	[$\underline{3}$ 4 4 7 7 6 6]	10	$t_9 = x_8 + t_8$	[0 0 0 $\underline{2}$ 4 3 3]
1	$t_0 = x_2 + x_{13}$	[$\underline{2}$ 3 3 7 7 6 6]	11	$t_{10} = x_6 + t_9$	[0 0 0 $\underline{1}$ 3 2 3]
2	$t_1 = x_1 + t_0$	[$\underline{1}$ 2 3 7 7 6 6]	12	$y_3 = t_5 + t_{10}$	[0 0 0 0 $\underline{2}$ 2 3]
3	$y_0 = x_0 + t_1$	[0 $\underline{2}$ 3 7 7 6 6]	13	$t_{12} = x_7 + y_3$	[0 0 0 0 $\underline{1}$ 1 3]
4	$t_3 = x_3 + x_4$	[0$\underline{1}$ 2 6 7 6 6]	14	$y_4 = x_3 + t_{12}$	[0 0 0 0 0 $\underline{1}$ 3]
5	$y_1 = t_1 + t_3$	[0 0 $\underline{2}$ 6 7 6 6]	15	$y_5 = t_3 + t_{12}$	[0 0 0 0 0 0 $\underline{3}$]
6	$t_5 = x_5 + t_3$	[0 0 $\underline{1}$ 5 7 6 6]	16	$t_{15} = x_{11} + t_9$	[0 0 0 0 0 0 $\underline{2}$]
7	$y_2 = t_0 + t_5$	[0 0 0 $\underline{5}$ 7 6 6]	17	$t_{16} = x_{12} + t_{15}$	[0 0 0 0 0 0 $\underline{1}$]
8	$t_7 = x_9 + x_{13}$	[0 0 0 $\underline{4}$ 6 5 5]	18	$y_6 = x_7 + t_{16}$	[0 0 0 0 0 0 0]
9	$t_8 = x_{10} + t_7$	[0 0 0 $\underline{3}$ 5 4 4]			

compared to that of the global optimisation algorithms, we propose that LocalOpt is to be used concurrently with the Yosys synthesis tool [Wol] on the circuits produced by the global optimisation algorithms, in order to maximise the potential for savings. Since Yosys tool is also very fast as compared to the global optimisation, we can actually implement Yosys first then LocalOpt and vice versa and take the best circuit among the two.

We have two LocalOpt techniques: swapping orders to rearrange parts of a circuit and a local exhaustive search to try out all the various permutations and choose the tree that has the most savings.

4.3.1 Swapping orders

This method helps to identify and rearrange special parts of the circuit such that some repeated gates may appear. This rearrangement may also cause the depth of some gates and possibly the depth of the entire circuit to be reduced.

We first identify gates that are used only once for the computation of a value, then we can perform the following operation. Suppose that a part of the circuit is composed of:

$$\begin{gathered} \vdots \\ a = b \oplus c \\ d = a \oplus e \\ f = b \oplus e \\ \vdots \end{gathered}$$

where a is only used in the evaluation of d (i.e. a is only used once). Let $depth(k)$ represent the depth of a value k in the circuit. If $depth(e) < max\{depth(b), depth(c)\}$, then we may

rewrite the above gates sequence as follows:

$$\begin{aligned} &\vdots \\ a &= e \oplus argmin\{depth(b), depth(c)\} \\ d &= a \oplus argmax\{depth(b), depth(c)\} \\ f &= b \oplus e \\ &\vdots \end{aligned}$$

If $depth(b) < depth(c)$, then the depth of both a and d will be reduced. Therefore, the other parts of the circuits may see a depth reduction as well. We can observe that this helps to save an XOR gate as well: now the gate a and f are doing the exact same operation. Thus, we will check the depth of the new a and f and eliminate the one that has a larger delay and swap the orders if necessary to maintain the order of the circuit.

4.3.2 Local exhaustive search

As a full exhaustive search is infeasible due to the large search space, we propose to perform local exhaustive searches to identify possible opportunities to reduce the number of XOR gates. To describe a circuit, we can use a tree representation for it. Each gate can be represented as a binary tree with three nodes. The parent (and in this case, the root) of the tree will be the output value and the two child nodes are the input values. Starting from any intermediate or target values as the root of the binary tree, we can form the binary tree recursively until all the leaf nodes are the input variables.

For each intermediate and target value in the circuit, we extract the sub-part of the circuit (a binary tree) that leads to this value. We permute to obtain all the possible configurations of binary trees with the same root and leaf nodes. Then, we check all possible configurations to see if there is any tree that could reduce the number of gates required as compared to the one in the circuit. If there is such a tree, we will do a substitution.

For example, if part of the circuit is

$$\begin{aligned} &\vdots \\ t_1 &= x_0 \oplus x_1 \\ t_2 &= x_0 \oplus x_2 \\ t_3 &= x_2 \oplus t_1 \\ t_4 &= x_3 \oplus t_2 \\ &\vdots \end{aligned}$$

then all the possible computation trees with gate t_3 as the root are shown in Figure 1. The tree on the left is the original circuit whereas the others are the possible alternatives that do the exact same computations. Assuming t_1 is only used in the computation of gate t_3, then the middle tree in Figure 1 will be the most efficient since t_2 has already been computed in the circuit and we can remove t_1 from it. Although the tree on the right in Figure 1 does not need t_1 (and therefore, we are able to remove t_1 too), at the same time it creates an additional node t_k, thus we will not have any savings overall. Due to the large permutation tree that we may encounter, we limit the number of leaf nodes of each tree to be 5. That is, start from the root, we start looking for the child nodes for each level recursively until we have at most 5 leaf nodes, regardless whether we have yet reached the input values or not.

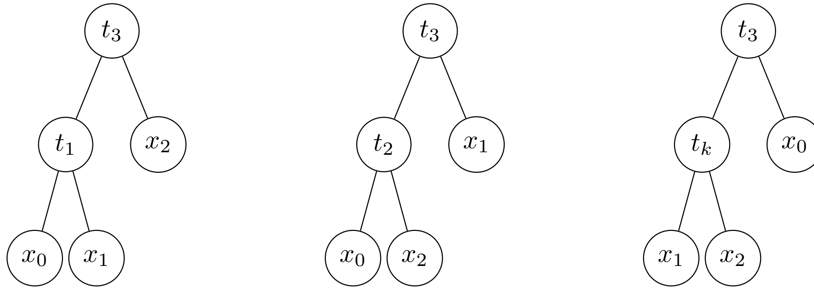


Figure 1: Computation trees of gate t_3

5 Results

Heuristics used for comparison. In this section, we will compare our results against various state-of-the-art algorithms. This includes the BP algorithm, as well as a randomised version of the Paar1 (RPaar1) and the SDF algorithm (RSDF). Similar to the random procedure added in RNBP, whenever we have a tie, a gate will be randomly selected. We did not implement the Focused-Search algorithm as it is extremely time-consuming for matrices that are larger. Since the search space of Focused-Search and SDF algorithms are a subset of that of RSDF, implementing RSDF may be a better comparison of how well the criteria actually performs in a randomised environment.

Implementation. All of the implementations (except RPaar1 and Paar1) are written in C++. As for RPaar1 and Paar1, they are written in Python. The reason for choosing a different language is that RPaar1 and Paar1 are extremely fast, thus we do not require the efficiency from writing in C++. Furthermore, we have no intention to compare the speed of these two algorithms with the other ones. Thus, we choose Python to have a quick and accurate implementation. For the BP algorithm, we use the implementation that is provided in the GitHub repository stated in [KLSW17]. As for RNBP, SDF, A1 and A2, we implemented them with several changes from the code for BP. All implementations were benchmarked on AMD EPYC 7401 24-Core Processor model with 96 CPUs and 60GB of RAM. The implementations were run on a cluster of 90 cores, with one core per matrix.

Test set. We have tested the algorithms on a smaller scale of the benchmark set used in [VSP18]. The benchmark set consists of random square matrices categorised by their size and density. For our test set, we have chosen a smaller subset: we have chosen 10 matrices from each size (ranging from 15×15 to 20×20) and density, ρ (ranging from 0.1 to 0.9). In total, we have 540 matrices for our benchmark subset. We have also tested the algorithms on the 16×16 and 32×32 matrices from [DL18].

Computational effort. Due to the non-deterministic nature of our algorithms, we run the matrices with different algorithms with a fixed time based on the size and density. We chose the amount of computational effort such that at least nine out of ten matrices in each category (size and density) will reach a *stable* state for all the algorithms implemented on it. We define *stable* state (or a result is stabilised) as a situation where the algorithm did not yield any improvements in terms of the number of XOR count for the last half of the time allocated. For instance, the ten matrices with size 15×15 with $\rho = 0.5$ were allocated a computational effort of 2000 seconds. This means that for at least nine of the matrices, no further improvement is observed in the last 1000 seconds for each algorithm. The circuit with the lowest XOR count yielded by each algorithm is then used for comparison.

LocalOpt & Yosys. We further optimised the circuits using Yosys synthesis tool and LocalOpt in the two possible orders (Yosys first, then LocalOpt, and vice versa). The circuit (with the lowest XOR count) among the two is then taken as the best circuit. We will term the circuit before this optimisation step as BLOY and the circuit after this step as ALOY.

5.1 Benchmark-subset

XOR count. The general trend of the average XOR count difference of the best ALOY circuits found by the different algorithms can be seen in Figures 2, 3, 4 and 5. The time allowed for each size/density can be found in Table 9 of Appendix B. In Table 4, the percentage of the smallest circuit yielded per size is shown. The ones in round bracket are the percentages for BLOY circuits.

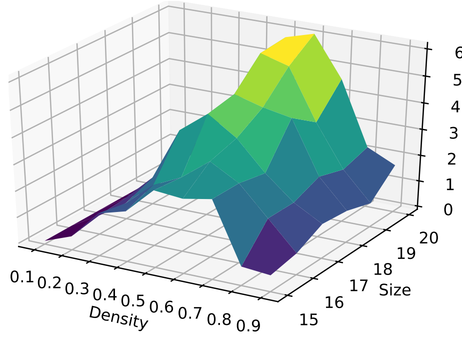


Figure 2: Average XOR count difference of the best ALOY circuits found by BP and A1

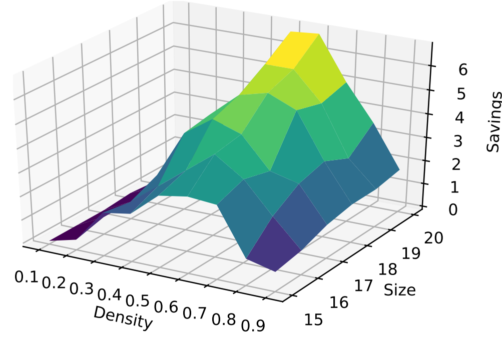


Figure 3: Average XOR count difference of the best ALOY circuits found by BP and A2

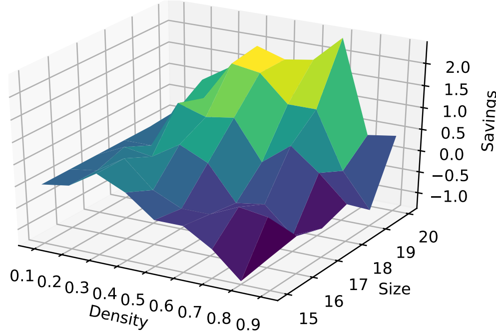


Figure 4: Average XOR count difference of the best ALOY circuits found by RSDF and A1

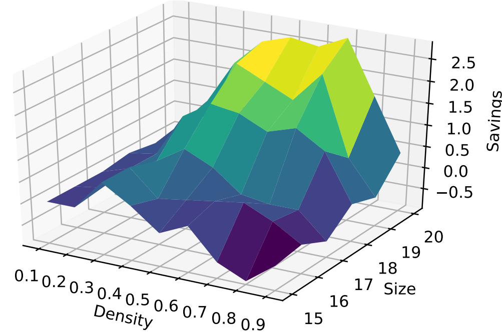


Figure 5: Average XOR count difference of the best ALOY circuits found by RSDF and A2

When the sizes of the matrices are small (i.e. 15×15), and if the density is small (0.1 to 0.2), all tested algorithms performed about the same. This is due to the fact that these problems have low branching, and thus low complexity. When the density is about 0.3 to 0.6, A1 and A2 performed better than the rest and when density is around 0.8 to 0.9, RSDF performed better as compared to the rest.

As the sizes of the matrices get larger, the distinction between each algorithm is clear. For size 20×20, when the density is small (0.1 to 0.2), all algorithms performed about the

same. But when the density is around 0.3 to 0.9, algorithms A1 and A2 performed the best among all algorithms, producing circuits with the lowest XOR count.

Table 4: Percentage of best circuits (XOR counts) produced by various algorithms for various square matrix sizes. The brackets contain the percentage of the BLOY circuits. The best percentages for both ALOY and BLOY circuits are highlighted in bold

Matrix Size	BP [BP10]	Paar1 [PR97]	RPaar1	SDF [RTA18]	RNBP Sec. 4.1	A1 Sec. 4.2	A2 Sec. 4.2
15	25.56 (20.00)	14.44 (14.44)	14.44 (14.44)	**70.00** **(66.67)**	38.89 (33.33)	58.89 (56.67)	66.67 **(66.67)**
16	21.11 (18.89)	8.89 (8.89)	10.00 (10.00)	61.11 (65.56)	28.89 (32.22)	53.33 (54.44)	**73.33** **(77.78)**
17	17.78 (16.67)	11.11 (11.11)	11.11 (11.11)	62.22 (58.89)	26.67 (26.67)	53.33 (52.22)	**72.22** **(74.44)**
18	15.56 (13.33)	8.89 (8.89)	11.11 (11.11)	41.11 (36.66)	31.11 (25.56)	52.22 (52.22)	**85.56** **(83.33)**
19	14.44 (13.33)	11.11 (11.11)	11.11 (11.11)	32.22 (31.11)	26.67 (27.78)	54.44 (48.89)	**74.44** **(80.00)**
20	12.22 (12.22)	11.11 (10.00)	11.11 (11.11)	25.56 (23.33)	23.33 (23.33)	58.89 (58.89)	**87.78** **(90.00)**

Depth Although the depths of the circuits generated by algorithms A1 and A2 are not as low as the BP algorithm, the depth of the circuits are generally better than that of the RSDF algorithm. Table 5 shows the percentage of best ALOY circuits (depth) produced by the various algorithms. Again, the ones in round bracket are the percentages for BLOY circuits. Note: we acknowledge that Paar1 and RandomPaar1 produce circuits with good depth levels (Paar1 yielded $>90\%$ of the best depth for each size). However, the results for the two algorithms are omitted from Table 5 for the other algorithms to compare.

Table 5: Percentage of best ALOY circuits (Depth) produced by various algorithms for various square matrix sizes. The brackets contain the percentage of the BLOY circuits. The best percentages for both ALOY and BLOY circuits are highlighted in bold

Matrix Size	BP [BP10]	RSDF [RTA18]	RNBP Sec. 4.1	A1 Sec. 4.2	A2 Sec. 4.2
15	**78.89 (70.00)**	25.56 (16.67)	52.22 (34.44)	36.67 (14.44)	30.00 (18.89)
16	**85.56 (74.44)**	21.11 (16.67)	55.56 (31.11)	26.67 (18.89)	30.00 (26.67)
17	**82.22 (80.00)**	20.00 (14.44)	51.11 (36.67)	28.89 (15.56)	30.00 (23.33)
18	**85.56 (76.67)**	27.78 (17.78)	56.67 (41.11)	24.44 (17.78)	30.00 (16.67)
19	**77.78 (74.44)**	23.33 (11.11)	47.78 (34.44)	26.67 (11.11)	35.56 (14.44)
20	**84.44 (73.33)**	17.78 (12.22)	48.89 (25.56)	25.56 (10.00)	28.89 (15.56)

Time to stability. On average, RSDF took the longest time to reach stability, followed by A2, A1 and then RNBP. Taking RNBP time as the base, RSDF, A2 and A1 required approximately 6.8, 3.2 and 1.7 more time in order to reach convergence. Some of the graphs

for the convergence to stability are shown in Appendix C. The average time taken for the best circuit to form in each category is also shown in Appendix D. The time complexity for all the algorithms is dominated by the computation of *Dist* vector. Roughly speaking, it is positively related to the Hamming weight of the targets as well as the number of elements in *Base*. Since RSDF (also, A1 and A2), attempts to reach the nearest targets first, the Hamming Weight of the further targets remains large as the size of the *Base* increases, causing the time required for most of the runs to be longer than that of BP.

5.2 Matrices from [DL18]

We applied the algorithms on the matrices from [DL18]. We compared the results across using RNBP, RSDF, A1 and A2 with the columns "Ours", "Yosys", "BP" and "Paar2" from Table 5 of [DL18]. The results are summarised in Table 6 and Table 7 for 16×16 and 32×32 matrices respectively. The columns "Yosys" and "Ours" can be found in the column "Const." as the first and second entry respectively as Duval and Leurent constructed the matrices by choosing optimal paths in order to minimise the number of XOR required. After they have constructed such a circuit, they implemented Yosys synthesis tool [Wol] to further optimise it.

Computational effort. For 16×16 matrices, we allowed 15000 seconds for it to run. Within the time limit, all twelve matrices have reached stability. For 32×32 matrices, we allowed a total of 432000 seconds (5 days) of running time, however, not all of them reached a stable state.

Global optimisation comparison. Comparing across various global optimisation algorithms (RNBP, A1, A2, RSDF, BP and Paar2), the number of best (BLOY) circuits (in terms of XOR count) achieved for these algorithms are 14, 5, 18, 2, 4 and 0 out of 24 matrices respectively.

Overall improvements. Despite the fact that these matrices are constructed specially for their optimised implementation, we are still able to improve 7 out of the 24 matrices. For the remaining 17 matrices, we have also achieved good results: 9 matrices with the same number of XOR gates as the construction and 8 matrices with a difference of at most 3 XOR.

5.3 The AES diffusion matrix

We have also applied the various algorithms on the AES MDS matrix. The algorithms were given a total of five days to run. As the AES InvMixColumn has a high density, we are unable to run it directly, thus, we used a decomposition from [DR02] and ran the algorithms on the two decomposed matrices. The results can be seen in Table 8. As compared to the previous AES record of 97 XORs [KLSW17], algorithm A2 has produced another even smaller circuit with 94 XORs. With a longer running time, A1 has also achieved a circuit with 94 XOR. A verification has shown that the circuits with 94 XOR yielded by A1 and A2 are not in the search space of RNBP.

Recent Works Two independent results [BFI19, Max19] have been published in IWSEC 2019 and ePrint after our submission to TCHES, in which the authors managed to achieve similar results for AES MixColumn matrix. More precisely, [BFI19] uses a method to avoid the problem of selecting only the first candidate lexicographically in the loop in the code given by [KLSW17]: they randomly generated two permutation matrices P and Q which help to shuffle the rows and columns for the targeted matrix M. The experiment is

Table 6: Circuit cost (XOR/Depth) of matrices of size 16 × 16 from [DL18] using several optimisation tools

Matrix	**Instantiation** (α, β, γ)	**Const.** [BP10]	**BP** [PR97]	**Paar2** [RTA18]	**RSDF** [DL18]	**RNBP** Sec. 4.1	**A1** Sec. 4.2	**A2** Sec. 4.2
$M^{9,3}_{4,5}$	$(A_4, -, -)$	$35/5^{*\frown}$ $39/5^{\frown}$	$38/7^{\frown}$	$45/5^{\frown}$	$\mathbf{36/7}^{\dagger}$ 38/8	$37/5^{\dagger}$ 38/7	$39/7^{\dagger}$ 39/9	$37/7^{\dagger}$ 38/8
$M^{9,3}_{4,5}$	$(A_4^{-1}, -, -)$	$36/5^{*\frown}$ $39/5^{\frown}$	$40/4^{\frown}$	$46/4^{\frown}$	$38/6^{\dagger}$ 39/9	$39/5^{\dagger}$ 39/7	$38/5^{\dagger}$ 38/6	$\mathbf{35/6}^{\dagger}$ 36/6
$M^{8,3}_{4,6}$	$(A_4, -, -)$	$35/5^{*\frown}$ $35/5^{\frown}$	$38/7^{\frown}$	$45/5^{\frown}$	$\mathbf{37/7}^{\dagger}$ 39/8	$38/7^{\dagger}$ 38/8	$39/7^{\dagger}$ 39/10	$38/5^{\dagger}$ 38/6
$M^{8,3}_{4,6}$	$(A_4^{-1}, -, -)$	$35/5^{*\frown}$ $35/6^{\frown}$	$40/4^{\frown}$	$46/4^{\frown}$	$36/8^{\dagger}$ 37/9	$38/7^{\dagger}$ 39/7	$38/5^{\dagger}$ 38/5	$\mathbf{35/6}^{\dagger}$ 36/6
$M^{8,3}_{4,5}$	$(A_4^{-1}, A_4, A_4^{-2})$	$36/6^{*\frown}$ $36/6^{\frown}$	$40/6^{\frown}$	$47/4^{\frown}$	$40/10^{\dagger}$ 40/17	$39/7^{\dagger}$ 39/7	$\mathbf{38/7}^{\dagger}$ 39/11	$\mathbf{38/7}^{\dagger}$ 39/9
$M^{9,4}_{4,4}$	$(A_4, -, -)$	$39/4^{*\frown}$ $40/4^{\frown}$	$41/9^{\frown}$	$47/5^{\frown}$	$41/10^{\dagger}$ 42/14	$40/7^{\dagger}$ 40/10	$\mathbf{39/6}^{\dagger}$ **39/9**	$\mathbf{39/6}^{\dagger}$ **39/9**
$M^{9,3}_{4,4}$	$(A_4^{-1}, A_4, A_4^{-2})$	$40/4^{*\frown}$ $40/4^{\frown}$	$40/7^{\frown}$	$43/4^{\frown}$	$40/8^{\dagger}$ 40/10	$\mathbf{39/6}^{\dagger}$ **39/7**	$41/8^{\dagger}$ 42/10	$41/7^{\dagger}$ 41/7
$M^{8,4}_{4,4}$	$(A_4, -, -)$	$38/4^{*\frown}$ $38/4^{\frown}$	$40/7^{\frown}$	$43/5^{\frown}$	$41/7^{\dagger}$ 41/10	$\mathbf{39/6}^{\dagger}$ **39/8**	$40/5^{\dagger}$ 40/7	$\mathbf{39/5}^{\dagger}$ 40/7
$M^{8,4'}_{4,4}$	$(A_4, -, -)$	$38/4^{*\frown}$ $38/4^{\frown}$	$43/6^{\frown}$	$41/4^{\frown}$	$\mathbf{38/6}^{\dagger}$ 40/6	$41/5^{\dagger}$ 42/6	$39/6^{\dagger}$ 41/7	$\mathbf{38/4}^{\dagger}$ 40/6
$M^{8,4''}_{4,4}$	$(A_4, -, -)$	$37/4^{*\frown}$ $37/4^{\frown}$	$40/5^{\frown}$	$43/5^{\frown}$	$40/7^{\dagger}$ 41/12	$40/5^{\dagger}$ 40/6	$40/5^{\dagger}$ 40/6	$\mathbf{39/5}^{\dagger}$ **39/7**
$M^{9,5}_{4,3}$	$(A_4, -, -)$	$41/3^{*\frown}$ $41/3^{\frown}$	$\mathbf{40/4}^{\frown}$	$43/4^{\frown}$	$41/5^{\dagger}$ 43/7	$\mathbf{40/4}^{\dagger}$ **40/5**	$41/6^{\dagger}$ 41/7	$\mathbf{40/4}^{\dagger}$ **40/5**
$M^{9,5}_{4,3}$	$(A_4^{-1}, -, -)$	$41/3^{*\frown}$ $41/3^{\frown}$	$43/5^{\frown}$	$44/3^{\frown}$	$44/7^{\dagger}$ 44/10	$41/5^{\dagger}$ 41/6	$41/5^{\dagger}$ 41/7	$\mathbf{40/4}^{\dagger}$ **40/6**

$^{\frown}$ obtained from [DL18]
* obtained after applying Yosys synthesis tool [Wol]
† ALOY circuits

then repeated multiple times and the best result is kept. With this method, they have obtained an AES MixColumn circuit of 95 XORs after around 4 hours of computation. In the Random-Normal-BP algorithm in Section 4.1, we will see that we can actually achieve the same effect by just storing all the possible base candidates and randomly output one of them. A circuit of 95 XOR gates for AES MixColumn can be easily obtained just after a few minutes of running the algorithm. In [Max19], Maximov found a 92 XORs circuit for AES. Although he provided a lower XOR count for the AES diffusion matrix, we note that no general algorithm applying to all matrices has been described in [Max19]. The improvement comes from an interesting trick that was accomplished using hand-analysis (by fixing and blocking certain gates) in order to later force the XOR reduction search to a very specific subspace.

6 Conclusion and Future Works

In this work, we have presented new algorithms for global optimisation of linear circuits. With the introduction of randomisation in the algorithms, our heuristics performed better

Table 7: Circuit cost (XOR/Depth) of matrices of size 32×32 from [DL18] using several optimisation tools

Matrix	Instantiation (α, β, γ)	Const. [DL18]	BP [BP10]	Paar2 [PR97]	RSDF [RTA18]	RNBP Sec. 4.1	A1 Sec. 4.2	A2 Sec. 4.2
$M_{4,5}^{9,3}$	$(A_8, -, -)$	67/5*⌢ 75/5⌢	74/5⌢	88/4⌢	74/7† 82/12	**67/5**† 69/6	77/5† 79/9	69/5† 70/6
$M_{4,5}^{9,3}$	$(A_8^{-1}, -, -)$	67/5*⌢ 75/5⌢	71/6⌢	89/5⌢	79/9† 89/15	69/5† 69/6	78/6† 80/7	**68/5**† **68/7**
$M_{4,6}^{8,3}$	$(A_8, -, -)$	67/5*⌢ 67/5⌢	74/5⌢	88/4⌢	71/8† 85/18	**67/5**† 68/6	76/5† 79/8	69/5† 71/8
$M_{4,6}^{8,3}$	$(A_8^{-1}, -, -)$	67/5*⌢ 67/5⌢	71/6⌢	89/5⌢	78/7† 85/15	69/5† 69/6	78/6† 80/7	**68/6**† **68/7**
$M_{4,5}^{8,3}$	$(A_8^{-1}, A_8, A_8^{-2})$	68/5*⌢ 68/5⌢	75/6⌢	77/4⌢	81/6† 84/16	**68/5**† 75/6	**68/5**† 74/8	**68/5**† 71/6
$M_{4,4}^{9,4}$	$(A_8, -, -)$	76/4*⌢ 76/4⌢	77/6⌢	92/4⌢	84/6† 89/14	**76/4**† **76/6**	**76/6**† **76/7**	**76/6**† 77/7
$M_{4,4}^{9,3}$	(A_8^{-1}, A_8, A_8^{2})	76/4*⌢ 76/4⌢	76/6⌢	83/6⌢	79/7† 85/16	**75/6**† **75/8**	76/6† 78/8	76/6† 76/8
$M_{4,4}^{8,4}$	$(A_8, -, -)$	70/4*⌢ 70/4⌢	72/5⌢	74/4⌢	77/8† 90/17	**70/6**† **70/6**	**70/6**† **70/7**	**70/6**† **70/7**
$M_{4,4}^{8,4'}$	$(A_8, -, -)$	70/4*⌢ 70/4⌢	81/7⌢	79/5⌢	76/6† 89/10	76/5† 79/6	72/6† 75/8	**71/6**† 74/6
$M_{4,4}^{8,4''}$	$(A_8, -, -)$	69/4*⌢ 69/4⌢	72/6⌢	85/5⌢	77/7† 90/14	**69/4**† 70/6	76/4† 77/6	70/5† 70/6
$M_{4,3}^{9,5}$	$(A_8, -, -)$	77/3*⌢ 77/3⌢	**76/7**⌢	86/4⌢	82/6† 87/10	**76/4**† **76/6**	**76/5**† 77/7	**76/6**† **76/6**
$M_{4,3}^{9,5}$	$(A_8^{-1}, -, -)$	77/3*⌢ 77/3⌢	79/5⌢	86/4⌢	85/8† 91/14	**77/4**† **77/6**	**77/6**† **77/7**	**77/4**† **77/5**

⌢ obtained from [DL18]
* obtained after applying Yosys synthesis tool [Wol]
† ALOY circuits

Table 8: AES diffusion matrix circuit cost (XOR/Depth) with several algorithms. The values in bracket are the BLOY values.

Matrix	BP [BP10]	RSDF [RTA18]	RNBP Sec. 4.1	A1 Sec. 4.2	A2 Sec. 4.2
AES MixCol	97 [KLSW17]	102/6 (103/10)	95/6 (95/7)	95/6 (95/9)	**94/6** **(94/10)**
AES InvMixCol	155 (155)	162/11 (162/19)	153/10 (153/12)	153/10 (153/12)	**152/9** **(152/12)**

on average as compared to various state-of-the-art heuristics. Especially in the case of AES, we can see an improvement of 3 XORs as compared to the previous record attained by [KLSW17]. While the matrices we have shown in this paper are square matrices, this can be applied to non-square matrices as well. Algorithms A1 and A2 can be implemented

efficiently to a size of 32×32 if ρ is approximately 0.3 or less.

We have also proposed two local optimisation techniques, LocalOpt, to reduce a few more XORs from an already built circuit and we recommend the use of LocalOpt along with Yosys to obtain the best circuit.

Future works: Large matrices. While algorithms A1 and A2 yield better linear circuits than algorithms BP and RNBP in most of the cases, they are also slower than them (A1 and A2 are, however, faster than RSDF). Most of the time is spent on the exhaustive computation of the *Dist* vector. As a result, these heuristics are infeasible for very large matrices with high density. It seems like A1 and A2 can go up to approximately 32×32 with a bias of 0.3. But anything with a large bias seems to be very computationally heavy. For further research, it would be interesting to focus on finding a good global optimisation algorithm for larger matrices (so far only Paar's algorithms can easily handle large matrices). Alternatively, if the cost of computing the *Dist* vector can be reduced, then A1 and A2 could also be used to evaluate larger matrices.

Local Optimisation. Local optimisation exploits the knowledge of an efficient circuit and attempts to optimise it locally even further. Currently, LocalOpt is still in a very rudimentary stage and with just a few rules that we have implemented, we can see a portion of matrices being optimised further. Possible further research may actually look into other techniques of reducing XOR count or an integration of global and local optimisation to maximise the potential of both optimisation methods or having more techniques to reduce the XOR/depth of circuits.

Comments on KECCAK. One may think that the above algorithms can be applied to KECCAK's Theta (linear) layer. However, we do not think there is any other savings beyond the trivial ones. In the Theta layer of KECCAK, we view the structure of the primitive as a three-dimensional cuboid as in [BDPA11]. To state briefly, each bit is XORed with all the bits on its left column and the column to the front-right (we refer to [BDPA11] for a detailed description). The Theta layer can also be viewed naturally as a 1600×1600 matrix. The trivial savings are to pre-compute the XOR sum for each KECCAK column first. Then, in order to compute the output of each bit, we then use two XOR operations: one to XOR the two involved columns together, and one to XOR that value to the target state bit. We verified our intuition using Paar1 algorithm, which yielded the same circuit count as described.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. The authors are supported by Temasek Laboratories, Singapore.

References

[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 (Full version). *IACR Cryptology ePrint Archive*, 2012:529, 2012.

[BDPA11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic Sponge Functions,Submission to NIST (Round 3). `http://sponge.noekeon.org/CSF-0.1.pdf`, 2011.

[BFI19] Subhadeep Banik, Yuki Funabiki, and Takanori Isobe. More Results on Shortest Linear Programs. In Nuttapong Attrapadung and Takeshi Yagi, editors, *Advances in Information and Computer Security - 14th International Workshop on Security, IWSEC 2019, Tokyo, Japan, August 28-30, 2019, Proceedings*, volume 11689 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2019.

[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. *IACR Cryptology ePrint Archive*, 2016:660, 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.

[BM10] Robert K. Brayton and Alan Mishchenko. {ABC:} An Academic Industrial-Strength Verification Tool. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010.

[BMP08] Joan Boyar, Philip Matthews, and René Peralta. On the Shortest Linear Straight-Line Program for Computing Linear Forms. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *Mathematical Foundations of Computer Science 2008, 33rd International Symposium, MFCS 2008, Torun, Poland, August 25-29, 2008, Proceedings*, volume 5162 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 2008.

[BMP13] Joan Boyar, Philip Matthews, and René Peralta. Logic Minimization Techniques with Applications to Cryptology. *J. Cryptology*, 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.

[BPP+17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 321–345. Springer, 2017.

[Can05] David Canright. A Very Compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

[CDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2009.

[DL18] Sébastien Duval and Gaëtan Leurent. MDS Matrices with Lightweight Circuits. *IACR Trans. Symmetric Cryptol.*, 2018(2):48–78, 2018.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[GPPR12] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED Block Cipher. *IACR Cryptology ePrint Archive*, 2012:600, 2012.

[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.

[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.

[Max19] Alexander Maximov. AES MixColumn with 92 XOR gates. *IACR Cryptology ePrint Archive*, 2019:833, 2019.

[ME19] Alexander Maximov and Patrik Ekdahl. New Circuit Minimization Techniques for Smaller and Faster AES SBoxes. Cryptology ePrint Archive, Report 2019/802, 2019. https://eprint.iacr.org/2019/802.

[PR97] Christof Paar and Martin Rosner. Comparison of arithmetic architectures for Reed-Solomon decoders in reconfigurable hardware. In *5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97), 16-18 April 1997, Napa Valley, CA, USA*, pages 219–225. IEEE Computer Society, 1997.

[RTA18] Arash Reyhani-Masoleh, Mostafa M. I. Taha, and Doaa Ashmawy. Smashing the Implementation Records of AES S-box. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):298–336, 2018.

[Sto16] Ko Stoffelen. Optimizing S-Box Implementations for Several Criteria Using SAT Solvers. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2016.

[VSP18] Andrea Visconti, Chiara Valentina Schiavo, and René Peralta. Improved upper bounds for the expected circuit complexity of dense systems of linear equations over GF(2). *Inf. Process. Lett.*, 137:1–5, 2018.

[Wol] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

A Algorithmic descriptions

A.1 Randomised-Normal-BP (RNBP)

Algorithm 1: RNBP Algorithm

```
1  Input: M, a (n × m) binary matrix
2  Output: Program that evaluates M
3  Initialisation
4  Program ← {};
5  PC ← 0; // Program counter
6  Targets[i] ← M.GET_ROW(i);
7  Base ← {x_0, x_2, ..., x_{n−1}};
8  Dist[i] ← HW(Targets[i]) − 1; // Initial distance equals Hamming Weight of the row minus one
9  Main Loop
10 while Dist.GET_SUM() > 0 do
11     B ← Base.GET_LEN();
12     if ∃ a s.t. Dist[a] = 1 then
13         // Pre-emptive gate found
14         for i ∈ {0...B − 1} do
15             for j ∈ {i + 1...B − 1} do
16                 if Target[a] = x_i ⊕ x_j then
17                     Program[PC] ← y_a = x_i ⊕ x_j;
18                     Base.ADD(y_a);
19                 end
20             end
21         end
22     else
23         bestDist ← ∞;
24         bestNorm ← −∞;
25         bestBases ← {};
26         for i ∈ {0...B − 1} do
27             for j ∈ {i + 1...B − 1} do
28                 tryBase ← x_i ⊕ x_j;
29                 Dist.UPDATE({tryBase} ∪ Base);
30                 tryDist ← Dist.GET_SUM();
31                 tryNorm ← ‖Dist‖;
32                 if tryDist < bestDist or (tryDist = bestDist and tryNorm > bestNorm) then
33                     bestDist ← tryDist;
34                     bestNorm ← tryNorm;
35                     bestBases.ERASE_ALL();
36                     bestBases.ADD({x_i, x_j});
37                 else if tryDist = bestDist and tryNorm = bestNorm then
38                     bestBases.ADD({x_i, x_j});
39                 end
40             end
41         end
42         best_elem ← bestBases.RAND_ELEM();
43         Program[PC] ← t_PC = best_elem[0] ⊕ best_elem[1];
44         Base.ADD(t_PC);
45     end
46     PC ← PC + 1;
47     Dist.UPDATE(Base);
48 end
49 return Program;
```

A.2 A1 heuristic

Algorithm 2: A1 Algorithm

```
1  Input: M, a (n × m) binary matrix
2  Output: Program that evaluates M
3  Initialisation
4  Program ← {};
5  PC ← 0; // Program counter
6  Targets[i] ← M.GET_ROW(i);
7  Base ← {x_0, x_2, ..., x_{n−1}};
8  Dist[i] ← HW(Targets[i]) − 1; // Initial distance equals Hamming Weight of the row minus one
9  Main Loop
10 while Dist.GET_SUM() > 0 do
11     B ← Base.GET_LEN();
12     if ∃ a s.t. Dist[a] = 1 then
13         // Pre-emptive gate found
14         for i ∈ {0...B − 1} do
15             for j ∈ {i + 1...B − 1} do
16                 if Target[a] = x_i ⊕ x_j then
17                     Program[PC] ← y_a = x_i ⊕ x_j;
18                     Base.ADD(y_a);
19                 end
20             end
21         end
22     else
23         bestDist ← ∞;
24         bestNorm ← −∞;
25         bestBases ← {};
26         for i ∈ {0...B − 1} do
27             for j ∈ {i + 1...B − 1} do
28                 near_tar ← {c : c = argmin{ Dist[k] ∀ k ∈ {0...m − 1}}}; // set of indices of the
                     nearest targets
29                 tryBase ← x_i ⊕ x_j;
30                 OldDist ← Dist;
31                 Dist.UPDATE({tryBase} ∪ Base);
32                 // filtering
33                 if OldDist[c] = Dist[c] for all c ∈ near_tar then
34                     continue; // reject
35                 end
36                 tryDist ← Dist.GET_SUM();
37                 tryNorm ← ||Dist||;
38                 if tryDist < bestDist or (tryDist = bestDist and tryNorm > bestNorm) then
39                     // Selection with tie breaker
40                     bestDist ← tryDist;
41                     bestNorm ← tryNorm;
42                     bestBases.ERASE_ALL();
43                     bestBases.ADD({x_i, x_j});
44                 else if tryDist = bestDist and tryNorm = bestNorm then
45                     bestBases.ADD({x_i, x_j});
46                 end
47             end
48         end
49         best_elem ← bestBases.RAND_ELEM(); // randomisation
50         Program[PC] ← t_PC = best_elem[0] ⊕ best_elem[1];
51         Base.ADD(t_PC);
52     end
53     PC ← PC + 1;
54     Dist.UPDATE(Base);
55 end
56 return Program;
```

A.3 A2 heuristic

Algorithm 3: A2 Algorithm

```
1  Input: M, a (n × m) binary matrix
2  Output: Program that evaluates M
3  Initialisation
4  Program ← {};
5  PC ← 0; // Program counter
6  Targets[i] ← M.GET_ROW(i);
7  Base ← {x_0, x_2, ..., x_{n−1}};
8  Dist[i] ← HW(Targets[i]) − 1; // Initial distance equals Hamming Weight of the row minus one
9  Main Loop
10 while Dist.GET_SUM() > 0 do
11     B ← Base.GET_LEN();
12     if ∃ a s.t. Dist[a] = 1 then
13         // Pre-emptive gate found
14         for i ∈ {0...B − 1} do
15             for j ∈ {i + 1...B − 1} do
16                 if Target[a] = x_i ⊕ x_j then
17                     Program[PC] ← y_a = x_i ⊕ x_j;
18                     Base.ADD(y_a);
19                 end
20             end
21         end
22     else
23         bestDist ← ∞;
24         bestBases ← {};
25         for i ∈ {0...B − 1} do
26             for j ∈ {i + 1...B − 1} do
27                 near_tar ← {c : c = argmin{ Dist[k] ∀ k ∈ {0...m − 1}}}; // set of indices of the
                    nearest targets
28                 tryBase ← x_i ⊕ x_j;
29                 OldDist ← Dist;
30                 Dist.UPDATE({tryBase} ∪ Base);
31                 // filtering
32                 if OldDist[c] = Dist[c] for all c ∈ near_tar then
33                     continue; // reject
34                 end
35                 tryDist ← Dist.GET_SUM();
36                 if tryDist < bestDist then
37                     // Selection with tie breaker
38                     bestDist ← tryDist;
39                     bestBases.ERASE_ALL();
40                     bestBases.ADD({x_i, x_j});
41                 else if tryDist = bestDist then
42                     bestBases.ADD({x_i, x_j});
43                 end
44             end
45         end
46         best_elem ← bestBases.RAND_ELEM(); // randomisation
47         Program[PC] ← t_PC = best_elem[0] ⊕ best_elem[1];
48         Base.ADD(t_PC);
49     end
50     PC ← PC + 1;
51     Dist.UPDATE(Base);
52 end
53 return Program;
```

B Time allowed for each category in Benchmark-subset

Table 9: Time allowed for different sizes and densities for RNBP, A1, A2 and RSDF

Size	Density	Time(s)
15	0.1	20
	0.2	1000
	0.3	2000
	0.4	2000
	0.5	2000
	0.6	2000
	0.7	3000
	0.8	2000
	0.9	500

Size	Density	Time(s)
16	0.1	20
	0.2	1500
	0.3	3000
	0.4	8000
	0.5	10000
	0.6	8000
	0.7	4000
	0.8	3000
	0.9	1500

Size	Density	Time(s)
17	0.1	20
	0.2	1500
	0.3	6000
	0.4	25000
	0.5	40000
	0.6	45000
	0.7	14000
	0.8	6000
	0.9	1000

Size	Density	Time(s)
18	0.1	20
	0.2	5000
	0.3	5000
	0.4	90000
	0.5	40000
	0.6	100000
	0.7	100000
	0.8	25000
	0.9	1500

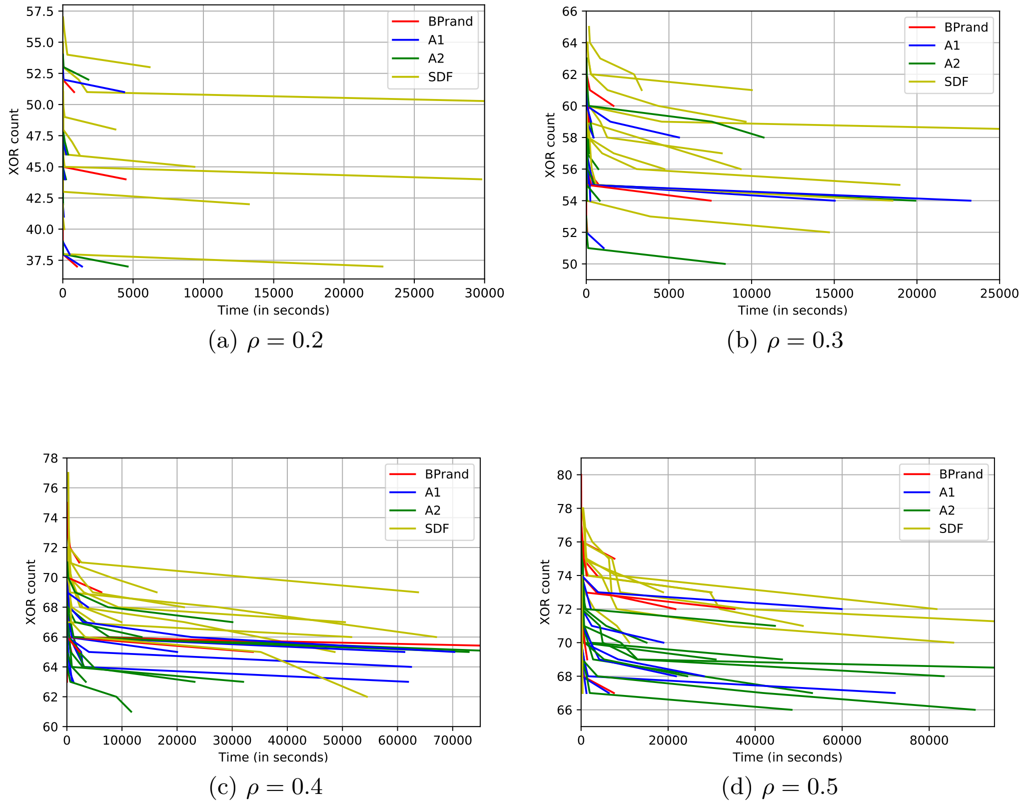
Size	Density	Time(s)
19	0.1	20
	0.2	7000
	0.3	30000
	0.4	110000
	0.5	180000
	0.6	100000
	0.7	25000
	0.8	30000
	0.9	2000

Size	Density	Time(s)
20	0.1	20
	0.2	60000
	0.3	50000
	0.4	150000
	0.5	190000
	0.6	240000
	0.7	60000
	0.8	60000
	0.9	2500

Note: As RPaar1 is extremely efficient, we allowed each matrix to have a total of 10,000 runs.

C Convergent rate of 20×20 matrices

Figure 6: Convergent rates of 20×20 matrices with ρ values from 0.2 to 0.9

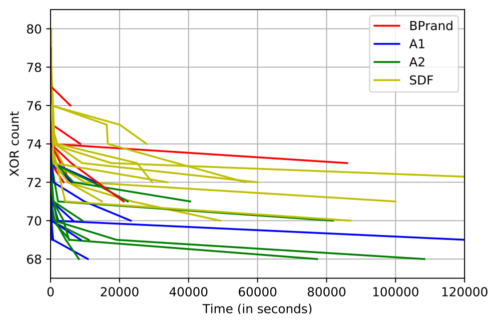


(a) $\rho = 0.2$

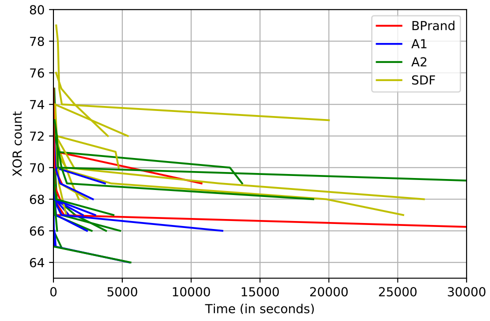
(b) $\rho = 0.3$

(c) $\rho = 0.4$

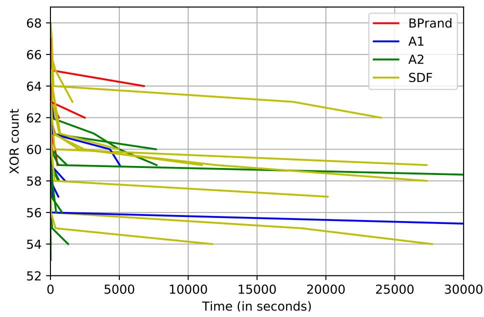
(d) $\rho = 0.5$



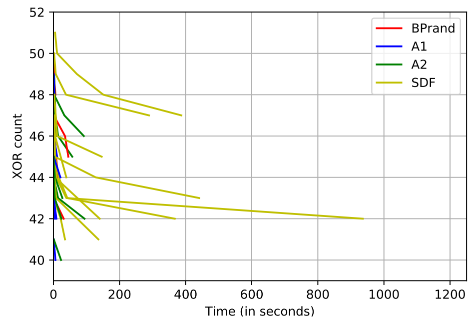
(e) $\rho = 0.6$



(f) $\rho = 0.7$



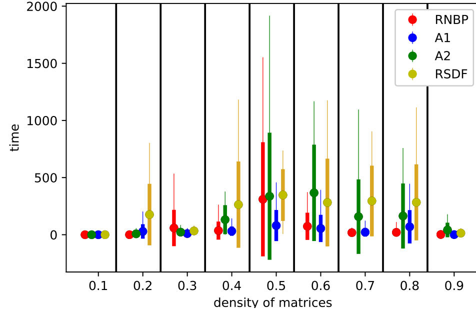
(g) $\rho = 0.8$



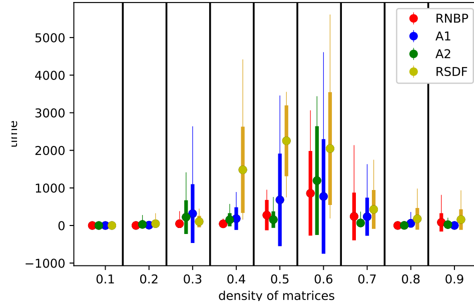
(h) $\rho = 0.9$

D Average time taken to stability

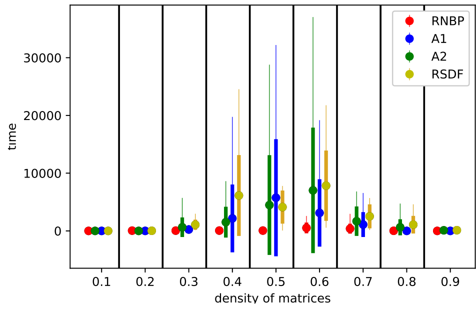
Figure 7: Average time taken to stability of 15×15 to 20×20 matrices. For every matrix, the time taken until the best circuit is formed is recorded. The average, standard deviation, minimum and maximum time (of ten matrices in each category) are shown below



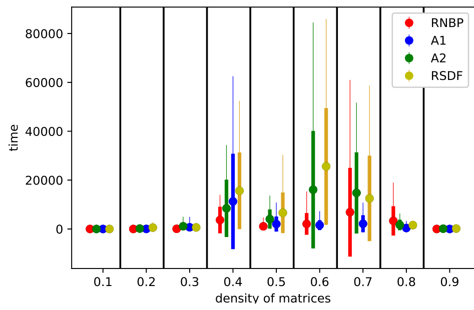
(a) 15×15



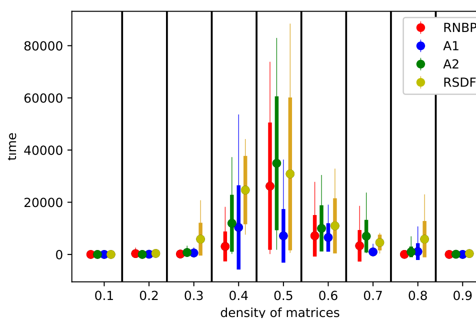
(b) 16×16



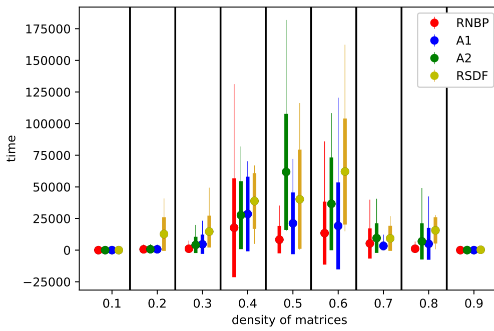
(c) 17×17



(d) 18×18



(e) 19×19



(f) 20×20

E AES circuit with 94 XORs

To avoid confusion, we use the same notations as in [BP10] to describe the circuit where $\{x_0, x_1, ..., x_{31}\}$ refers to the inputs, $\{t_i : i \in \mathbb{Z}\}$ refers to the intermediate steps and $\{y_0, y_1, ..., y_{31}\}$ refers to the targets.

$$t0 = x8 + x16$$
$$t1 = x7 + x31$$
$$t2 = x23 + t0$$
$$y15 = t1 + t2$$
$$t4 = x16 + x24$$
$$t5 = x15 + t4$$
$$y23 = t1 + t5$$
$$t7 = x1 + x25$$
$$t8 = x0 + t0$$
$$y24 = t7 + t8$$
$$t10 = x10 + x18$$
$$t11 = x17 + t10$$
$$y9 = t7 + t11$$
$$t13 = x3 + x27$$
$$t14 = x2 + t13$$
$$y26 = t10 + t14$$
$$t16 = x1 + x9$$
$$t17 = x8 + t16$$
$$y0 = t4 + t17$$
$$t19 = x18 + x26$$
$$t20 = x25 + t19$$
$$y17 = t16 + t20$$
$$t22 = x11 + x19$$
$$t23 = x2 + t19$$
$$y10 = t22 + t23$$
$$t25 = x11 + t10$$
$$t26 = x3 + t25$$
$$y2 = x26 + t26$$
$$t28 = x27 + y10$$
$$y18 = t25 + t28$$
$$t30 = x26 + t16$$
$$t31 = y9 + t23$$
$$y1 = t30 + t31$$
$$t33 = x2 + t30$$
$$y25 = x17 + t33$$
$$t35 = x17 + t4$$
$$t36 = x1 + t35$$
$$y16 = y24 + t36$$
$$t38 = x0 + t16$$
$$y8 = t36 + t38$$
$$t40 = x0 + x8$$
$$t41 = x31 + t40$$
$$t42 = x15 + t41$$
$$y7 = x23 + t42$$
$$t44 = y15 + t41$$
$$y31 = t5 + t44$$
$$t46 = x14 + x22$$
$$t47 = x21 + x29$$
$$t48 = x5 + t46$$
$$y13 = t47 + t48$$
$$t50 = t1 + t46$$
$$t51 = x30 + t42$$
$$y6 = t50 + t51$$
$$t53 = x6 + x14$$
$$t54 = t47 + t53$$
$$y5 = x13 + t54$$
$$t56 = y6 + t53$$
$$y14 = t44 + t56$$
$$t58 = x0 + x24$$
$$t59 = x6 + t50$$
$$y30 = t58 + t59$$
$$t61 = x22 + x30$$
$$t62 = t44 + y30$$
$$y22 = t61 + t62$$
$$t64 = x5 + x29$$
$$t65 = t61 + t64$$
$$y21 = x13 + t65$$
$$t67 = t46 + t65$$
$$y29 = y5 + t67$$
$$t69 = x4 + x12$$
$$t70 = x28 + t4$$
$$t71 = t47 + t69$$
$$y20 = t70 + t71$$
$$t73 = x20 + t13$$
$$t74 = x11 + t70$$
$$y19 = t73 + t74$$
$$t76 = t58 + t64$$
$$t77 = t69 + t76$$
$$y28 = x20 + t77$$
$$t79 = x12 + t0$$
$$t80 = x19 + t79$$
$$y11 = t73 + t80$$
$$t82 = t40 + t69$$
$$t83 = t28 + t82$$
$$y3 = t23 + t83$$
$$t85 = x3 + y19$$
$$t86 = y11 + t85$$
$$y27 = t82 + t86$$
$$t88 = y28 + t79$$
$$t89 = x13 + t88$$
$$t90 = x21 + t89$$
$$y4 = y20 + t90$$
$$t92 = x28 + t90$$
$$y12 = t76 + t92$$