New Circuit Minimization Techniques for Smaller and Faster AES SBoxes

Alexander Maximov and Patrik Ekdahl

Ericsson Research, Lund, Sweden
{alexander.maximov,patrik.ekdahl}@ericsson.com

Abstract. In this paper we consider various methods and techniques to find the smallest circuit realizing a given linear transformation on n input signals and m output signals, with a constraint of a maximum depth, $maxD$, of the circuit. Additional requirements may include that input signals can arrive to the circuit with different delays, and output signals may be requested to be ready at a different depth. We apply these methods and also improve previous results in order to find hardware circuits for forward, inverse, and combined AES SBoxes, and for each of them we provide the fastest and smallest combinatorial circuits. Additionally, we propose a novel technique with "floating multiplexers" to minimize the circuit for the combined SBox, where we have two different linear matrices (forward and inverse) combined with multiplexers. The resulting AES SBox solutions are the fastest and smallest to our knowledge.

Keywords: AES SBox · circuit area · circuit depth · multiplexers · linear matrices

1 Introduction

Efficient hardware design of AES SBoxes is a well-known subject. If you want the absolute maximum clocking speed of the hardware, you'd probably use a straightforward table-lookup implementation, which naturally leads to a large area. In many practical situations the area of the cryptographic subsystem is limited, and the designer cannot afford to implement table-lookup for the 16 SBoxes involved in an AES round. For these situations, we need to study how to implement an AES SBox with logical gates only, focusing on both area and maximum clocking speed. The maximum clocking speed of a circuit is determined by the *critical path* or *depth* of the circuit; the worst case time it takes to get stable output signals from a change in input signals.

Another aspect when implementing AES is, in particular, the need for the inverse cipher. Many modes of operation for a block cipher only use the encryption functionality and hence there is no need for the inverse cipher. In case you need both the forward and inverse SBox, it is often beneficial to combine the two circuits. This is because the main operation of the AES SBox is taking the inverse of a field element, which naturally is its own inverse, and we expect that many gates of the two circuits can be shared.

From a mathematical perspective, the forward AES SBox is defined as the composition of a non-linear function $I(g)$ and an affine function $A(g)$, such that $SBox(g) = A(I(g))$. The non-linear function $I(g) = g^{-1}$ is the multiplicative inverse of an element g in the finite field $GF(2^8)$ defined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. We will assume that the reader is familiar with the AES SBox, and refer to [oST01] for a more comprehensive description.

The first step towards a small area implementation was described by Rijmen [Rij00], where results from [IT88] was used. The idea is that the inverse calculation in $GF(2^8)$

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2019, No. 4, pp. 91–125
DOI:10.13154/tches.v2019.i4.91-125

can be reduced to a much simpler inverse calculation in the subfield $GF(2^4)$ by doing a base change to $GF((2^4)^2)$. In 2001, Satoh et al [SMTM01] took this idea further and reduced the inverse calculation to the subfield $GF(2^2)$. In 2005, Canright [Can05] built on the work of Satoh et al and investigated the importance of the representation of the subfield, testing many different isomorphisms that led to the smallest area design. This construction is perhaps the most cited and used implementation of an area-constrained combined AES SBox.

In a series of papers, Boyar, Peralta et al presented some very interesting ideas for both the subfield inverter as well as new heuristics for minimizing the area of logical circuits [BP10a, BP10b, BP12, BFP18]. They derived an inverter over $GF(2^4)$ with depth 4 and a gate count of only 17. The construction in [BP12] is the starting point for this paper.

After Boyar, several other papers followed focusing on low depth implementations [JKL10, NNT+10, UHS+15]. In 2018 two papers by Reyhani et al [RMTA18a, RMTA18b] presented the best known implementation (up until now) of both the forward SBox as well as the combined SBox. In [LSL+19] the authors present a very nice way to include the depth into Boyar's SLP problem [BMP13]. But the algorithm does not work with multiplexers, and hence cannot be applied to the combined SBox.

As pointed out in [RMTA18a], there are misalignments between researchers in how to present and compare implementations of combinatorial circuits. One way is to simply count the total number of standard gates in the design and find the path through the circuit that contains the critical path to determine and compare the speed. In practice it is much more complicated than that. For this paper, we present both the simple measure using only the number of gates, as well as giving a Gate Equivalent (GE) number based on the typical area required for the gate compared to the NAND gate. So for example the 2-input NAND gate will have GE=1, while the XOR gate will have a GE=2.33. The relative numbers for the GE are dependent on the specific ASIC process technology used, as well as the drive strength needed from the gate. We have used the GE values obtained from the Samsung's STD90/MDL90 0.35μm 3.3V CMOS technology [Sam00]. A comprehensive discussion on our choices for circuits comparison can be found in Appendix A. Additionally, we propose to count technological depth of a circuit normalized in terms of the delays of a XOR gate, which makes it possible to compare depths and the speed of various academic results.

The rest of the paper is organized as follows. In Section 2 we introduce the standard hardware architecture for the AES SBox. In Section 3 we describe the fundamental problem we are addressing, together with improvements to previously known techniques for solving it. The new idea of considering "floating multiplexers" is introduced in Section 4, followed by architectural improvements to the AES SBox in Section 5. The results, both theoretical and practical synthesis results, are given in Section 6. The paper ends with some conclusions and acknowledgements in Sections 7 and 7.

2 Preliminaries

We will follow the notation used in both [Can05] and [BP12] when we now construct our tower field representation. The irreducible polynomials, roots, and normal basis can be found in Table 1.

Table 1: Definition of the subfields used to construct $GF(2^8)$.

Target Field	Irreducible Poly.	Root	Coefficients in Field	Normal Base
$GF(2^2)$	$x^2 + x + 1$	W	$GF(2)$	$[W, W^2]$
$GF(2^4)$	$x^2 + x + W^2$	Z	$GF(2^2)$	$[Z^2, Z^8]$
$GF(2^8)$	$x^2 + x + WZ$	Y	$GF(2^4)$	$[Y, Y^{16}]$

Following [Can05] and [BP12], we can now derive the expression for inverting a general element $A = a_0Y + a_1Y^{16}$ in $GF(2^8)$ as

$$\begin{aligned} A^{-1} &= (AA^{16})^{-1}A^{16} \\ &= ((a_0Y + a_1Y^{16})(a_1Y + a_0Y^{16}))^{-1}(a_1Y + a_0Y^{16}) \\ &= ((a_0^2 + a_1^2)Y^{17} + a_0a_1(Y^2 + Y^{32}))^{-1}(a_1Y + a_0Y^{16}) \\ &= ((a_0 + a_1)^2Y^{17} + a_0a_1(Y + Y^{16})^2)^{-1}(a_1Y + a_0Y^{16}) \\ &= ((a_0 + a_1)^2WZ + a_0a_1)^{-1}(a_1Y + a_0Y^{16}). \end{aligned}$$

The element inversion in $GF(2^8)$ can be done over $GF(2^4)$ according to

$$\begin{array}{lll} T_1 = (a_0 + a_1) & T_2 = (WZ)T_1^2 & T_3 = a_0a_1 \quad T_4 = T_2 + T_3 \\ T_5 = T_4^{-1} & T_6 = T_5a_1 & T_7 = T_5a_0 \end{array} \tag{1}$$

where the result is obtained as $A^{-1} = T_6Y + T_7Y^{16}$. In these equations we utilize several operations (addition, multiplication, scalar, and squaring) but only two of them are non-linear over $GF(2)$; multiplication and inversion. Furthermore, the standard multiplication operation also contains some linear operations. If we separate all the linear operations from the non-linear and combine the former with the linear equations needed to do the base change for the AES SBox input, which is represented in polynomial base using the AES SBox irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, we will end up with an architecture of the SBox according to Figure 1, where we also indicate were the different parts of equations 1 are calculated.

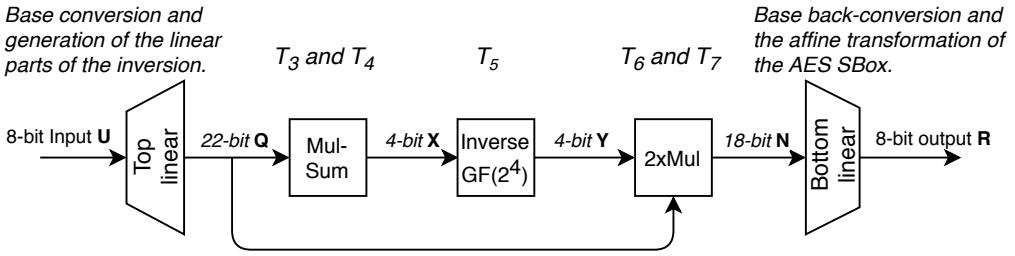

Figure 1: Architecture of the forward SBox according to [Can05] and [BP12].

In case we are dealing with the inverse SBox, we naturally need to apply the inverse affine transform to the top linear matrix instead of the bottom.

This architecture will be our starting point, and we will now provide a set of new or enhanced algorithms for minimizing both the area and the depth of the two linear top and bottom matrices.

3 Circuits for binary linear system of equations

In this section, we will recapitulate the known techniques for linear circuit minimization and propose a few improvements. We start by stating the objectives.

3.1 Basic problem statement

Given a binary matrix $M_{m\times n}$ and the maximum allowed depth $maxD$, find the circuit of depth $D \leq maxD$ with the minimum number of 2-input XOR gates such that it computes $Y = M \cdot X$. In other words, given n bits of input $X = (x_0 \dots x_{n-1})$, the circuit should

compute m linear combinations $Y = (y_0 \dots y_{m-1})$. Any circuit realization that implements a given system of linear expressions is called a **solution**.

The above problem is NP-hard [BMP08], and we have seen various heuristic approaches that help finding a sub-optimal solution in the literature. In all previous work we have studied, the assumption is that all input signals arrive in the same time, and all output signals are "ready" with delays at most $maxD$. In this paper we extend the original problem with AIR and AOR defined as follows.

Additional Input Requirement (AIR). The problem may be extended with an additional requirement on input signals X, such that each input bit x_i arrives with its own delay d_i, in terms of XOR-gates delays. The resulting depth $D \leq maxD$ then includes input delays. For example, if some input x_i has the delay $d_i > maxD$ then no solution exists. The AIR is useful while deriving the bottom matrix as described in Section 2, since after the non-linear part, the signals entering the bottom matrix will have different delays.

Additional Output Requirement (AOR). The problem may be extended by an additional requirement on the output signals. Each output signal y_i may be required to be "ready" at depth at most $e_i \leq maxD$. This is useful when some output signals continue to propagate in the critical path and other signals may be computed with larger delays, but still at most $maxD$. The AOR is used while deriving the top matrix as described in Section 2, since when we introduce multiplexers for the combined SBox, the output signals of the top matrix will be required to have different delays.

3.2 Cancellation-free heuristics

Cancellation-free heuristics are algorithms that produce linear expressions $z = a \oplus b$, where both a and b are Boolean linear expressions in the input variables, and a and b share no common terms. In other words, as we add a and b we will not cancel out any term.

Paar [Paa97] suggested a greedy approach to solving the Basic Problem in 3.1. That solution starts with the matrix M and considers all pairs of columns (i, j) in M. Then a metric is defined (on the pairs of columns) as the number of rows where $M_{r,i} = M_{r,j} = 1$, i.e. where the input variables x_i and x_j both occur. For the column pair with the highest metric, we form a new variable $x_n = x_i \oplus x_j$ and add that to the matrix (which now is of size $m \times (n + 1)$), and set positions $M_{r,i} = M_{r,j} = 0$, and $M_{r,n+1} = 1$.

Canright [Can05] also used this technique but instead of using the metric function, he performed an exhaustive search over all possible column pairs. This was possible due to the fact that the target matrix in his case was the base conversion matrix only of size 8×8. As we saw in Section 2, our bottom matrix will be considerably larger, and hence we need to take another approach. We also need to consider the AIR and the AOR.

Satisfying the AIR. When performing the above algorithm we should keep track of the depth of the newly added XOR gates. This is done by having a vector $D = (d_0 \dots d_{n-1})$ with the current depth of all inputs and newly added signals x_i. When the new signal $x_n = x_i \oplus x_j$ is added, the delay of x_n is trivially $d_n = \max(d_i, d_j) + 1$. We then also restrict the algorithm such that if $d_n > maxD$ then we are not allowed to add x_n as a new input signal. The AIR is hereby satisfied automatically.

Satisfying the AOR. Similarly, when adding a new input variable x_n, we need to check if a solution is theoretically possible. An elegant solution to this is presented in Theorem 2 in [LSL+19] where they calculate the shortest circuit given additional delay constraints.

Probabilistic heuristic approach. Since we cannot perform a full exhaustive search on the bottom matrix due to its size, we need to confine the number of pairs to keep and further evaluate. We have found that keeping the K best candidates (based on the original metric by Paar) and then randomly selecting which one to pick for the next XOR gate is a good strategy. In our simulations, this probabilistic approach gave us much smaller circuits than only considering the best metric candidates. Naturally, the execution time will be too

long if we pick a too large K, and conversely picking a too small K decreases the chances of deriving a good circuit. In practice we found that $K = 2, \ldots, 6$ is a reasonable number of candidates to keep and try.

3.3 Cancellation-allowed heuristic

The cancellation-free approaches give sub-optimal results, as it was shown by Boyar and Peralta in [BP10a], where they also introduced a new algorithm that allows cancellations. This was later improved by Reyhani et al in [RMTA18a]. Next, we briefly describe the basic idea of that heuristic.

3.3.1 Basic cancellation-allowed algorithm [BP10a]

Every row of M is an n-bit binary vector. That vector can be seen as an n-bit integer value. We define that integer value as a **target point**. Thus, the matrix M can be seen as the column vector of m target points. The input signals $\{x_0, \ldots, x_{n-1}\}$ can also be represented as integer values $x_i = 2^i$, for $i = 0, \ldots, n-1$.

Let the *base set* $S = \{s_0, \ldots, s_{n-1}\} = \{1, 2, 4, \ldots, 2^n\}$ initially represent the input signals. The key function of the algorithm is the distance function $\delta_i(S, y_i)$ that returns the smallest number of XOR gates needed to compute a target point y_i from the set of known points S. The algorithm keeps a vector $\Delta = [\delta_0, \delta_1, \ldots, \delta_{n-1}]$ which is initially set to the Hamming weight minus one of the rows of M, which would be the number of XOR gates needed without any sharing of intermediate gates.

The algorithm then proceeds by combining two base points s_i and s_j in the base set S, and xor them together producing a candidate point $c = s_i \oplus s_j$. The selection of s_i and s_j is performed by an exhaustive search over all distinct pairs, and then for each candidate point, the sum of the distance vector $\sum \delta_i$, for $i \in [0, n-1]$, is calculated. Note that the distance functions δ_i now is computed over the set $S \cup \{c\}$. The pair which gives the smallest distance sum is picked and S is updated $S = S \cup \{c\}$. In case there is a tie, the algorithm picks the pair that *maximizes* the Euclidean norm $\sqrt{\sum \delta_i^2}$, for $i \in [0, n-1]$. If there is a tie after this step too, the authors in [BP10a] investigated different strategies and concluded that all strategies tested performed similarly, and hence a simple random selection can be used. The algorithm then repeats the step of picking two new base points and calculating the distance vector sum, until the distance vector is all-zeros and the targets are all found. In the original description, there is also a notion of "preemptive" choices. A preemptive choice is a candidate point c such that it directly fulfils a target row in the matrix M. If such a candidate is found, it is immediately used as the new point and added to S.

Reyhani et al [RMTA18a] improved the original algorithm from [BP10a] by directly searching for preemptive candidates in each round and add them all to the set S before the "real" candidate is added and the distance vector recalculated. They also improved the tie resolution strategy and kept all the candidates that were equally good under the Euclidean norm and recursively tried them all, keeping the one that was best in the next round.

When the maximum depth $maxD$ is a required constraint, the newly proposed algorithm in [LSL+19] can be used. However, in our simulations for bottom matrices, it didn't produce better results than the cancellation-free algorithm with randomization factor.

3.4 Exhaustive search methods

In this section we present an algorithm for an efficient exhaustive search of the minimal circuit. The overall complexity is exponential in the number of input signals, and linear in

the number of output signals. From our experiments we can conclude that this exhaustive search algorithm can be readily applied to circuits of up to approximately 10 input bits.

3.4.1 Notations and data representation

Using the same integer representation of the rows of M, and the input signals x_i as in Section 3.3.1, we can re-phrase the basic problem statement: given the set of input points x_i we want to find the sequence of XORs on those points such that we get all the m wanted target points y_i, the rows of the matrix M, with the maximum delay $maxD$. Input and output points may have different delays d_i and e_i, respectively.

For data structures, we can store a set of 2^n points as either a normal set, and/or as a bit-vector. The set makes it possible to loop through the points while the bit-mask representation is efficient to test set membership.

3.4.2 Basic idea

The proposed exhaustive search algorithm is a recursive algorithm, *iterating over the depths*, starting at depth 1 and ending at $maxD$. At each depth D, we try to construct new points from the previous depths, thereby constructing circuits that are of exactly depth D. When all target points are found, we check the number of required XOR gates, keeping track of the smallest solution. We will need the following sets of points:
`known`[maxD+1] – the set of known points at certain depth D.
`ignored`[maxD+1] – the set of points that will be ignored at depth D.
`targets` – the set of target points.
`candidates` – the set of candidate points that can be added to the set `known` at the current recursion step.

The initial set of known points is x_i, for $i = 0 \ldots n-1$, and the set of target points is y_i, for $i = 0 \ldots m-1$. AIR is met by initially placing the input point x_i to the known set at depth d_i. AOR is satisfied by setting the point y_i with output delay e_i to the ignore list on all depth levels that are larger than e_i.

We will now explain the steps executed at each depth of the recursion, assuming that we currently are at depth D.

Step 1 – Preemptive points. Check the `known`[D] set to see if any pair can be combined (XOR:ed) to give a target point not yet found. If all targets are found, or if we have reached $maxD$, we return from this level of the recursion.

Step 2 – Collect candidates. Form all possible pairs of points from the `known`[$0..D-1$] sets, where at least one of the points is from `known`[$D-1$], and XOR the pair to derive a new point. If the derived point is in the set `ignored`[D] then we skip it, otherwise we add it to the `candidate` set.

Step 3 – In this step we try to add points from the `candidate` set to the known list, and call the algorithm recursively again. We start by trying to add 1 point and do the recursive call. If that's not solving the target points, we'll try to add 2 points, and so on until all combinations (or a maximum number of combinations) of the points in the `candidate` set have been tried.

3.4.3 Ignored points and other optimizations

In step 2, we check the candidate set against the `ignored`[D] set, the set of ignored points at depth D. The `ignored` set is constructed from a set of rules; **Intersection:** A candidate point p should be ignored if for all target points w_i we get $(w_i \& p) \neq p$. This means that the point p covers too many of the input variables, and is not covered by any of the points in the `targets` set; **Forward Propagation:** We can calculate all possible points on each level starting from the top level $D = 0$ with n known points and

going down to $D = maxD$. Those points that can never appear at some level d are then included into the `ignored`[d] set. If some target point w has another desired maximum delay $e_i < maxD$, then that point on the following depths should be ignored, i.e., we add w to `ignored`$[e_i + 1..maxD]$; **Sum of Direct Inputs:** If any of the input signals x_i, x_j give the point $p = x_i \oplus x_j$ on level d, then all consecutive levels $> d$ must have the point p in the ignored list; **Backward Propagation:** As a last check, we can go backwards level by level, starting from $d = maxD$ and ending at level $d = 1$, and for each allowed (not ignored) point on the level d we check whether there is still a not-ignored pair a, b at the previous levels (one of a or b must be on the level $d-1$) such that it gives $p = a \oplus b$. If not, then the point p should be added to the `ignore`[d] set; **Ignore Candidates:** *dynamically* add a point w to the `ignore`$[d]$ set if w has been one of the candidates at previous levels $< d$.

3.5 Remarks

Simulations show that regarding searching for the minimum solution the top matrix (with only 8 inputs) can be solved with the *exhaustive cancellation-allowed search* as in Section 3.4. The bottom matrix (with 18 inputs) is too large for a direct exhaustive search, and we should start with a *probabilistic cancellation-free heuristic* from Section 3.2, and then use a full exhaustive search for the ending part, when the Hamming weights of the remaining rows become small enough to perform the exhaustive search. This approach gave us the best result.

4 System of linear circuits with multiplexers

Assume we want to find a solution for the combined AES SBox, where the top and the bottom linear matrices need to be multiplexed based on the SBox direction. This means that the circuit for the combined linear expressions is basically doubled in size, plus the set of multiplexers. In this section we will show how to deal with multiplexed systems of linear expressions. We will show that the MUX and XOR gates can be considered in a combined way in order to achieve a very compact circuit.

4.1 Floating multiplexers

Consider that for some signal Y we have to compute two linear expressions Y^F and Y^I for the forward and the inverse SBoxes respectively. Then we apply a multiplexer so that only one of the signals continues as Y. Assume further that the signals Y^F and Y^I share some part of the expression. Then it may be better to push that shared part after the multiplexer, and the resulting solution can be simplified.

For example, let $Y^F = X_0 \oplus X_1$ and $Y^I = X_0 \oplus X_2$, then normally we should spend 2 XOR gates and 1 multiplexer, so that we get $Y = MUX(\texttt{select}, X_0 \oplus X_1, X_0 \oplus X_2)$ with 3 gates. However, we can push the common part X_0 after the multiplexer as follows:

$$Y = MUX(\texttt{select}, X_1, X_2) \oplus X_0,$$

then we get a circuit with only 2 gates. In general, one can pick any linear combination Δ on input signals and make a substitution:

$$Y = MUX(\texttt{select}, Y^F, Y^I) \to MUX(\texttt{select}, Y^F \oplus \Delta, Y^I \oplus \Delta) \oplus \Delta,$$

where Δ is then added to the linear matrix as an additional target signal to compute. If that substitution leads to a shorter circuit then we keep it. We should also choose such Δ that the overall depth is not increased. Thus, various multiplexers will be "floating" over the depth of the circuit. Signals with $\Delta \neq 0$ should have their maximum depth decreased by 1.

4.1.1 Metrics and linear expressions to solve

We have n input signals $X_1 \dots X_n$ and m output signals $Y_1, \dots, Y_m$, where each Y_i is represented in its most general form as a triple (A_i, B_i, C_i) such that

$$Y_i = A_i \oplus MUX(\texttt{select}, B_i, C_i),$$

where A_i, B_i, and C_i are linear expressions on the input signals. We are allowed to modify the above expression as $(A_i \oplus \Delta_i, B_i \oplus \Delta_i, C_i \oplus \Delta_i)$ for any Δ_i, since the Boolean function of Y_i will not change.

Let ABC represents the linear matrix that describes all the rows A_i, B_i, and C_i, for $i = 0, \dots, m$, such that

$$ABC \times X$$

gives the wanted linear system to realize using minimal number of gates and a given $maxD$. By choosing favorable values of Δ_i, one can shrink the number of total gates, since some of the target points of ABC may become equal to each other, and hence ABC can be reduced by at least one row. Also, some of the targets may become 0 or having only one bit - i.e., they are equal to corresponding input signals. These targets are also removed from the linear system as they are trivial and cost zero gates. After the above reductions we get a system of linear expressions where all rows are distinct and have Hamming weight at least 2. As before, we interpret the rows of ABC as integers, and adding (XORing) a Δ_i to the three rows A_i, B_i, and C_i will change those three target points, but not the resulting Y_i.

Metric. The search for a good combination of Δs requires a lot of computations and it rapidly becomes infeasible to compute a minimal solution for each selection. Thus, we need to decide on a good metric that allows us to truncate the search space down to promising sets of Δs. We propose to adopt a metric that is based on the lower bound of the number of gates of a fixed system (when Δ values are selected), and define the metric to be the number of rows of the reduced ABC matrix, plus the minimum number of extra gates needed to complete the circuit, such as multiplexers.

In the following we present several heuristic approaches to finding a good set of Δs while minimizing the metric.

4.1.2 Iterative algorithms to find Δs: metric→minimize

The below techniques only work for small n, but in our case they are readily applicable to the 8-input top matrix of the AES SBox.

Algorithm-A(k) – Select k triplets (A_i, B_i, C_i) and try to find k matching Δ_is that minimize the metric. If some choice results in a smaller metric, we keep that choice and continue searching with the updated ABC matrix. The algorithm runs in a loop until the metric is not decreasing any more. Algorithm-A(1) is quite fast, and Algorithm-A(2) also has acceptable speed. For larger ks it becomes infeasible. Algorithm-A(k) works fine for a very quick/brief analysis of the given system but the result is quite unstable since for a random set of initial values of Δ_is the resulting metric fluctuates heavily.

Algorithm-B – unlike Algorithm-A this algorithm is trying to construct a linear system of expressions, starting from an empty set of knowns S and then trying to add new points to S one by one, until all targets of ABC become included in the set S. While testing whether a new candidate c should be added to S we loop through all (A_i, B_i, C_i) and for each one try to find a Δ_i that minimizes the overall metric. This heuristic algorithm is a lot more stable and gives quite good results.

However, the smallest possible metric does not guarantee that the final solution will have the smallest number of gates, and the number of non-target intermediates needed is unclear. Thus, it would be a good idea to collect a number of promising systems whose metric is the smallest possible, then try to find the smallest solution amongst them. We will investigate this in the next section.

4.2 New generic heuristic technique for linear systems with floating multiplexers

If we generalize the idea of floating multiplexers and let them float even higher up in the circuit, and also sharing them wider, we could achieve better results. In this section we propose a generic heuristic algorithm that finds good circuits for such systems.

4.2.1 Problem statement

We are given n-bit input signal X_n, binary matrices $M^F_{m\times n}$ and $M^I_{m\times n}$, binary vectors A^F_n, A^I_n, B^F_m, B^I_m, and vectors of delays D^X_n and D^Y_m. We want to find a smallest and shortest solution that computes the m-bit output signal Y:

$$\begin{aligned} Y^F &= M^F \cdot (X \oplus A^F),\\ Y^I &= M^I \cdot (X \oplus A^I),\\ Y &= MUX(ZF, Y^F \oplus B^F, Y^I \oplus B^I), \end{aligned}$$

where each input signal X_i has an input arrival delay D^X_i and each output signal Y_j must have the total delay *at most* D^Y_j. A^* and B^* are constant masking vectors for the input and output signals respectively (NOT-gates). ZF is the mux selector, when $ZF = 1$ we pick the first (Y^F = "forward") output otherwise the second (Y^I = "inverse") output. We also assume there is a complement signal $ZI = ZF \oplus 1$ that is also available as an input control signal.

4.2.2 Preliminaries

Similar to our previous notation, we define a "point" to be tuple of a point value (.p) and a delay (.d):

$$\texttt{point:=}\{\texttt{.p=}[f(1\ bit)|\mathbf{F(n\ bits)}|i(1\ bit)|\mathbf{I(n\ bits)}],\ \texttt{.d=}Delay\},$$

which is then translated into a 1-bit signal circuit

$$\texttt{signal:=}MUX(ZF, F\cdot X \oplus f, I \cdot X \oplus i),$$

with a total output delay `point.d`. I.e., F and I are linear combinations of the n-bit input X, and f and i are negate bits applied to the result in case the selector ZF is "forward" or "inverse", respectively. The n input points are then represented as:

$$\texttt{input point}\ X_k := \{\texttt{.p=}[A^F_k|\mathbf{2^k}|A^I_k|\mathbf{2^k}],\ \texttt{.d=}D^X_k\},\ \text{for } k = 0,\ldots,n-1,$$

and the target m points are:

$$\texttt{target point}\ Y_k := \{\texttt{.p=}[B^F_k|\mathbf{Y^F_k}|B^I_k|\mathbf{Y^I_k}],\ \texttt{.d=} \le D^Y_k\},\ \text{for } k = 0,\ldots,m-1.$$

We should also include the following 4 trivial points to the set of inputs:

`signal ZF :=` $\{$`.p=`$[1|\mathbf{0}|0|\mathbf{0}]$, `.d=`$0\}$, `signal 0 :=` $\{$`.p=`$[0|\mathbf{0}|0|\mathbf{0}]$, `.d=`$0\}$,
`signal ZI :=` $\{$`.p=`$[0|\mathbf{0}|1|\mathbf{0}]$, `.d=`$0\}$, `signal 1 :=` $\{$`.p=`$[1|\mathbf{0}|1|\mathbf{0}]$, `.d=`$0\}$.

Given any two (ordered) points v and w there are at most 6 possible new points that can be generated based on the following gates:

$$\begin{aligned}
\texttt{MUX}(v,w) &:= \{\texttt{.p=}[v.f|\mathbf{v.F}|w.i|\mathbf{w.I}],\ \texttt{.d=}D_{new}\},\\
\texttt{NMUX}(v,w) &:= \{\texttt{.p=}[v.f\oplus 1|\mathbf{v.F}|w.i\oplus 1|\mathbf{w.I}],\ \texttt{.d=}D_{new}\},\\
\texttt{MUX}(w,v) &:= \{\texttt{.p=}[w.f|\mathbf{w.F}|v.i|\mathbf{v.I}],\ \texttt{.d=}D_{new}\},\\
\texttt{NMUX}(w,v) &:= \{\texttt{.p=}[w.f\oplus 1|\mathbf{w.F}|v.i\oplus 1|\mathbf{v.I}],\ \texttt{.d=}D_{new}\},\\
\texttt{XOR}(v,w) &:= \{\texttt{.p=}[w.f\oplus v.f|\mathbf{w.F}\oplus\mathbf{v.F}|w.i\oplus v.i|\mathbf{w.I}\oplus\mathbf{v.I}],\ \texttt{.d=}D_{new}\},\\
\texttt{NXOR}(v,w) &:= \{\texttt{.p=}[w.f\oplus v.f\oplus 1|\mathbf{w.F}\oplus\mathbf{v.F}|w.i\oplus v.i\oplus 1|\mathbf{w.I}\oplus\mathbf{v.I}],\ \texttt{.d=}D_{new}\},
\end{aligned}$$

where $D_{new} = \max\{\texttt{v.d}, \texttt{w.d}\}+1$. Note that the inclusion of the 4 trivial points is important, since then we can limit the number of gate types to be considered. For example, a NOT-gate is then implemented as $\texttt{XOR}(v,1)$, AND gate with ZF can be implemented as $\texttt{MUX}(v,0)$, OR gate with ZI is $\texttt{MUX}(v,1)$, etc.

4.2.3 The floating multiplexers algorithm

We start with the set S of input points (of size $n+4$), and place all target points into the set T. At each step, we compute the set of candidate points C that is generated by applying the above 6 gates to any two points from the set S. Naturally, C should only contain unique points and exclude those already in S. We try to add one candidate point from C to S and compute the distances from S to each of the target points in T. Thereafter we compare metrics to decide which candidate point will be included into S at this step, and start over by calculating the possible candidates. The algorithm stops when the overall distance δ-metric is 0.

The metric consists of several values. The distance $\delta(S,t_i)$ is the minimum number of basic gates (the above 6) required to get the target point t_i from the points in S, such that the delay is at most D_i^Y. Subsection 4.2.5 discusses how to compute $\delta(S,t_i)$. The applied metrics and their order of importance are then as follows:

$$\begin{aligned}
\gamma &= (|S|-n-4)+\sum_{i=0}^{m-1}\delta(S,t_i)\rightarrow\min,\\
\delta &= \sum_{i=0}^{m-1}(\delta(S,t_i)-(\delta(S,t_i)==1))\rightarrow\max,\\
\tau &= \text{delay of the recent candidate point from } C \text{ added to } S\rightarrow\min,\\
\nu^2 &= \sum_{i=0}^{m-1}(\delta(S,t_i)-(\delta(S,t_i)==1))^2\rightarrow\max.
\end{aligned}$$

The metric γ is the *projected number of gates* in case there will be no more shared gates; that metric we should definitely minimize. In case there are several candidates that give the same value, then we look into the second metric δ.

δ is the sum of distances *excluding* distances where only 1 gate is needed. Given the smallest γ, we must maximize δ. The larger δ the more opportunities to shrink γ. We exclude distances 1 because of the inclusion of the preemptive step that we will describe below. When we accept candidates to S one by one as described above, the metrics δ and γ are similar, but will become distinct when we, in the next subsection, introduce a search tree where the size of $|S|$ may differ.

τ selects the candidate having the minimum depth in case the above two metrics showed the same values for two candidates. In case there are no maximum depth constraints for target points then this metric is not needed.

ν is the Euclidean norm excluding the preemptive points (similar to δ). This is the last decision metric since it is not a very good predictor, a worse value may give a better result and vice versa. However, if there are two candidates with equal metrics δ, γ, and τ, then ordering of the two candidates may be done based on ν. An alternative approach in case of tie-candidates is to choose one of them randomly.

Preemptive points. If some distance $\delta(S, t_i) = 1$ then we accept the point t_i into S immediately without the search through the candidates C. The inclusion of this step in the algorithm forces us to exclude such points from the metrics δ and ν.

In [RMTA18a] preemptive points were included into the metric, but we believe it was not fully correct. E.g., when two distance vectors $\{1, 2, \ldots\}$ and $\{0, 2, \ldots\}$ have the same projected gates, then they fall into a totally equal situation in terms of possible shared gates, thus they should result in the same δ. The point with the distance 1 in the above vector will be included into the circuit immediately (preemptive point), and it does not give any advantage over the second choice where we have a point with the distance 0. Therefore, distances with the value 1 should be ignored in δ and μ, but they should be accounted in the projected gates γ, instead.

4.2.4 Search tree

Additionally to the above algorithm, we propose to have a *search tree* where each node is a set S with metrics. Children of such a node are also nodes where S' is derived from S by adding one of the candidate point $S \leftarrow C$. Thus, every path from the root node to a leaf represents a sequence of accepted candidate points to the root set S. If, at some point, a leaf has metric $\delta = 0$ then that leaf represents a possible solution path.

We keep a number of children nodes (in our experiments we kept at least 20-50 best children) whose metrics are the best (they may even have different projected gates γ). We also define the maximum depth TD of the search tree (in our experiments we tried $TD = 1, \ldots, 20$). When the tree at depth TD is constructed, we then examine the leaves and see where we get the best metric over all leaves at all different branches. Tracking back to the root, we then choose to keep the top branch that leads to the best leaf(s). Other top branches from the root are removed. We then advance the root node to the first child of the selected branch and try to extend the tree's depth again from the remaining leaves, thus, keeping the search tree at a constant depth TD.

If, at every depth of the tree, each leaf is extended with additional 20-50 sub-branches, then the number of leaves will increase exponentially. However, we can apply a **truncation algorithm** to the leaves before extending the tree to the next depth. We simply keep no more than a certain number of promising leaves that will be expanded to the next depth, and other, less promising leaves we just remove from the tree (in our experiments the truncation level was up to 400 leaves overall for the whole tree). This type of truncation makes it possible to select the best top branch of the root node by "looking further" basically *at any depth* TD. Notably, the complexity does not depend on the depth TD, but it depends on the truncation level.

Truncation strategy. In brief, we keep those leaves with the best metrics, but try to distribute nearly equal leaves among different branches, so that we keep as many *diverted solution paths* as possible.

4.2.5 Computation of $\delta(S, t_i)$

The "heart" and the critical part of the algorithm is the subalgorithm to compute the distances $\delta(S, t_i)$, given a fresh S. There are many candidates to test at each step, and there are many branches to track, so we need to make this core algorithm as fast as possible.

Note that the length of a point (`.p` is an integer) is $2n + 2$ bits, plus the delay value. We will ignore the delay (`.d`) value when doing Boolean operations over two points. Let us assign the number of possible points as:

$$N = 2^{2n+2}.$$

Let $V_k[]$ be a vector of length N cells, each cell $V_k[p]$ corresponds to a $(2n + 2)$-bit point p represented as an integer index, and the value stored in the cell will be the minimum delay $p.d$ of that point such that it can be derived from S with *exactly* k gates.

Set the initial vector V_0 as $\forall p \rightarrow V_0[p] = p.d$, if $p \in S$, and $V_0[p] = \infty$, otherwise. Thereafter, the vector V_{k+1} can be derived from the previously derived vectors $V_0 \dots V_k$ by applying the allowed 6 gates to points from some level $0 \leq l < k$ (V_l) and the level $k - l$ (V_{k-l}), thus resulting in total $l + (k - l) + 1 = k + 1$ gates. After a new V_{k+1} is derived, we simply check if it contains new distance values for the targets from T, and we repeat the procedure until all distances $\delta(S, t_i)$ for all t_i in T are found. A high-level description of the algorithm is given in Algorithm 1, and in Appendix B.1 we provide a more detailed description alongside multiple computational tricks that can be made.

Algorithm 1 Algorithm for computing $\delta(S, t_i)$

```
1:  function DISTANCES(S, T, maxδ) → Δ = {δ_i},  i = 0, ..., m − 1
2:      Init δ_i = ∞ for i = 0, ..., m − 1
3:      Init ∀p : V_0[p] = p.d if p ∈ S, otherwise ∞
4:      Init k = 0
5:      while true do
6:          while ∃i : δ_i = ∞ and V_k[t_i] ≤ t_i.d do
7:              δ_i = k
8:          if ∀i : δ_i < ∞ then return OK
9:          if k ≥ maxδ then return FAIL
10:         k ← k + 1
11:         Init ∀p : V_k[p] = ∞
12:         for all a, b : do
13:             for p in {MUX(a, b), NMUX(a, b), MUX(b, a), NMUX(b, a), XOR(a, b), NXOR(a, b)} do
14:                 for l ← ⌊k/2⌋ to k − 1 do
15:                     d ← max(V_{k−l−1}[a], V_l[b]) + 1
16:                     V_k[p] ← min(V_k[p], d)
```

4.2.6 Double and Useless points

"Double" points. When, at some step of the algorithm, we find a candidate point c that is already in S but now having a smaller depth, then the point c is kept in C and tested along with other candidates. If it turns that adding c to S gives the best metric, we add it to S. An alternative strategy would be to update the point $c.p$ in S with the lower depth

c.d and recalculate depths of dependent points. However, it is not clear what to do with the parent points that were used to generate that previous *c.p* in S. We leave this as an open question for further research.

"Useless" points. At the end of the algorithm (when $\delta = 0$), it could happen that S contains points that can be safely excluded while a solution can still be derived. As a final step, we try to remove points from S one by one and test if every target is reachable from the remaining S under the given depth constraints. In our experience this situation is rare, but it helped to remove 1-2 gates, mainly caused by "double" points.

The above problems with "double" and "useless" points are generic for such class of algorithms where certain depth constraints should be met, and Algorithm 1 in [LSL+19] also falls under this category.

5 Architectural improvements

Most known AES SBox architectures look quite similar, consisting of the Top and Bottom linear parts, and the middle non-linear part, as previously described in Section 2. In this section, we take that classic design and propose a number of improvements, along with a completely new architecture that focuses on low depth solutions.

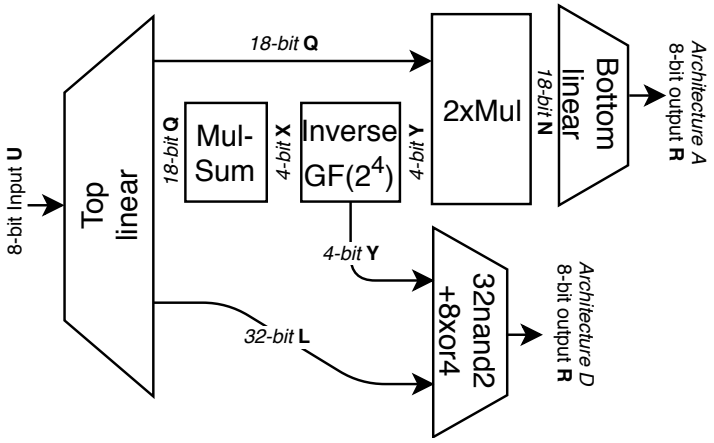

Figure 2: Difference between the architectures A and D.

5.1 Two SBox architectures – A*rea* and D*epth*

Referring to Figure 2, the **architecture A (Area)** is the classical one that implements designs based on tower and composite fields. It starts with the 8-bit input signal **U** to the **Top linear matrix**, which produces a 22-bit signal **Q** (as in [BP12]). We managed to reduce the number of needed **Q**-signals to 18, and refactored the multiplication and linear summation block `Mul-Sum` to 24 gates and depth 3. (See Appendix D.2 for equations). The output from the `Mul-Sum` block is the 4-bit signal **X** which is the input to the inversion over GF(2^4). The output from the inversion, **Y**, is non-linearly mixed with the **Q** signals, derived in the top matrix, and produces 18-bit signal **N**. The final step is the **Bottom linear matrix** that takes 18-bit **N** and linearly derives the output 8-bit signal **R**. The top and bottom matrices incorporate the SBox's affine transformation that depends on the direction.

In the new **architecture D (Depth)** we tried to remove the "irregular" bottom matrix and as a result shrinking the depth of the circuit as much as possible. The idea behind is that the bottom matrix only depends on the set of multiplications of the 4-bit signal **Y**

and some linear combinations of the 8-bit input **U**. Thus, the result **R** can be achieved as follows:

$$\mathbf{R} = Y_0 \cdot M_0 \cdot \mathbf{U} \oplus \ldots \oplus Y_3 \cdot M_3 \cdot \mathbf{U},$$

where each M_i is a 8×8 matrix representing 8 linear equations on the 8-bit input **U**, to be scalar multiplied by the Y_i-bit. Those 4x8 linear circuits can be computed as a 32-bits signal **L** *in parallel* with the circuit for the 4-bits of **Y**. The result **R** is achieved by summing up four 8-bit sub-results. Therefore, in the architecture D we get the depth 3 after the inversion step (critical path: `MULL` and `8XOR4` blocks), instead of the depth 5-6 in the architecture A. That new architecture D requires a bit more gates, since the assembling bottom circuit needs 56 gates: `32NAND2+8XOR4`. The reward is the lower depth.

A more detailed sketch of the two architectures is given in Figure 3, that includes the components of the designs, delays and the number of gates.

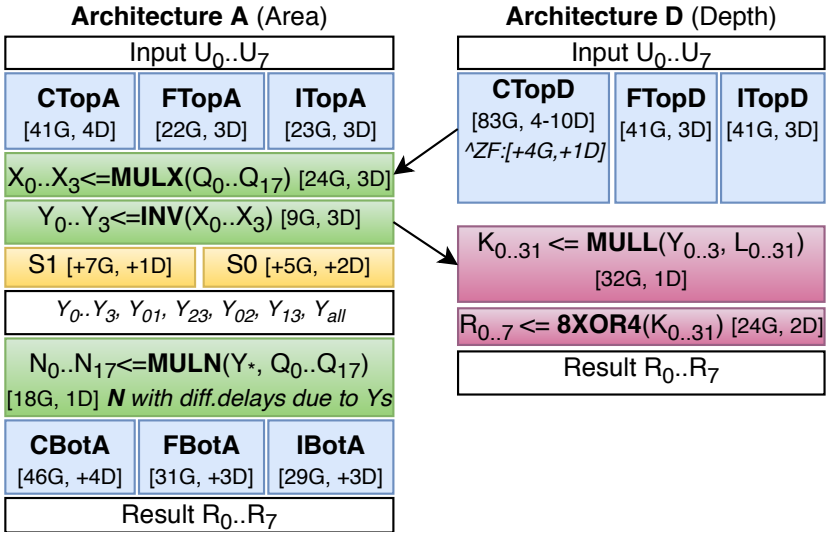

Figure 3: More details on the architectures A and D.

5.2 Six different scenarios of `MULN`

In the `MULN` block, where the 18-bit **N**-signals are computed, we need as input the 18-bit **Q**-signals and the inversion result **Y**. But we also need the following additional linear combinations of **Y**: $Y_{02} = Y_0 \oplus Y_2$, $Y_{13} = Y_1 \oplus Y_3$, $Y_{23} = Y_2 \oplus Y_3$, $Y_{01} = Y_0 \oplus Y_1$, $Y_{00} = Y_{01} \oplus Y_{23}$ – these correspond to the signals `M41-M45` in [BP12]. Thus, the **Y** vector is actually extended to 9 bits, and the delays of **N** bits become different, depending on which of the Y_i is used in the multiplication. For example, in the worst case, the delay of Y_{00} is +2 compared to the delay of Y_1. Thus, the resulting signals **N** will have different output delays. However, it is possible to compute these 5 additional **Y**s in parallel with the base signals $Y_0, \ldots, Y_3$. This will cost some extra gates, but then the +2 delay can either shrink down to +1 or +0. In general one can consider the following 6 scenarios:

- **S0**. We compute only the base signals $Y_0, \ldots, Y_3$, and the remaining $\{Y_{01}, Y_{23}, Y_{02}, Y_{13}, Y_{00}\}$ we compute with XORs as above. The delay is +2 but it has the smallest number of gates;
- **S1**. Compute $\{Y_{01}, Y_{23}\}$ in parallel, the delay is +1;
- **S2**. Compute $\{Y_{02}, Y_{13}\}$ in parallel, the delay is +1;

- **S3**. Compute $\{Y_{00}\}$ in parallel, the delay is +1;
- **S4**. Compute $\{Y_{01}, Y_{23}, Y_{02}, Y_{13}\}$ in parallel, the delay is +1;
- **S5**. Compute $\{Y_{01}, Y_{23}, Y_{02}, Y_{13}, Y_{00}\}$ in parallel, the delay is +0 as there is no signal left to compute afterwards.

In the next subsection we show how to find Boolean expressions for the above scenarios.

5.3 `INV`. Inversion over GF(2^4)

The inversion formulae are as follows:

$$\begin{aligned}
Y_0 &= X_1X_2X_3 \oplus X_0X_2 \oplus X_1X_2 \oplus X_2 \oplus X_3,\\
Y_1 &= X_0X_2X_3 \oplus X_0X_2 \oplus X_1X_2 \oplus X_1X_3 \oplus X_3,\\
Y_2 &= X_0X_1X_3 \oplus X_0X_2 \oplus X_0X_3 \oplus X_0 \oplus X_1,\\
Y_3 &= X_0X_1X_2 \oplus X_0X_2 \oplus X_0X_3 \oplus X_1X_3 \oplus X_1.
\end{aligned}$$

In [BP12] they found a circuit of depth 4 and 17 XORs, but we would like to shrink the depth even further by utilizing a wider range of standard gates.

We have adapted the algorithm from Section 4.2 to also find a small solution for the `INV` block. The idea is simple; each Y_i is a truth table of length 16 bits, based on a 4-bit input $X_0, \ldots, X_3$. We define our "point" to be a 16-bit value. All standard gates, `AND, OR, XOR, MUX, NOT`, including their negate versions, can be applied to any combination of "known" points (S), and distances to target points T can be computed in a similar manner as before. Using this slightly modified algorithm for floating multiplexers, we found a solution with only 9 gates and depth 3. The results are shown in Equation 2 and Table 2. The full listing of the formulae for scenarios S0-S5 can be found in D.2.

$$\begin{array}{lll}
T0 = \texttt{NAND}(X0, X2) & T3 = \texttt{MUX}(X1, X2, 1) & Y1 = \texttt{MUX}(T2, X3, T3)\\
T1 = \texttt{NOR}(X1, X3) & T4 = \texttt{MUX}(X3, X0, 1) & Y2 = \texttt{MUX}(X0, T2, X1)\\
T2 = \texttt{XNOR}(T0, T1) & Y0 = \texttt{MUX}(X2, T2, X3) & Y3 = \texttt{MUX}(T2, X1, T4)
\end{array} \tag{2}$$

Table 2: Refactored INV block and scenarios S0-S5 .

	INV	S0	S1	S2	S3	S4	S5
Std. area (gates)	9	14	16	17	16	19	19
Std. depth (gates)	3	5	4	4	4	4	3
Tech. area (GE)	18.31	29.96	35.30	39.63	36.62	42.29	44.63
Tech. depth(XORs)	2.31	4.31	3.31	3.77	3.76	3.59	3.11

In our tradeoff circuits we have used scenario S1, as it showed best results with respect to the area and depth. For the bonus circuits, we used S0 as it has the smallest area. For the fast circuit, only the `INV` formulae are needed. We also derived an alternative circuit for the inversion block without multiplexers, the results and formulae are given in B.2.

5.4 Additional Transformation Matrices (ATM)

We are solving the top matrices through exhaustive search and the bottom matrices with various heuristic techniques. The way those matrices look, naturally influence the final number of gates in the solution. Here we present a simple method to try different top and bottom matrices for the best solution.

Assume that the SBox is a black box and, when *excluding* the final addition of the constant, it performs the function:

$$SBox(x) = x^{-1} \cdot A_{8\times 8},$$

where x^{-1} is the inverse element in the Rijndael field $GF(2^8)$, and the matrix $A_{8\times 8}$ is the affine transformation. In any field of characteristic 2: squaring, square root, and multiplication by a constant – are linear functions, thus for a non-trivial choice (α, β) we have:

$$Z(x) = \left(\alpha \cdot x^{2^\beta}\right)^{-1},$$
$$SBox(x) = \sqrt[2^\beta]{\alpha \cdot Z(x)} \cdot A_{8\times 8}.$$

If the initial Top and Bottom matrices for the forward and inverse SBoxes were T_F, B_F, T_I, F_I, respectively, then one can choose any $\alpha = 1, \ldots, 255$ and $\beta = 0, \ldots, 7$, and change the matrices as follows:

$$T'_F = T_F \cdot E \cdot C_\alpha \cdot P_\beta \cdot E,$$
$$B'_F = E \cdot A \cdot P_\beta^{-1} \cdot C_\alpha \cdot A^{-1} \cdot E \cdot B_F,$$
$$T'_I = T_I \cdot E \cdot A \cdot C_\alpha \cdot P_\beta \cdot A^{-1} \cdot E,$$
$$B'_I = E \cdot P_\beta^{-1} \cdot C_\alpha \cdot E \cdot B_I,$$

where:
E – is the 8x8 matrix that switches bits endianness (in our circuits input and output bits are in Big Endian)
A – is the 8x8 matrix that performs the SBox's affine transformation
C_α – is the 8x8 matrix that multiplies a field element by the selected **constant** α
P_β – is the 8x8 matrix that raises an element of the Rijndael field to the **power** of 2^β
T_F/T_I – are the original (without modifications) 18x8 matrixes for the top linear transformation of the Forward/Inverse SBoxes, resp.
B_F/B_I – are the original (without modifications) 8x18 matrixes for the bottom linear transformation of the Forward/Inverse SBoxes, resp.

There are 2040 choices for (α, β) pair and each choice gives new linear matrices. It is easy to test all of them and find the best combination that gives the smallest SBox circuit. We have applied this idea to both the forward as well as the inverse SBox, for both architectures A and D. Note that a similar approach was recently and independently considered in [UHNA19] but in that work they only considered multiplication with a constant, and not squaring.

5.4.1 ATM approach for the combined SBox

For the combined SBoxes we can apply the ATM approach to the forward and the inverse parts independently. This means that we have $2040^2 = 4,161,600$ variants of linear matrices to test. We have focused on the architecture D, since there is no bottom matrix and thus we can do a more extensive search. We searched through all those 4 million variants and applied the heuristic algorithm from the Section 4.1 as a quick analysis method to select a set of around 4000 promising cases. We then applied the algorithm given in Section 4.2 to find a solution with floating multiplexers. In our case we have $n = 8$ input bits and thus each point is encoded with 18 bits, and the complexity of calculating the distance $\delta(S, t_i)$ is quadratic over $N = 2^{18}$ points. In the search we used the search tree with the maximum depth $TD \leq 20$ and the truncation level of 400 leaves.

Table 3: Summary of which algorithms were used to derive the new SBoxes. BM is Bottom Matrix, and TM is Top Matrix.

		Bonus and Tradoff (Arch. A)		Fast (Arch. D, no BM)	
Section	Our contribution	Fwd/Inv	Combined	Fwd/Inv	Combined
3.2 Cancellation-free heuristic	Probabilistic approach with final exhaustive search.	BM	BM + optimization of MUXes by hand		
3.3 Boyar's basic algorithm and [LSL+19]		Not used since probabilistic heuristic with final exhaustive search gave better results.			
3.4 Exhaustive search	New contribution	TM		TM	
4.1 Floating multiplexers (approximative solution)	New contribution				In combination with ATM to select preliminary set from ≈4M choices.
4.2 Generic floating multiplexers	New contribution		TM but applied after a first selection using ATM approach.		TM but applied after a first selection using 4.1+ATM.
5.2 and 5.3 MULN Scenario used	New contribution	S0(Bonus), S1(Tradeoff)	S0(Bonus), S1(Tradeoff)	INV	INV
5.4 Additional Transformation Matrixes	New contribution. A similar approach was independently derived in [UHNA19], but only constant multiplication was considered.	Used	Used	Used	Used

6 Results and comparisons

In this section we present our best solutions for the AES SBox, both forward and combined. The stand-alone inverse SBox is perhaps not as widely used, and those results can be found in Appendix C. We compare our area and depth using the techniques described in Appendix A and where possible, we have recalculated the corresponding GE for other academic results for easier comparison. We present three different solutions for each SBox (forward, inverse, and combined): "fast", "tradeoff", and "bonus". The fast one is the solution with the lowest critical path, the tradeoff solution is a well-balanced trade-off between area and speed, and the bonus solution is given to establish a new record in terms of the smallest number of gates. Exact circuit expressions for all the derived solutions can be found in Appendix D, where we also indicate which algorithm was used in deriving the solution.

6.1 Synthesis results

We have performed a synthesis of the results and compared with other recent academic work. The technology process is GlobalFoundries 22nm CSC20L [Glo19], and we have synthesized using Design Compiler 2017 from Synospys in *topological mode* with the `compile_ultra` command. We also turned on the flag `compile_timing_high_effort` to force the compiler to make as fast circuits as possible. In those graphs, the X axis is the clock period (in ps) and the Y axis is the resulting topology estimated area (in μm^2). We have not restricted the available gates in any way, so the compiler was free to use non-standard gates e.g., a 3 input AND-OR gate. To get the graphs in the following subsections, we have started at a 1200 ps clock period (∼833 MHz) and reduced the clock period by 20 ps until the timing constraints could not be met. We note that the area estimates by the compiler fluctuate heavily, and we believe that this is a result of the many different strategies the compiler has to minimize the depth. One strategy might be successful for say a 700 ps clock period, but a different strategy (which results in a significantly larger area) could be successful for 720 ps. There is also an element of randomness involved in the strategies for the compiler.

Table 4: Forward SBox: Comparison of the results.

Forward SBox	Area Size/Gates		Critical Path/Depth	
	Std. gates	Tech. GE	Std. gates	Tech. XORs
Previous Results				
Canright [Can05]	`80XO+34ND+6NR`		`19XO+3ND+1NR`	
most famous design	120	226.40	23	20.796
Boyar et al [BP12]	`94XO+34AD`		`13XO+3AD`	
our starting point	128	264.24	16	14.932
Boyar et al [Boy]	`81XO+32ND`		`21XO+6ND`	
record smallest	113	220.73	27	23.508
Ueno et al [UHS+15]	`91XO+48ND+13NR (+4IV)`		`10XO+5ND (+1IV)`	
record fastest, formulas from [RMTA18a]	151(+4)	270.71	15(+1)	12.449
Reyhani-Light [RMTA18a]	`69XO+43ND+7NR (+4IV)`		`16XO+4ND (+1IV)`	
at CHES 2018	119(+4)	213.45	20(+1)	18.031
Reyhani-Fast [RMTA18a]	`79XO+43ND+7NR (+4IV)`		`11XO+5ND (+1IV)`	
at CHES 2018	129(+4)	236.75	16(+1)	13.449
Ueno et al [UHNA19]	`90XO+4XN+10OR+45AD (+10IV)`		`11XO+1OR+3AD (+1IV)`	
recent result	149(+10)	298.87	15(+1)	14.131
Our Results				
Forward (fast)	`77XO+1XN+4AD+37ND+5NR+6MX`		`7XO+1XN+1AD+2NR+1MX`	
fast with depth 12	130	243.04	12	10.496
Forward (tradeoff)	`61XO+8XN+27ND+5NR+8MX+2MI`		`8XO+2ND+1ND+2NR+1MX`	
area/speed tradeoff	111	216.75	14	12.263
Forward (bonus)	`58XO+6XN+27ND+5NR+6MX`		`18XO+2XN+1ND+2NR+1MX`	
new record smallest	102	195.10	24	22.263

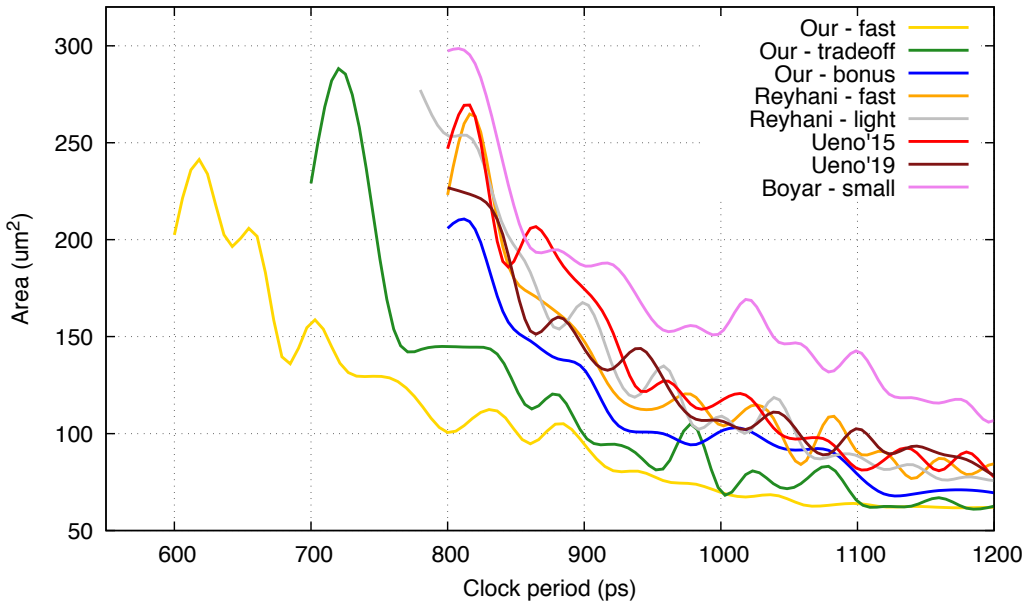

Figure 4: Forward SBox: Synthesis results (the closer the curve is to the axes the better the result in terms of the area/speed trade-off).

Table 5: Combined SBox: Comparison of the results.

Combined SBox	Area Size/Gates		Critical Path/Depth	
	Std. gates	Tech. GE	Std. gates	Tech. XORs
Previous Results				
Canright [Can05]	`94XO+34ND+6NR+16MX (+2IV)`		`20XO+3ND+2OR+5NR`	
most famous design	150(+2)	297.64	30	25.644
Reyhani et al [RMTA18b]	`81XO+32ND+4OR+16NR+16MI (+8IV)`		`17XO+2ND+3OR+6NR`	
	149(+8)	290.13	28	23.608
Ueno et al [UHNA19]	`112XO+7XN+10OR+45AN+16MX (+10IV)`		`11XO+3AN+1OR+2MX (+1IV)`	
recent result	190 (+10)	393.40	17(+1)	15.681
Our Results				
Combined (fast)	`77XO+27XN+41ND+6NR+13MX+12MI`		`6XO+3XN+1ND+2NR+1MX+1MI`	
fast with depth 14	176	351.65	14	12.312
Combined (tradeoff)	`70XO+21XN+27ND+5NR+17MX+5MI`		`7XO+4XN+1ND+2NR+1MX+1MI`	
area/speed tradeoff	145	296.99	16	14.305
Combined (bonus)	`70XO+9XN+27ND+5NR+16MX`		`15XO+4XN+2ND+1NR+3MX`	
new record smallest	127	253.35	25	22.675

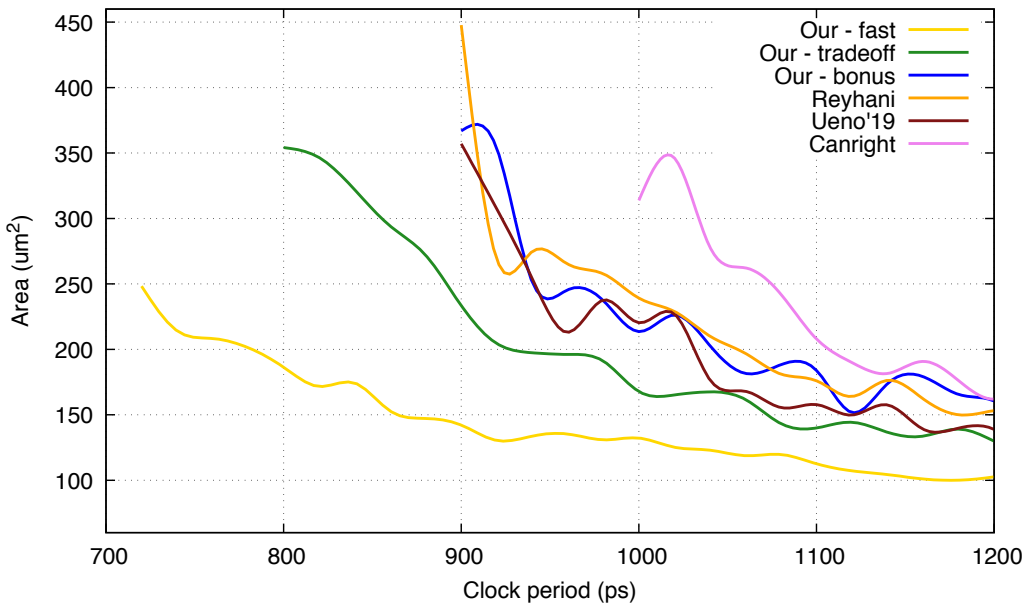

Figure 5: Combined SBox: Synthesis results (the closer the curve is to the axes the better the result in terms of the area/speed trade-off).

6.2 Forward SBoxes

We have included a number of interesting previous results for comparison in Table 4. The most famous design by Canright is widely used and cited. Our tradeoff SBox is both faster and smaller. We also included the work done by Boyar et al as their design was the starting point for our research.

The two results from CHES'18 by Reyhani et al are the most recent, and our tradeoff SBox has a similar area as their "lightweight" version in terms of GE, but around 30% faster. The tradeoff SBox is both smaller and faster than their "fast" circuit. Also, our "fast" version is faster by 25% than their "fast" version, while maintaining a decent area increase. The currently fastest SBox done by Ueno has 270.71GE and 12.449XORs depth, while our fast version is only 243GE with depth 10.496XORs, outperforming the known fastest circuit by around 23%.

We also included the current known smallest circuit (in terms of standard gates) done by Boyar in 2016, which has 113 gates (220.73GE) and depth 27 gates. Our "bonus" circuit is even smaller with only 102 gates and depth 24, reaching as low as 195.10GE. Synthesis results are shown in Figure 4.

6.3 Combined SBoxes

Table 5 shows our results compared to the three previously known best results. Our tradeoff combined SBox has a similar size to that of [Can05] and [RMTA18b], but its speed is a lot faster due to a much lower depth of the circuit. The tradeoff circuit has depth 16 (in reality only 14.305XORs) and 145 gates (297GE), while Canright's combined SBox is of size 150(+2) gates (298GE) and the depth 30 (25.644XORs). The bonus solution in this paper has slightly smaller depth than the most recent result [RMTA18b] but is significantly smaller in size (127 vs 149(+8) standard gates). Finally, the proposed "fast" design using Architecture D has the best currently known depth. Our synthesis results are shown in the comparison Figure 5.

7 Conclusions

In this paper we have introduced a number of heuristic and exhaustive search methods for minimizing the circuit realization of the AES SBox. We have proposed a novel idea on how to include the multiplexers of the combined SBox in the minimization algorithms, and derived smaller and faster circuit realizations for the forward, inverse, and combined AES SBox. We also introduced a new architecture where we remove the "irregular" bottom linear matrix, in order to derive a faster solution than previously known.

Acknowledgements

We would like to thank the Ericsson Research Data Center team for their patience and help with the compute resources that made this work possible, and our colleague Ben Smeets and all reviewers for providing valuable comments to the manuscript.

References

[Art01] Artisan Components, Inc. TSMC 0.18μm Process 1.8-Volt SAGE-XTM Standard Cell Library Databook, 2001. http://www.utdallas.edu/~mxl095420/EE6306/Final%20project/tsmc18_component.pdf.

[BFP18] Joan Boyar, Magnus Find, and René Peralta. Small low-depth circuits for cryptographic applications. *Cryptography and Communications*, 11, 03 2018.

[BHWZ94] Michael Bussieck, Hannes Hassler, Gerhard J. Woeginger, and Uwe T. Zimmermann. Fast algorithms for the maximum convolution problem. *Oper. Res. Lett.*, 15(3):133–141, April 1994. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.5023&rep=rep1&type=pdf.

[BMP08] Joan Boyar, Philip Matthews, and René Peralta. On the shortest linear straight-line program for computing linear forms. In Edward Ochmański and Jerzy Tyszkiewicz, editors, *Mathematical Foundations of Computer Science 2008*, pages 168–179, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

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

[Boy] Joan Boyar. Circuit minimization work. http://www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html.

[BP10a] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms*, pages 178–189, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BP10b] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In *of Lecture Notes in Computer Science*, pages 178–189. Springer, 2010.

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the AES S-Box. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *SEC*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012. https://link.springer.com/chapter/10.1007/978-3-642-30436-1_24.

[Can05] D. Canright. A very compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 441–455, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. https://www.iacr.org/archive/ches2005/032.pdf.

[FAR06] FARADAY Technology Co. FSD0A_A 90 nm Logic SP-RVT (Low-K) Process , 2006. https://www.cl.cam.ac.uk/research/srg/han/ACS-P35/documents/90nm-cell.pdf.

[Glo19] GlobalFoundries. 22nm FDX process, 2019. https://www.globalfoundries.com/technology-solutions/cmos/fdx/22fdx.

[Int01] International Business Machines Corporation. ASIC SA-27E Databook, Part I Base Library and I/Os. Data Book, 2001. http://people.csail.mit.edu/jasonm/nigel/base_06-01.pdf.

[IT88] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^M) using normal bases. *Inf. Comput.*, 78(3):171–177, September 1988. http://dx.doi.org/10.1016/0890-5401(88)90024-7.

[JKL10] Yong-Sung Jeon, Young-Jin Kim, and Dong-Ho Lee. A compact memory-free architecture for the AES algorithm using resource sharing methods. *Journal of Circuits, Systems, and Computers*, 19:1109–1130, 2010.

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

[MNG00] Microelectronics Group, Carl F. Nielsen, and Samuel R. Girgis. WPI 0.5 mm CMOS Standard Cell Library Databook , 2000. https://lsm.epfl.ch/files/content/sites/lsm/files/shared/Resources%20documents/data_book.pdf.

[NNT+10] Yasuyuki Nogami, Kenta Nekado, Tetsumi Toyota, Naoto Hongo, and Yoshitaka Morikawa. Mixed bases for efficient inversion in $\mathbb{F}((2^2)^2)^2$ and conversion matrices of SubBytes of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 234–247. Springer Berlin Heidelberg, 08 2010.

[oST01] National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001.

[Paa97] Christof Paar. Optimized arithmetic for Reed-Solomon encoders. In *Proceedings of IEEE International Symposium on Information Theory*. IEEE, 1997.

[Pet] Graham Petley. Internet resource: VLSI and ASIC Technology Standard Cell Library Design. http://www.vlsitechnology.org/index.html.

[Rij00] Vincent Rijmen. Efficient implementation of the Rijndael S-Box, 2000. https://www.researchgate.net/publication/2621085_Efficient_Implementation_of_the_Rijndael_S-box.

[RMTA18a] Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. Smashing the implementation records of AES S-Box. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):298–336, May 2018.

[RMTA18b] Arash Reyhani-Masoleh, Mostafa M. I. Taha, and Doaa Ashmawy. New area record for the AES combined S-Box/inverse S-Box. *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 145–152, 2018.

[Sam00] Samsung Electronics Co., Ltd. STD90/MDL90 0.35μm 3.3V CMOS Standard Cell Library for Pure Logic/MDL Products Databook, 2000. https://www.digchip.com/datasheets/download_datasheet.php?id=935791&part-number=STD90.

[SMTM01] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with S-Box optimization. In Colin Boyd, editor, *ASIACRYPT*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

[UHNA19] Rei Ueno, Naofumi Homma, Yasuyuki Nogami, and Takafumi Aoki. Highly efficient GF(2^8) inversion circuit based on hybrid GF representations. *Journal of Cryptographic Engineering*, 9(2):101–113, Jun 2019.

[UHS+15] Rei Ueno, Naofumi Homma, Yukihiro Sugawara, Yasuyuki Nogami, and Takafumi Aoki. Highly efficient GF(2^8) inversion circuit based on redundant GF arithmetic and its application to AES design. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2015.

A Area and speed measurement methods

Firstly, we introduce some notations. Gate names are written in capital letters GATE (examples: AND, OR). The notation mGATEn means m gates of type GATE, each of which has n inputs (example: XOR4, 8XOR4, NAND3, 2AND2). When the number of inputs n is missing then the assumption is that the gate has minimum inputs, usually only 2 (3 for MUX).

Cells that are constructed as gates combinations can be described as GATES1-GATE2, meaning that we first perform one or more gates on the first level GATES1, then the result goes to the gate on the second level 2. Example: NAND2-NOR2, means that the cell has 3 inputs (a, b, c) and the corresponding Boolean function is NOR2(a, NAND2(b, c)).

We present two different methods of comparing circuits; the standard method and the technology method.

A.1 Standard method

Cells. The basic elements that are considered in the standard method are:
`{XOR, XNOR, AND, NAND, OR, NOR, MUX, NMUX, NOT}`.

Negotiation of NOT gates. In some places of the circuit there can be a need to use the inverted version of a signal. This can be done in several ways, without the explicit use of a NOT gate. Here we list a few of them.

Method 1. One way to implement a NOT gate is to change the previous gate that generates that signal to instead produce an inverted signal. For example, switch XOR into XNOR, AND into NAND, etc.

Method 2. In several technologies some gates can produce both the straight signal and the inverted version. For example, XOR gates in many implementations produce both signals simultaneously, and thus the inverted value is readily available.

Method 3. We can change the gates following the inverted signal such that the resulting scheme would produce the correct result given the inverted input, using e.g. De Morgan's laws.

Summarizing the above, we believe that NOT gates may be ignored while evaluating a circuit with the standard method, since it can hardly be counted as a full gate. However, for completeness, we will print out the number of NOT gates in the resulting tables.

Area. For area comparisons the number of basic elements is counted without any size distinction between them. The NOT-gates are ignored.

Depth. The depth is counted in terms of the number of basic elements on the circuit path. The overall depth of a circuit is therefore the delay of the critical path. The NOT-gates are ignored.

A.2 Technology method

Cells. Some papers complement the standard cells with a few extra combinatorial cells, often available in various technologies. For example, the gates `NAND2-NAND2, NOR2-NOR2, 2AND2-NOR2, XOR4` could be highly useful to improve and speed up our SBox circuits in this paper. However, for comparison purposes with previous academic results, we will stay with the set of standard cells in order to make a more fair comparison. In this method we do count `NOT` gates in both the delay and the area.

Area. There exist many ASIC technologies (90nm, 45nm, 14nm, etc) from different vendors (Intel, Samsung, GlobalFoundries, etc), with different specifics. In order to develop an ASIC one needs to get a "standard cells library" of a certain technology, and that library usually includes much more versatile cells than the basic elements listed above, so that the designer has a wider choice of building blocks.

However, even if we take a standard cell, for example XOR, then for different technologies that cell has different areas and delays. This makes it harder to compare two circuits of the same logic developed by two academic groups, when they chose to apply different technologies.

For a fair comparison of circuit area of various solutions in academia we usually utilize the term of **gates equivalence (GE)**, where 1GE is the size of the smallest NAND gate. The size of a circuit in GE terms is then computed as Area(Circuit)/Area(NAND) → t GE. Knowing the estimated GE values for each standard or technology cell makes it possible to compute an estimated area size of a circuit in terms of GE. Although various technologies have slightly different GEs for standard cells, those GE numbers are still pretty close to each other.

We have studied several technologies, where data books are available, and came to the decision to utilize GE values given in the data book by the Samsung's STD90/MDL90 0.35μm 3.3V CMOS technology [Sam00]. The cells to be used are without the speed x-factor.

Other data books that we checked include IBM's 0.18μm [Int01], WPI 0.5mm [MNG00], FARADAY's 90μm [FAR06], TSMC 0.18μm [Art01], Web resource [Pet], etc.; we verified that GE numbers given in [Sam00] are quite fair and close to the reality. This makes it possible to have an approximated comparison of the effectiveness of different circuits, even though they may be developed for different technologies.

Depth. Different cells, like XOR and NAND, not only differ in terms of GEs but also differ in terms of the maximum delay of the gates.

Normally data books include the delays (e.g., in ns.) for each gate, and for all input-output combinations.

We propose to normalize the delays of all used gates by the delay of the XOR gate. I.e., we adopt the worst-case delay of the XOR gate as 1 unit in our measurements of the critical path. Then we look at each standard cell and pick the maximum of the switching characteristics for all in-out paths of the cell and divide it by the maximum delay of the XOR gate, so that we get the normalized delay-units for each of the gates utilized.

For multiplexers (MUX and NMUX), we ignore propagation delays for the select bit since in most cases, the select bit is the input to the circuit. For example, in the combined SBox the select bit says if we compute the forward or the inverse SBox, and that selection is ready as an input signal and not switching over the circuit signals propagation, so it can be regarded as a stable signal.

The proposed above method is similar to the idea of GEs, but is adopted for computing the depth of a circuit, normalized in XOR delays. The reason to choose XOR as the base element for delay counting is that circuits often have a lot of XOR gates, and thus, it now becomes possible to compare the depths between the standard and the technology methods as well. For example, in our SBox the critical path contains 14 gates, most of which are XORs, but in reality the depth would be equivalent to only 12.26 XOR-delays, due to the critical path contains also faster gates.

The area and delays for the Samsung's STD90/MDL90 0.35μm gates are summarized in Table 6.

Table 6: Technology gates' area and delays based on [Sam00].

Std. cell	XOR	XNOR	AND	NAND	OR	NOR	MUX	NMUX	NOT	D-Flop/Q
Ref. in [Sam00]	[XO2]	[XN2]	[AD2]	[ND2]	[OR2]	[NR2]	[MX2]	[MX2I]	[IV]	[FD1Q]
Our short ref.	XO	XN	AD	ND	OR	NR	MX	MI	IV	FD
Area (GE)	2.33	2.33	1.33	1.00	1.33	1.00	2.33	2.67	0.67	4.33
Delay (XORs)	1.000	0.993	0.644	0.418	0.840	0.542	0.775	1.056	0.359	1.242

B Algorithmic details and improvements

In this section we present some more details to various algorithms previously described in the paper.

B.1 On the computation of $\delta(S, t_i)$ in Section 4.2.5

In this section we give a more detailed presentation on how the computation of $\delta(S, t_i)$ can be done. A slightly re-organized set of algorithms for computing $\delta(S, t_i)$ is given by Algorithms 2, 3, and 4.

Algorithm 2 Computation of all distances

function DISTANCES2(S, T, $max\delta$) $\to \Delta = \{\delta_i\},\ i = 0, \ldots, m-1$
  Init $\delta_i = \infty$ for $i = 0, \ldots, m-1$
  Init $\forall p : V_0[p] = p.d$ if $p \in S$, otherwise ∞
  Init $k = 0$
  while true do
    while $\exists i : \delta_i = \infty$ and $V_k[t_i] \le t_i.d$ **do** $\delta_i = k$
    if $\forall i : \delta_i < \infty$ **then return** OK
    if $k \ge max\delta$ **then return** FAIL
    $k \leftarrow k + 1$
    Init $\forall p : V_k[p] = \infty$
    for $l \leftarrow \lfloor k/2 \rfloor$ to $k-1$ **do**
      CONVOLUTIONXOR(V_k, V_{k-l-1}, V_l)
      CONVOLUTIONMUX(V_k, V_{k-l-1}, V_l)
      CONVOLUTIONMUX(V_k, V_l, V_{k-l-1})

Algorithm 3 Convolution of XOR gates

function CONVOLUTIONXOR(V, A, B)
  for $a = 0 \ldots 2^{2n+2} - 1$ **do**
    for $b = 0 \ldots 2^{2n+2} - 1$ **do**
      $d = \max\{A[a], B[b]\} + 1$
      $p = a \oplus b$ ▷ XOR(a, b) gate
      if $V[p] > d$ **then** $V[p] = d$
      $p = a \oplus b \oplus (1; 0 \ldots 0; 1; 0 \ldots 0)$ ▷ NXOR(a, b) gate
      if $V[p] > d$ **then** $V[p] = d$

There are two convolution algorithms, for XOR gates and for MUX gates, and they can be performed independently. The MUX-convolution can be done in linear time $O(N)$. We first collect the smallest distances for all possible F-values and I-values independently (each of which has $\sqrt{N}$ possible indexes), then the gate MUX can be applied to any of the combinations, so the convolution is $O(\sqrt{N}^2 = N)$. The XOR-convolution is a bit more complicated and it has quadratic complexity $O(N^2)$ in general case.

Algorithmic improvements. Assume for some S we have already computed all distances $\delta_i = \delta(S, t_i)$. *For each* candidate c from C, we add it to S so that $S' = S \cup c$, then we need to compute all distances $\delta'_i = \delta(S', t_i)$ in order to compute the metrics and decide on which c is good. Note that adding a single candidate c implies $\delta'_i \le \delta_i$ for every target t_i. Therefore, we should modify the algorithm DISTANCES(S', T, $max\delta$) such that we set $max\delta = \max\{\delta_i\} - 1$, and check in the end that if $\delta'_i == \infty$ then $\delta'_i = max\delta$. This

Algorithm 4 Convolution of MUX gates

1: **function** ConvolutionMUX(V, A, B)
2: Set $\forall i = 0, \ldots, 2^{n+1} - 1 : F[i] = I[i] = \infty$
3: **for** $a = 0 \ldots 2^{2n+2} - 1$ **do**
4: Set $f = a \div 2^{n+1}$ ▷ high half of a related to F part
5: Set $i = a \mod 2^{n+1}$ ▷ low half of a related to I part
6: **if** $F[f] > A[a]$ **then then**$F[f] = A[a]$
7: **if** $I[i] > B[a]$ **then then**$I[i] = B[a]$
8: **for** $f = 0 \ldots 2^{n+1} - 1$ **do**
9: **for** $i = 0 \ldots 2^{n+1} - 1$ **do**
10: $d = \max\{F[f], I[i]\} + 1$
11: $p = (f \cdot 2^{n+1} + i)$ ▷ MUX(ZF; f; i) gate
12: **if** $V[p] > d$ **then then**$V[p] = d$
13: $p = p \oplus (1; 0 \ldots 0; 1; 0 \ldots 0)$ ▷ NMUX(ZF; f; i) gate
14: **if** $V[p] > d$ **then then**$V[p] = d$

simple trick helps to avoid the computation of the last vector V_k and effectively speed up the computations by up to x20 times.

Generation of the candidates C involves testing if a candidate is already in C or in S (with the same delay) – those needs to be ignored. To speed up this part we can use a temporary vector $Z[N]$ of length N, where all cells are initialized to ∞, and then for each point s from S we set $Z[s.p] = s.d$. Then, when a new candidate c is generated we simply update the table $Z[c.p] = \min\{c.d, Z[c.p]\}$. In the end we remove S points from Z, and generate C from Z as follow: for all $i = 0, \ldots, N - 1$, if $Z[i] < \infty$ then add a candidate $c = \{.p = i, .d = Z[i]\}$ to C. This way we construct C with unique candidates and also having the smallest depths.

Architectural improvements. MUX(a, b) and MUX(b, a) can be combined in a single MUX-convolution function. In $\max\{d_1, d_2\} + 1$, move the $+1$ operation outside the convolution functions, and do it after the convolutions, instead. $p \oplus \{$`.p=`$[1|\mathbf{0}|1|\mathbf{0}]$, `.d=`$0\}$ is done in order to include gates with negated output; those can be moved outside the convolution functions as well and be performed in the main function Distances() in linear time. This helps to reduce the number of operations in the critical loop of the function ConvolutionXOR(), basically this doubles the speed. When $A = B$, then in ConvolutionXOR() we only need to run b starting from a. When B is not equal to V_0, then ConvolutionXOR() can be done only on the half values of b, since we know that all vectors V_k for $k > 0$ are symmetric in regards to NOT-gates. When $A[a] = \infty$ in ConvolutionXOR() then we do not need to enter the inner loop for b. The same check for $B[b] \neq \infty$ is not justified since it adds an unnecessary branching in the critical loop.

Leveraging SIMD (SSSE3). It is quite clear that ConvolutionMUX() can be easily refactored to utilize SIMD vectorized instructions and, for example, 128-bit registers (SSE). However, it is a bit tricky to find a way how to use SIMD instructions for the function ConvolutionXOR(). First of all, assume each cell $A[a], B[b]$ are all of char type (byte), then we must start b aligned to 16 bytes, since our registers are 128-bit long. Secondly, the result of $p = a \oplus b$ for $a = 0, \ldots, 15 \mod 16$ will end up in a permuted location p, but that permutation would only happen in the low 4 bits. With the help of `_mm_shuffle_epi8()` we can make a permutation of the destination 16-byte block, where the permutation vector only depends on the value of $a \mod 16$ (recall that $b = 0 \mod 16$). Those permutation vectors can be hard coded in a constant table. Other operations within that ConvolutionXOR() are trivial to implement. One could also try to utilize 256-bit long registers, thus speeding up the algorithms even more.

B.1.1 More on ConvolutionXOR()

One can notice that ConvolutionXOR() may be done with the help of the following convolution:

$$V[p] = \sum_{a=0}^{N} A[a] \cdot B[p \oplus a],$$

where the operation $x \cdot y \mapsto \max\{a, b\}$, and $x + y \mapsto \min\{a, b\}$. Thus, we have a convolution to be done in the (min, max)-algebra. One could think of applying Fast Walsh-Hadamard Transform (FWHT) in $O(N \log N)$ but the problem is that that algebra does not have an inverse element.

In [BHWZ94] there is an algorithm "MinConv" that can be converted into our convolution problem, and it is claimed to work "around and in average" $O(N \log N)$ time. The idea behind MinConv is to sort A and B vectors, then we get the smallest delays in the beginning of the vectors A and B. Thus, we can enumerate the $\max\{A[a], B[b]\}$ delays starting from the smallest. Also, we should take care of the indexes while sorting A and B, so that we can find the destination point $p = a \oplus b$. Every point p hit the first time will receive the smallest possible delay, and thus can be skipped later on. The idea is that the predicted number of hits to cover all N points of the result should be around $N \log N$.

We have programmed that but it did not demonstrate a speed up on our input size (n=8, N=2^{18}) and actually performed slower than our SIMD-improved quadratic algorithm, at least on our input size. Also, the above algorithm cannot be parallelized.

B.1.2 ConvolutionXOR() in $O(maxDelay^2 \cdot N \log N)$ time

Usually the delay values stored in V vectors are small. We can rely on that fact in order to develop an algorithm that may be faster than $O(N^2)$.

The idea is simple. Construct two vectors $A_x[]$ and $B_y[]$ such that $A_x[p] = 1$ if $A[p] = x$, otherwise $A_x[p] = 0$, do the same for $B_y[]$. Then compute the convolution of two Boolean vectors A_x and B_y through the classical FWHT transform in $O(N \log N)$. Let $C_d[]$ be the result of that convolution with $d = \max\{x, y\} + 1$. Then we know that if $C_d[p] \neq 0$ then the point p may have the depth d. So we just make a linear loop over $C_d[p]$ and check if $C_d[p] \neq 0$ and $V[p] > d$ then $V[p] = d$. We should repeat the above for all combinations of $x, y = 0, \ldots, maxD$, each step of which has the complexity $O(N \log N)$. The value of $maxDelay$ can also be determined in the beginning of the algorithm linearly. Also note that $maxDelay$ may be different for A and B, so that x and y may have different ranges.

B.1.3 ConvolutionXOR() in $O(|S|^2)$ time

When constructing the vector V_1 from the initial V_0 is it worth to do the classical way and run through pairs of points of S, instead of doing the full scale convolution over N points. However, the number of newly generated points grows very rapidly and this method can only be applied to the very first Vs (in our experiments we have seen some "win" only in V_1, then for further $V_k, k > 1$ we have used our SIMD optimized convolution algorithms).

B.2 Alternative equations for `INV` block

In case we want to avoid multiplexers in the `INV` block then there is an alternative set of equations that we also present in this section. We have considered each expression independently, using a general depth 3 expression:

$$Y_i = ((X_a \ op_1 \ X_b) \ op_5 \ (X_c \ op_2 \ X_d)) \ op_7 \ ((X_e \ op_3 \ X_f) \ op_6 \ (X_g \ op_4 \ X_h)),$$

where X_{a-h} are terms from $\{0, 1, X_0, X_1, X_2, X_3\}$ and op_{1-7} are operators from the set of standard gates `{AND, OR, XOR, NAND, NOR, XNOR}`. Note that the above does not need to have all terms, for example, the expression `AND(x, x)` is simply `x`.

The exhaustive search can be organized as follows. Let us have an object `Term` which consists of a truth table `TT` of length 16 bits, based on the 4 bits $X_0, \ldots, X_3$, and a Boolean function that is associated with the term. We start with the initial set of available terms $T^{(0)} = \{0, 1, X_0, \ldots, X_3\}$, and then construct an expression for a chosen Y_i iteratively. Assume at some step k we have the set of available terms $T^{(k)}$, then the next set of terms and associated expressions can be obtained as:

$$T^{(k+1)} = \{T^{(k)}, T^{(k)} \texttt{ operator } T^{(k)}\},$$

taking care of unique terms. At some step k we will get one or more term(s) whose `TTs` are equal to target `TTs` (Y_is).

Since we could actually get multiple Boolean functions for each Y_i, we should select only the "best" functions following the criteria: there are no `NOT` gates (due to better sharing capabilities), there is a maximum number of gates that can be shared between the 4 expressions for $Y_0, \ldots, Y_3$, and the area/depth in terms of GE is small.

Using this technique, we have found a depth 3, 15 gates solution for the inversion. The equations are given below, where we also provide depth 3 solutions for the additional 5 signals $\{Y_{01}, Y_{23}, Y_{02}, Y_{13}, Y_{00}\}$ such that they can share a lot of gates in the mentioned scenarios S0-S5.

```
Y0  = xnor(     and(X0, X2)              , nand(nand(X1, X2),  xor(X2, X3)))
Y1  =  xor(nand(xor(X2, X3), X1)         ,  nor( and(X0, X2), X3))
Y2  = xnor(     and(X0, X2)              , nand( xor(X0, X1), nand(X0, X3)))
Y3  =  xor(nand(xor(X0, X1), X3)         ,  nor( and(X0, X2), X1))
Y01 = nand(nand(xor(X2, X3), X1)         , nand(nand(X0, X3), X2))
Y23 = nand(nand(xor(X0, X1), X3)         , nand(nand(X1, X2), X0))
Y13 =  xor( nor(and(X0, X2), xnor(X1, X3)),  xor(nand(X0, X3), nand(X1, X2)))
Y02 =  xor(nand(xor(X2, X3), nand(X1, X2)), nand( xor(X0, X1), nand(X0, X3)))
Y00 =  and(nand(and(X0, X2), xnor(X1, X3)),  nor( nor(X0, X2),  and(X1, X3)))
```

Listing 1: `INV` refactored, without multiplexers.

When implementing the above circuits for the scenarios S0-S5, and sharing the gates in a best possible way, we then got the results shown in Table 7.

Table 7: Alternative INV block and scenarios S0-S5.

	INV	S0	S1	S2	S3	S4	S5
Std. area (gates)	15	20	22	23	25	25	29
Std. depth (gates)	3	5	4	4	4	4	3
Tech. area (GE)	23.31	34.96	34.30	40.62	40.62	39.96	43.29
Tech. depth(XORs)	2.42	4.42	3.42	3.54	3.42	2.84	2.54

C Inverse SBoxes

The stand-alone inverse SBox is as far as we know, not used very much. But we provide the comparison with previously known solutions in Table 8.

Table 8: Inverse SBox: Comparison of the results.

Inverse SBox	Area Size/Gates		Critical Path/Depth	
	Std. gates	Tech. GE	Std. gates	Tech. XORs
Previous Results				
Canright [Can05]'05	`81XO+34ND+6NR`		--	
most famous design	121	228.73	25?	?
Boyar et al. [BP12]'12	`93XO+34AD`		`13XO+3AD`	
our starting point	127	261.91	16	14.932
Our Results				
Inverse (fast)	`68XO+10XN+41ND+5NR+6MX`		`7XO+1XN+1ND+2NR+1MX`	
fast with depth 12	130	241.72	12	10.270
Inverse (tradeoff)	`64XO+4XN+27ND+5NR+8MX+2MI (+1IV)`		`9XO+1XN+1ND+2NR+1MX`	
area/speed tradeoff	110(+1)	215.09	14	12.270
Inverse (bonus)	`56XO+7XN+27ND+5NR+6MX (+1IV)`		`19XO+2XN+1ND+2NR+1MX`	
new record smallest	101(+1)	193.44	25	23.263

D Circuits

D.1 Preliminaries

In the below listings we present 9 circuits for the forward, inverse, and combined SBoxes that utilize two architectures A(small) and D(fast).

The used symbols are:

- `##comment` – a comment line
- `@filename` – means that we should include the code from another file `` `filename' ``, the listing of which is then given in this section as well.
- `a ^ b` – is the usual `XOR` gate, other gates are explicitly denoted and taken from the set of `{XNOR, AND, NAND, OR, NOR, MUX, NMUX, NOT}`
- `(a op b)` – where the order of execution (the order of gate connections) is important we specify it by brackets.

The input to all SBoxes are the 8 signals `{U0..U7}` and the output are the 8 signals `{R0..R7}`. The input and output bits are represented in Big Endian bit order. For combined SBoxes the input has additional signals `ZF` and `ZI` where `ZF=1` if we perform the forward SBox and `ZF=0` if inverse, otherwise; the signal `ZI` is the complement of `ZF`. We have tested all the proposed circuits and verified their correctness.

The circuits are divided into sub-programs, according to Figure 3. In Section D.2 we describe the common shared components, and then for each solution we give components (common or specific) for the circuits.

D.2 Shared components

```
# File: mulx.a
T20 = NAND(Q6, Q12)
T21 = NAND(Q3, Q14)
T22 = NAND(Q1, Q16)

T10 = (NOR(Q3, Q14) ^ NAND(Q0, Q7))
T11 = (NOR(Q4, Q13) ^ NAND(Q10, Q11))
T12 = (NOR(Q2, Q17) ^ NAND(Q5, Q9))
T13 = (NOR(Q8, Q15) ^ NAND(Q2, Q17))

X0 = T10 ^ (T20 ^ T22)
X1 = T11 ^ (T21 ^ T20)
X2 = T12 ^ (T21 ^ T22)
X3 = T13 ^ (T21 ^ NAND(Q4, Q13))

# File: 8xor4.d
```

```
R0 = (K0  ^ K1 ) ^ (K2  ^ K3 )
R1 = (K4  ^ K5 ) ^ (K6  ^ K7 )
R2 = (K8  ^ K9 ) ^ (K10 ^ K11)
R3 = (K12 ^ K13) ^ (K14 ^ K15)
R4 = (K16 ^ K17) ^ (K18 ^ K19)
R5 = (K20 ^ K21) ^ (K22 ^ K23)
R6 = (K24 ^ K25) ^ (K26 ^ K27)
R7 = (K28 ^ K29) ^ (K30 ^ K31)
```

Listing 2: MULX/8XOR4: Shared components.

```
# File: inv.a
T0 = NAND(X0, X2)
T1 = NOR(X1, X3)
T2 = XNOR(T0, T1)
Y0 = MUX(X2, T2, X3)
Y2 = MUX(X0, T2, X1)
T3 = MUX(X1, X2, 1)
Y1 = MUX(T2, X3, T3)
T4 = MUX(X3, X0, 1)
Y3 = MUX(T2, X1, T4)

# File: s0.a
@inv.a
Y02 = Y2 ^ Y0
Y13 = Y3 ^ Y1
Y23 = Y3 ^ Y2
Y01 = Y1 ^ Y0
Y00 = Y02 ^ Y13

# File: s1.a
@inv.a
T5  = MUX(X0, T0, X3)
Y23 = MUX(X1, T5, X0)
T6  = NMUX(T3, X2, X3)
Y01 = NMUX(T0, T6, X3)
Y02 = Y2 ^ Y0
Y13 = Y3 ^ Y1
Y00 = Y01 ^ Y23

# File: s2.a
T0  = XNOR(X1, X3)
T1  = OR(X1, X3)
T2  = XOR(X0, X2)
T3  = XOR(T1, T2)
Y0  = MUX(X2, T3, X3)
Y2  = MUX(X0, T3, X1)
T4  = MUX(T2, X3, T0)
Y3  = MUX(T4, X0, X1)
T5  = NMUX(X1, X0, X2)
Y02 = MUX(T0, T2, T5)
T6  = MUX(X0, T0, X1)
Y1  = MUX(T6, X2, X3)
T7  = NMUX(X2, X3, X1)
Y13 = NMUX(T2, T7, T0)
Y23 = Y3 ^ Y2
Y01 = Y1 ^ Y0
Y00 = Y02 ^ Y13

# File: s3.a
T0  = XNOR(X1, X3)
T1  = OR(X1, X3)
T2  = XNOR(X0, X2)
T3  = XNOR(T1, T2)
Y0  = MUX(X2, T3, X3)
Y2  = MUX(X0, T3, X1)
T4  = MUX(T2, T0, X3)
Y3  = MUX(T4, X0, X1)
T5  = MUX(T2, X0, T1)
Y00 = NMUX(T5, T0, T2)
T6  = MUX(T2, T0, X1)
Y1  = MUX(T6, X2, X3)
Y02 = Y2 ^ Y0
Y13 = Y3 ^ Y1
Y23 = Y3 ^ Y2
Y01 = Y1 ^ Y0

# File: s4.a
T0  = NAND(X0, X2)
T1  = NOR(X1, X3)
T2  = NMUX(X2, T1, X3)
Y0  = XOR(T0, T2)
T3  = MUX(X0, T1, X1)
Y2  = XNOR(T0, T3)
Y02 = XNOR(T2, T3)
T4  = MUX(X3, X0, T0)
T5  = MUX(X1, X2, T0)
Y13 = XOR(T4, T5)
T6  = XOR(X3, T0)
Y3  = MUX(T6, X1, X0)
T7  = MUX(X0, T0, X3)
Y23 = MUX(X1, T7, X0)
T8  = NMUX(T0, X1, T1)
Y1  = MUX(T8, X3, X2)
T9  = MUX(X2, T0, X1)
Y01 = MUX(X3, T9, X2)
Y00 = Y02 ^ Y13

# File: s5.a
T0  = XOR(X0, X2)
T1  = XNOR(X1, X3)
T2  = OR(X1, X3)
T3  = XOR(T0, T2)
Y0  = MUX(X2, T3, X3)
Y2  = MUX(X0, T3, X1)
T4  = MUX(X2, T1, X3)
Y3  = MUX(T4, X0, X1)
T5  = MUX(T0, X1, T1)
Y1  = MUX(T5, X2, X3)
T6  = NMUX(T1, T0, X0)
Y00 = NMUX(T3, T6, T1)
T7  = MUX(X0, T0, T1)
Y23 = MUX(X1, T7, X0)
Y13 = NMUX(T5, T6, T7)
T8  = MUX(X1, X0, X2)
Y02 = NMUX(T1, T6, T8)
T9  = MUX(X2, T0, T1)
Y01 = MUX(X3, T9, X2)
```

Listing 3: `INV`/S0-S5: Shared components.

An alternative set of equations for the `INV` block is given in Appendix B.2.

```
# File: muln.a
N0  = NAND(Y01, Q11)
N1  = NAND(Y0 , Q12)
N2  = NAND(Y1 , Q0 )
N3  = NAND(Y23, Q17)
N4  = NAND(Y2 , Q5 )
N5  = NAND(Y3 , Q15)
N6  = NAND(Y13, Q14)
N7  = NAND(Y00, Q16)
N8  = NAND(Y02, Q13)
N9  = NAND(Y01, Q7 )
N10 = NAND(Y0 , Q10)
N11 = NAND(Y1 , Q6 )
N12 = NAND(Y23, Q2 )
N13 = NAND(Y2 , Q9 )
N14 = NAND(Y3 , Q8 )
N15 = NAND(Y13, Q3 )
N16 = NAND(Y00, Q1 )
N17 = NAND(Y02, Q4 )

# File: mull.d
K0  = NAND(Y0, L0 )
K12 = NAND(Y0, L12)
K16 = NAND(Y0, L16)
K20 = NAND(Y0, L20)
```

```
K1  = NAND(Y1, L1 )
K5  = NAND(Y1, L5 )
K9  = NAND(Y1, L9 )
K13 = NAND(Y1, L13)
K17 = NAND(Y1, L17)
K21 = NAND(Y1, L21)
K25 = NAND(Y1, L25)
K29 = NAND(Y1, L29)

K2  = NAND(Y2, L2 )
K6  = NAND(Y2, L6 )
K10 = NAND(Y2, L10)
K14 = NAND(Y2, L14)
K18 = NAND(Y2, L18)
K22 = NAND(Y2, L22)
K26 = NAND(Y2, L26)
K30 = NAND(Y2, L30)

K3  = NAND(Y3, L3 )
K7  = NAND(Y3, L7 )
K11 = NAND(Y3, L11)
K15 = NAND(Y3, L15)
K19 = NAND(Y3, L19)
K23 = NAND(Y3, L23)
K27 = NAND(Y3, L27)
K31 = NAND(Y3, L31)

# File: mull.f
K4  = AND(Y0, L4 )
K8  = AND(Y0, L8 )
K24 = AND(Y0, L24)
K28 = AND(Y0, L28)

# File: mull.i
K4  = NAND(Y0, L4 )
K8  = NAND(Y0, L8 )
K24 = NAND(Y0, L24)
K28 = NAND(Y0, L28)

# File: mull.c
K4  = NAND(Y0, L4 ) ^ ZF
K8  = NAND(Y0, L8 ) ^ ZF
K24 = NAND(Y0, L24) ^ ZF
K28 = NAND(Y0, L28) ^ ZF
```

Listing 4: MULN/MULL: Shared components.

D.3 Forward SBox (fast)

```
# Forward (fast)
@ftop.d
@mulx.a
@inv.a
@mull.f
@mull.d
@8xor4.d

# File: ftop.d
# Exhaustive
↪  search
Z18 = U1  ^ U4
L28 = Z18 ^ U6
Q0  = U2  ^ L28
Z96 = U5  ^ U6
Q1  = U0  ^ Z96
Z160= U5  ^ U7
Q2  = U6  ^ Z160
Q11 = U2  ^ U3
L6  = U4  ^ Z96
Q3  = Q11 ^ L6
Q16 = U0  ^ Q11
Q4  = Q16 ^ U4
Q5  = Z18 ^ Z160
Z10 = U1  ^ U3
Q6  = Z10 ^ Q2
Q7  = U0  ^ U7
Z36 = U2  ^ U5
Q8  = Z36 ^ Q5
L19 = U2  ^ Z96
Q9  = Z18 ^ L19
Q10 = Z10 ^ Q1
Q12 = U3  ^ L28
Q13 = U3  ^ Q2
L10 = Z36 ^ Q7
Q14 = U6  ^ L10
Q15 = U0  ^ Q5
L8  = U3  ^ Q5
L12 = Q16 ^ Q2
L16 = U2  ^ Q4
L15 = U1  ^ Z96
L31 = Q16 ^ L15
L5  = Q12 ^ L31
L13 = U3  ^ Q8
L17 = U4  ^ L10
L29 = Z96 ^ L10
L14 = Q11 ^ L10
L26 = Q11 ^ Q5
L30 = Q11 ^ U6
L7  = Q12 ^ Q1
L11 = Q12 ^ L15
L27 = L30 ^ L10
Q17 = U0
L0  = Q10
L4  = U6
L20 = Q0
L24 = Q16
L1  = Q6
L9  = U5
L21 = Q11
L25 = Q13
L2  = Q9
L18 = U1
L22 = Q15
L3  = Q8
L23 = U0
```

Listing 5: Forward SBox with the smallest delay (fast)

D.4 Forward SBox (tradeoff)

```
# Forward
↪  (tradeoff)
@ftop.a
@mulx.a
@s1.a
@muln.a
@fbot.a

# File: ftop.a
# Exhaustive
↪  search
Z6  = U1  ^ U2
Q12 = Z6  ^ U3
Q11 = U4  ^ U5
Q0  = Q12 ^ Q11
Z9  = U0  ^ U3
Z80 = U4  ^ U6
Q1  = Z9  ^ Z80
Q7  = Z6  ^ U7
Q2  = Q1  ^ Q7
Q3  = Q1  ^ U7
Q13 = U5  ^ Z80
Q5  = Q12 ^ Q13
Z66 = U1  ^ U6
Z114= Q11 ^ Z66
Q6  = U7  ^ Z114
Q8  = Q1  ^ Z114
Q9  = Q7  ^ Z114
Q10 = U2  ^ Q13
Q16 = Z9  ^ Z66
Q14 = Q16 ^ Q13
Q15 = U0  ^ U2
Q17 = Z9  ^ Z114
Q4  = U7

# File: fbot.a
# Probabilistic
↪  heuristic
H0  = N3  ^ N8
H1  = N5  ^ N6
H2  = XNOR(H0, H1)
H3  = N1  ^ N4
H4  = N9  ^ N10
H5  = N13 ^ N14
H6  = N15 ^ H4
H7  = N0  ^ H3
H8  = N17 ^ H5
H9  = N3  ^ H7
H10 = N15 ^ N17
H11 = N9  ^ N11
```

```
H12 = N12 ^ N14
H13 = N1  ^ N2
H14 = N5  ^ N16
H15 = N7  ^ H11
H16 = H10 ^ H11
H17 = N16 ^ H8
H18 = H6  ^ H8
H19 = H10 ^ H12
H20 = N2  ^ H3
H21 = H6  ^ H14
H22 = N8  ^ H12
H23 = H13 ^ H15
R0  = XNOR(H16,
 ↪  H2)
R1  = H2
R2  = XNOR(H20,
 ↪  H21)
R3  = XNOR(H17,
 ↪  H2)
R4  = XNOR(H18,
 ↪  H2)
R5  = H22 ^ H23
R6  = XNOR(H19,
 ↪  H9)
R7  = XNOR(H9 ,
 ↪  H18)
```

Listing 6: Forward SBox circuit with area/depth trade-off (tradeoff)

D.5 Forward SBox (bonus)

We include these bonus circuits just to update the world record for the smallest SBox. The new record is 102 gates with depth 24.

```
# Forward (bonus)
@ftop.b
@mulx.a
@s0.a
@muln.a
@fbot.b

# File: ftop.b
Z24 = U3 ^ U4
Q17 = U1 ^ U7
Q16 = U5 ^ Q17
Q0 = Z24 ^ Q16
Z66 = U1 ^ U6
Q7 = Z24 ^ Z66
Q2 = U2 ^ Q0
Q1 = Q7 ^ Q2
Q3 = U0 ^ Q7
Q4 = U0 ^ Q2
Q5 = U1 ^ Q4
Q6 = U2 ^ U3
Q10 = Q6 ^ Q7
Q8 = U0 ^ Q10
Q9 = Q8 ^ Q2
Q12 = Z24 ^ Q17
Q15 = U7 ^ Q4
Q13 = Z24 ^ Q15
Q14 = Q15 ^ Q0
Q11 = U5

# File: fbot.b
H0 = N1 ^ N5
H1 = N4 ^ H0
R2 = XNOR(N2, H1)
H2 = N9 ^ N15
H3 = N11 ^ N17
R6 = XNOR(H2, H3)
H4 = N11 ^ N14
H5 = N9 ^ N12
R5 = H4 ^ H5
H6 = N16 ^ H2
H7 = R2 ^ R6
H8 = N10 ^ H7
R7 = XNOR(H6, H8)
H9 = N8 ^ H1
H10 = N13 ^ H8
R3 = H5 ^ H10
H11 = H9 ^ H10
H12 = N7 ^ H11
H13 = H4 ^ H12
R4 = N1 ^ H13
H14 = XNOR(N0, R7)
H15 = H9 ^ H14
H16 = H7 ^ H15
R1 = XNOR(N6, H16)
H17 = N4 ^ H14
H18 = N3 ^ H17
R0 = H13 ^ H18
```

Listing 7: Forward SBox circuit with the smallest number of gates (bonus)

D.6 Combined SBox (fast)

```
# Combined (fast)
@ctop.d
@mulx.a
@inv.a
@mull.c
@mull.d
@8xor4.d

# File: ctop.d
# Floating multiplexers
A0 = XNOR(U2, U4)
A1 = XNOR(U1, A0)
A2 = XNOR(U5, U7)
A3 = U0 ^ U5
A4 = XNOR(U3, U6)
A5 = U2 ^ U6
A6 = NMUX(ZF, A4, U1)
Q11 = A5 ^ A6
Q16 = U0 ^ Q11
A7 = U3 ^ A1
L24 = MUX(ZF, Q16, A7)
A8 = NMUX(ZF, A3, U6)
L5 = A0 ^ A8
L11 = Q16 ^ L5
A9 = MUX(ZF, U2, U6)
A10 = XNOR(A2, A9)
Q5 = A1 ^ A10
Q15 = U0 ^ Q5
A11 = U2 ^ U3
A12 = NMUX(ZF, A2, A11)
Q13 = A6 ^ A12
Q12 = Q5 ^ Q13
A13 = A5 ^ A12
Q0 = Q5 ^ A13
Q14 = U0 ^ A13
A14 = XNOR(U3, A3)
A15 = NMUX(ZF, A0, U3)
A16 = XNOR(U5, A15)
Q3 = A4 ^ A16
L6 = Q11 ^ Q3
A17 = U2 ^ A10
Q7 = XNOR(A8, A17)
A18 = NMUX(ZF, A14, A2)
Q1 = XNOR(A4, A18)
Q4 = XNOR(A16, A18)
L7 = Q12 ^ Q1
L8 = Q7 ^ L7
A19 = NMUX(ZF, U1, A4)
A20 = XNOR(U6, A19)
Q9 = XNOR(A16, A20)
Q10 = A18 ^ A20
L9 = Q0 ^ Q9
A21 = U1 ^ A2
A22 = NMUX(ZF, A21, A5)
Q2 = A20 ^ A22
Q6 = XNOR(A4, A22)
Q8 = XNOR(A16, A22)
A23 = XNOR(Q5, Q9)
L10 = XNOR(Q1, A23)
L4 = Q14 ^ L10
A24 = NMUX(ZF, Q2, L4)
```

```
L12 = XNOR(Q16, A24)
L25 = XNOR(U3, A24)
A25 = MUX(ZF, L10, A3)
L17 = U4 ^ A25
A26 = MUX(ZF, A10, Q4)
L14 = L24 ^ A26
L23 = A25 ^ A26
A27 = MUX(ZF, A1, U5)
L30 = Q12 ^ A27
A28 = NMUX(ZF, L10, L5)
L21 = XNOR(L14, A28)
L27 = XNOR(L30, A28)
A29 = XNOR(U5, L4)
L29 = A28 ^ A29
L15 = A19 ^ A29
A30 = XNOR(A3, A10)
L18 = NMUX(ZF, A19, A30)
A31 = XNOR(A7, A21)
L16 = A25 ^ A31
L26 = L18 ^ A31
A32 = MUX(ZF, U7, A5)
L13 = A7 ^ A32
A33 = NMUX(ZF, A15, U0)
L19 = XNOR(L6, A33)
A34 = NOR(ZF, U6)
L20 = Q0 ^ A34
A35 = XNOR(A4, A8)
L28 = XNOR(L7, A35)
A36 = NMUX(ZF, Q6, L11)
L31 = A30 ^ A36
A37 = MUX(ZF, L26, A0)
L22 = Q16 ^ A37
Q17 = U0
L0 = Q10
L1 = Q6
L2 = Q9
L3 = Q8
```

Listing 8: Combined SBox circuit with the smallest delay

D.7 Combined SBox (tradeoff)

```
# Combined (tradeoff)
@ctop.a
@mulx.a
@s1.a
@muln.a
@cbot.a

# File: ctop.a
# Floating multiplexers
A0 = XNOR(U0, U6)
Q1 = XNOR(U1, ZF)
A1 = U2 ^ U5
A2 = XNOR(U3, U4)
A3 = XNOR(U3, U7)
A4 = MUX(ZF, A2, U2)
A5 = A0 ^ A1
Q6 = A4 ^ A5
A6 = XNOR(Q1, A1)
A7 = NMUX(ZF, U0, A3)
Q4 = A5 ^ A7
Q3 = Q1 ^ Q4
A8 = NMUX(ZF, U6, A2)
A9 = Q1 ^ A3
Q9 = A8 ^ A9
Q10 = Q4 ^ Q9
A10 = XNOR(A4, A7)
Q7 = XNOR(Q9, A10)
Q8 = XNOR(Q1, A10)
A11 = XNOR(U0, U2)
Q0 = ZF ^ A11
A12 = U1 ^ U3
A13 = A1 ^ A12
A14 = MUX(ZF, A13, A11)
Q15 = U4 ^ A14
A15 = NMUX(ZF, U5, A0)
Q5 = XNOR(A14, A15)
Q17 = XNOR(U4, A15)
A16 = MUX(ZF, A5, A2)
Q16 = XNOR(A13, A16)
A17 = A3 ^ A8
Q2 = XNOR(A10, A17)
A18 = U4 ^ U6
A19 = U1 ^ U2
Q11 = Q6 ^ A19
A20 = MUX(ZF, A18, A19)
Q13 = U5 ^ A20
A21 = XNOR(U4, Q0)
Q14 = XNOR(A14, A21)
A22 = XNOR(A4, A6)
Q12 = XNOR(U6, A22)

# File: cbot.a
# Probabilistic heuristic
H1 = N1 ^ N3
H3 = N15 ^ N17
H4 = N12 ^ N13
H5 = N0 ^ H1
H6 = N7 ^ N8
H8 = N10 ^ N11
H9 = H4 ^ H8
S4 = H3 ^ H9
H10 = N12 ^ N14
H11 = N16 ^ H8
S14 = N17 ^ H11
H12 = N1 ^ N2
H13 = N3 ^ N5
H14 = N4 ^ N5
H15 = N9 ^ N11
H16 = N6 ^ H13
H17 = H6 ^ H14
H18 = N4 ^ H5
H30 = H18 ^ ZF
S1 = H17 ^ H30
H19 = H3 ^ H15
S6 = XNOR(H18, H19)
S11 = H17 ^ H19
H20 = H10 ^ H15
S0 = XNOR(S6, H20)
S5 = H17 ^ H20
H21 = N7 ^ H12
H22 = H16 ^ H21
S12 = H20 ^ H22
S13 = S4 ^ H22
H23 = N15 ^ N16
H24 = N9 ^ N10
H25 = N8 ^ H24
H26 = H12 ^ H14
S7 = XNOR(S4, H26)
H27 = H4 ^ H23
S2 = H30 ^ H27
H28 = N8 ^ H16
S3 = S14 ^ H28
H29 = H21 ^ H25
S15 = H23 ^ H29

R0 = S0
R1 = S1
R2 = S2
R3 = MUX(ZF, S3, S11)
R4 = MUX(ZF, S4, S12)
R5 = MUX(ZF, S5, S13)
R6 = MUX(ZF, S6, S14)
R7 = MUX(ZF, S7, S15)
```

Listing 9: Combined SBox circuit with a good area/depth trade-off (tradeoff)

D.8 Combined SBox (bonus)

```
# Combined (bonus)
@ctop.b
@mulx.a
@s0.a
@muln.a
@cbot.b

# File: ctop.b
# Floating multiplexers
A0 = XNOR(U3, U6)
Q15 = XNOR(U1, ZF)
A1 = U5 ^ Q15
A2 = U2 ^ A0
A3 = U4 ^ A1
A4 = U4 ^ U6
A5 = MUX(ZF, A2, A4)
Q4 = XNOR(A3, A5)
Q0 = U0 ^ Q4
Q14 = Q15 ^ Q0
A6 = XNOR(U0, U2)
Q3 = ZF ^ A6
Q1 = Q4 ^ Q3
A7 = MUX(ZF, U1, Q0)
Q6 = XNOR(A5, A7)
Q8 = Q3 ^ Q6
A8 = MUX(ZF, Q1, A4)
Q9 = U6 ^ A8
Q2 = Q8 ^ Q9
Q10 = Q4 ^ Q9
Q7 = Q6 ^ Q10
A9 = MUX(ZF, A0, U4)
Q12 = XNOR(U7, A9)
Q11 = Q0 ^ Q12
A10 = MUX(ZF, A6, Q12)
A11 = A2 ^ A10
A12 = A4 ^ A11
Q5 = Q0 ^ A12
Q13 = Q11 ^ A12
Q17 = Q14 ^ A12
Q16 = Q14 ^ Q13

# File: cbot.b
H0 = N9 ^ N10
H1 = N16 ^ H0
H2 = N4 ^ N5
S4 = N7 ^ (N8 ^ H2)
H4 = N0 ^ N2
H6 = N15 ^ H1
H7 = H4 ^ (N3 ^ N5)
H20= H6 ^ ZF
S2 = H20 ^ H7
S14 = S4 ^ H7
H8 = N13 ^ H0
H9 = N12 ^ H8
S1 = H20 ^ H9
H10 = N17 ^ H1
H12 = H2 ^ (N1 ^ N2)
S0 = H6 ^ H12
S5 = N6 ^ (H9 ^ (N8 ^
↪  H4))
S11 = H12 ^ S5
S6 = S1 ^ S11
H15 = N14 ^ H10
H16 = H8 ^ H15
S12 = S5 ^ H16
S7 = XNOR(S4, H10 ^ (N9 ^
↪  N11))
H19 = XNOR(H7, S7)
S3 = H16 ^ H19
S15 = S11 ^ H19
S13 = S4 ^ (N12 ^ H15)
R0 = S0
R1 = S1
R2 = S2
R3 = MUX(ZF, S3, S11)
R4 = MUX(ZF, S4, S12)
R5 = MUX(ZF, S5, S13)
R6 = MUX(ZF, S6, S14)
R7 = MUX(ZF, S7, S15)
```

Listing 10: Combined SBox circuit with the smallest number of gates (bonus)

D.9 Inverse SBox (fast)

```
# Inverse (fast)
@itop.d
@mulx.a
@inv.a
@mull.i
@mull.d
@8xor4.d

# File: itop.d
# Exhaustive
↪  search
Q8  = XNOR(U1, U3)
Q0  = Q8  ^ U5
Q1  = U6  ^ U7
Q7  = U3  ^ U4
Q2  = Q7  ^ Q1
Q3  = U0  ^ U4
Q4  = Q3  ^ Q1
Q5  = XNOR(U1, Q3)
Q10 = XNOR(U0, U1)
Q6  = Q10 ^ Q7
Q9  = Q10 ^ Q4
L12 = U4  ^ U5
Z132= U2  ^ U7
Q11 = L12 ^ Z132
Q12 = Q0  ^ Q11
L27 = U3  ^ Z132
Q13 = U0  ^ L27
Q14 = XNOR(Q10,
↪  U2)
Q15 = Q14 ^ Q0
Q16 = XNOR(Q8, U7)
Q17 = Q16 ^ Q11
L23 = Q15 ^ Z132
L0  = U0  ^ L23
L3  = Q11 ^ Q2
L4  = Q6  ^ L3
L16 = Q3  ^ L27
L1  = XNOR(U2, U3)
L6  = L1  ^ Q0
L20 = L6  ^ Q2
L15 = XNOR(U2, Q6)
L24 = U0  ^ L15
L5  = L27 ^ Q2
L19 = Q14 ^ U5
L26 = Q3  ^ L3
L13 = L19 ^ L26
L17 = U0  ^ L12
L21 = XNOR(U1, Q1)
L25 = Q5  ^ L3
L14 = U3  ^ Q12
L18 = U0  ^ Q1
L22 = XNOR(Q5, U6)
L8  = Q11
L28 = Q7
L9  = Q12
L29 = Q10
L2  = U5
L10 = Q17
L30 = Q2
L7  = U4
L11 = Q5
L31 = Q9
```

Listing 11: Forward SBox with the smallest delay (fast)

D.10 Inverse SBox (tradeoff)

```
# Inverse
 ↪  (tradeoff)
@itop.a
@mulx.a
@s1.a
@muln.a
@ibot.a

# File: itop.a
# Exhaustive
 ↪  search
Z20 = U2  ^ U4
Z129= U0  ^ U7
Q0  = Z20 ^ Z129
Q4  = U1  ^ Z20
Z66 = U1  ^ U6
Q3  = U3  ^ Z66
Q1  = Q4  ^ Q3
Q2  = U6  ^ Z129
Z40 = U3  ^ U5
Z132= U2  ^ U7
Q6  = Z40 ^ Z132
Q5  = U0  ^ Q6
Q7  = U3  ^ Q0
Q17 = Z66 ^ Z132
Q8  = U5  ^ Q17
Z33 = U0  ^ U5
Q10 = U4  ^ Z33
Q9  = Q4  ^ Q10
Q12 = XNOR(U4,
 ↪  Z129)
Q13 = XNOR(Z20,
 ↪  Z40)
Q16 = XNOR(Z66,
 ↪  U7)
Q14 = Q13 ^ Q16
Q15 = Z33 ^ Q3
Q11 = NOT(U2)

# File: ibot.a
# Probabilistic
 ↪  heuristic
H0  = N2  ^ N14
H1  = N1  ^ N5
H2  = N10 ^ N11
H3  = N13 ^ H0
H4  = N16 ^ N17
H5  = N1  ^ H2
H6  = N4  ^ H1
H7  = N0  ^ H4
H8  = N15 ^ N16
H9  = N9  ^ N10
H10 = N6  ^ N8
H11 = H3  ^ H6
H12 = N7  ^ N12
H13 = N8  ^ H0
H14 = N3  ^ N5
H15 = H5  ^ H8
H16 = N6  ^ N7
H17 = H12 ^ H13
H18 = H5  ^ H16
H19 = H3  ^ H10
H20 = H10 ^ H14

R0 = H7 ^ H18
R1 = H7 ^ H19
R2 = H2 ^ H11
R4 = H8 ^ H9
R3 = R4 ^ H20
R5 = N2 ^ H6
R6 = H15^ H17
R7 = H4 ^ H11
```

Listing 12: Inverse SBox circuit with good area/depth trade-off (tradeoff)

Note: the above 'NOT(U2)' in the file 'itop.a' is removable by setting Q11=U2 and accurately negating some of the gates and variables downwards where Q11 is involved. For example, the variable Y01 should be negated as well due to: N0 = NAND(Y01, Q11); consequently, all gates involving Y01 should be negated, leading to negation of other Q variables, and so on.

D.11 Inverse SBox (bonus)

```
# Inverse (bonus)
@itop.b
@mulx.a
@s0.a
@muln.a
@ibot.b

# File: itop.b
Z33 = U0 ^ U5
Z3 = U0 ^ U1
Q1 = XNOR(Z3, U3)
Q16 = XNOR(Z33,
 ↪  U6)
Q17 = XNOR(U1,
 ↪  Q16)
Q8 = U4 ^ Q17
Q3 = XNOR(U2, Z33)
Q4 = Q1 ^ Q3
Q15 = XNOR(U4, U7)
Q10 = U3 ^ Q15
Q9 = Q4 ^ Q10
Q2 = Q8 ^ Q9
Q7 = Q1 ^ Q2
Q0 = Z33 ^ Q7
Q5 = Q17 ^ Q15
Q6 = Q3 ^ Q8
Q12 = XNOR(U1, Q0)
Q14 = Q15 ^ Q0
Q13 = Q16 ^ Q14
Q11 = NOT(U1)

# File: ibot.b
H0 = N4 ^ N5
H1 = N1 ^ N2
R6 = H0 ^ H1
H2 = N13 ^ N14
H3 = R6 ^ H2
H4 = N17 ^ H3
R0 = N16 ^ H4
H5 = N15 ^ H4
H6 = N10 ^ N11
R3 = H3 ^ H6
H7 = N9 ^ H5
R5 = N10 ^ H7
H8 = N8 ^ H0
H9 = N6 ^ H8
H10 = N7 ^ R3
H11 = N1 ^ R0
H12 = N0 ^ H11
R2 = H9 ^ H12
H13 = H8 ^ H10
R1 = R2 ^ H13
H14 = H5 ^ H13
H15 = N13 ^ H14
R7 = N12 ^ H15
H16 = N4 ^ H9
H17 = R5 ^ H16
R4 = N3 ^ H17
```

Listing 13: Inverse SBox circuit with the smallest number of gates (bonus)