Published in final edited form as:
Cryptogr Commun. 2019 ; 11: . doi:10.1007/s12095-018-0296-3.

Small Low-Depth Circuits for Cryptographic Applications

Joan Boyar[1], **Magnus Gausdal Find**[2], and **René Peralta**[2]

[1]Department of Mathematics and Computer Science University of Southern Denmark, joan@imada.sdu.dk

[2]Information Technology Laboratory, National Institute of Standards and Technology, rene.peralta@nist.gov

Abstract

We present techniques to obtain small circuits which also have low depth. The techniques apply to typical cryptographic functions, as these are often specified over the field $GF(2)$, and they produce circuits containing only AND, XOR and XNOR gates. The emphasis is on the linear components (those portions containing no AND gates). A new heuristic, DCLO (for depth-constrained linear optimization), is used to create small linear circuits given depth constraints. DCLO is repeatedly used in a See-Saw method, alternating between optimizing the upper linear component and the lower linear component. The depth constraints specify both the depth at which each input arrives and restrictions on the depth for each output.

We apply our techniques to cryptographic functions, obtaining new results for the S-Box of the Advanced Encryption Standard, for multiplication of binary polynomials, and for multiplication in finite fields. Additionally, we constructed a 16-bit S-Box using inversion in $GF(2^{16})$ which may be significantly smaller than alternatives.

Keywords

circuit size; circuit depth; cryptographic functions; Boolean functions; See-Saw Method; depth-constrained circuit optimization

1 Introduction

Constructing optimal combinational circuits is an intractable problem under almost any meaningful metric (gate count, depth, energy consumption, etc.). In practice, no known techniques can reliably find optimal circuits for functions with as few as eight Boolean inputs and one Boolean output (there are 2^{256} such functions). Thus, heuristic or specialized techniques are necessary in practice.

Reducing the number of gates is important for reducing area and power consumption. Reducing depth, i.e. the number of gates on a longest path, leads to faster circuits. However, obvious ways to reduce depth lead to an explosion in size. In this paper we reduce size and depth simultaneously.

2 Combinational circuit optimization

Many different logically complete bases are possible for circuits. Since the operations in the basis (XOR, AND) are equivalent to addition and multiplication modulo 2 (i.e., in $GF(2)$), much work on circuits for cryptographic functions uses this basis. For logical completeness, we use the basis (XOR,AND,XNOR), although most of the paper uses only (XOR,AND). This platform-independent basis leads to easy comparison with previous results.

Classic results by Shannon [17] and Lupanov [10] show that almost all predicates on n bits have circuit complexity about $\frac{2^n}{n}$.The multiplicative complexity of a function is the number of AND gates necessary and sufficient to compute the function. Analogous to the Shannon-Lupanov bound, it was shown in [12, 3] that almost all Boolean predicates on n bits have multiplicative complexity about$2^{\frac{n}{2}}$. Strictly speaking, these theorems say nothing about the class of functions with polynomial circuit complexity. However, it is reasonable to expect that, in practice, the multiplicative complexity of these functions is significantly smaller than their Boolean complexity. This is one of the principles that guide our design strategy.

Circuits with few AND gates will naturally have large sections which are purely linear, i.e., contain no AND gates. Boyar and Peralta [2] and Courtois et al. [7] have used this insight to construct circuits much smaller than previously known for a variety of applications (see [16]). Both of those papers use a twostep process which first reduces multiplicative complexity and then optimizes linear components. The second of these steps involves solving a problem which is NP-hard and MAX-SNP hard [2], implying limits to its approximability. Early published heuristics for this step [15, 1, 2] do not consider depth. We do so here, and obtain circuits that are smaller in both size and depth for several functions of interest to cryptography. In particular, we improved on the results in [5, 14] for the S-Box of the Advanced Encryption Standard (AES).

We note that our results should not be interpreted as trading AND gates for XOR gates. We typically are able to produce circuits which have fewer XOR gates, fewer AND gates, and smaller depth than previously published circuits for the same functions.

3 Algorithm to find small low-depth circuits

We can consider a circuit as a directed acyclic graph where the nodes are either gates or inputs of fan-in zero. Nodes for gates that produce circuit outputs are referred to as “outputs”. The *depth* of a node $X = A\ op\ B$ is

$$depth(X) = 1 + max\{depth(A), depth(B)\}$$

where *op* is a binary operation (AND,XOR,XNOR). Note that this definition allows us to assign arbitrary depths to the input nodes, though nodes which are inputs to the entire circuit are always assigned depth zero.

3.1 The Depth-Constrained Linear Optimization (DCLO) problem

We consider a linear component of a circuit as a set of functions with fixed depths associated to the input variables and depth constraints associated to the outputs. This problem is best represented as a matrix with *input depth* constraints associated with columns and *goal depth* constraints associated with rows. The optimization problem is to find a circuit that satisfies the constraints and minimizes the number of gates. We call this problem *DCLO* (for Depth-Constrained Linear Optimization). For example, the matrix of Figure 1 represents the problem of computing the four functions $\{y_i \mid i = 1,... 4\}$ given by

$$
\begin{aligned}
y_1 &= x_1 + x_3 + x_4 \\
y_2 &= x_2 + x_3 + x_4 \\
y_3 &= x_1 + x_2 + x_3 + x_4 \\
y_4 &= x_1 + x_2 + x_4
\end{aligned}
$$

(recall that addition is modulo 2).

The column heading $x_i : d_i$ states that input x_i has input depth d_i. The row heading $y_i : t_i$ states that a solution must compute function y_i at depth no more than t_i.

For example, the straight-line program

$$
\begin{aligned}
t_1 &= x_1 + x_3 \\
y_1 &= t_1 + x_4
\end{aligned}
$$

is not allowedbecaue the depth of y_1 is 3. A valid straight-line program is

$$
\begin{aligned}
t_1 &= x_1 + x_4 \\
y_1 &= t_1 + x_3 \\
y_3 &= y_1 + x_2 \\
t_2 &= x_3 + x_4 \\
y_2 &= t_2 + x_2 \\
y_4 &= t_1 + x_2
\end{aligned}
$$

3.2 The See-Saw Method

We now describe the method to find a small circuit given an overall depth constraint *TargetD*. A node is *upper linear* if it is an input to the circuit or if it is a linear gate and all its ancestors are upper linear. A node is *lower linear* if it is either an output gate to the circuit or if it is a linear gate and all its descendants are lower linear. The upper linear nodes define a circuit whose inputs are the inputs to the original circuit. We call this circuit the *upper linear component*. The lower linear nodes define a circuit whose outputs are outputs of the original circuit. We call this circuit the *lower linear component*. This naturally decomposes a nonlinear circuit into the upper and lower linear components, plus a *middle nonlinear component*. The outputs of both upper linear and lower linear nodes are linear functions of their respective inputs. The depth of an input node in the upper linear circuit is defined to be

0. The depth of an input node in the lower linear circuit is the depth of the corresponding node in the original circuit.

Our overall approach uses the observation that one can obtain smaller lowdepth circuits by repeatedly optimizing the upper and lower linear components of the circuit based on gate-depth information from previous optimizations. This leads to what we call the *See-Saw Method*. The middle nonlinear component is assumed to be optimized already and is not changed. The technique can be generalized for use with more components without much difficulty.

We alternate restructuring the upper linear and lower linear components until there is no further improvement in size or depth. When one of these linear components is being restructured, the other is fixed. Each restructuring step is an instance of the DCLO problem described in the previous section. It is solved using a heuristic, DCLO, which we define later.

The process starts with the upper linear component. The input depth constraints for the upper linear component are always set to 0, and initially the goal depth constraints are set to the minimum feasible. The minimum feasible goal depths for the upper linear component are calculated as follows: If the Hamming weight (number of 1s) of the row corresponding to the output is w, at least depth $\log_2(w)$ is necessary. This *required depth* can be achieved by placing the XOR gates in a balanced binary tree. Starting with these required depths allows us to jump-start the process and also will give us a lower bound on the depth of any solution that does not restructure the middle component.

After finding a new circuit for the upper linear component, we replace the subcircuit for this upper linear component with the new circuit. We thus create a new circuit, possibly with lower depth than the original. Now the top linear component, together with the nonlinear component, are fixed while we apply DCLO to the bottom linear component. For the lower linear component, if we know that some inputs are available at lower depth than others, this slack may help in creating an implementation with fewer gates. The input depths of the inputs to the lower linear component are set to the depth at which these inputs are calculated in the portion of the circuit which we have fixed. The goal depths of all outputs of the lower linear component are set to TargetD (if this depth is not feasible, the algorithm aborts).

The new circuit for the lower linear component replaces the old subcircuit. For the upper linear component, we may now be able to allow some of its outputs to be computed at a larger depth than in the previous phase. This allows us to force the DCLO to be more strict for some outputs which are critical for the total depth of the circuits and allows us to be less strict for others. If, for example, an output, y_i, of the upper linear component requires depth w_1 because of its Hamming weight, but is not used before depth $w_2 > w_1$ in the entire circuit, it might be possible to create a circuit using fewer gates if output y_i is allowed to be computed at depth w_2. Starting with the second iteration of See-Saw, we calculate these allowed depths as follows: the *height* of a node v, denoted by *height*(v), is the length of the longest path from the node to an output of the entire circuit. If v is an output node of the

upper linear component, then we set its goal depth to $TargetD - height(v)$. Note that this goal depth is at least as large as the required depth.

Note that, for the lower linear component, using the variable, calculated input depths is important to get the minimal depth, even if one is unconcerned about size. One might assume that if minimum depth circuits are found for the upper linear and lower linear components (where one can also assume that the circuit for the upper linear component satisfies the goal depths for each of its outputs), then attaching them to the middle nonlinear component would always give the smallest depth circuit (given the fixed middle component). However, this is not always true. For the AES S-Box, one of the outputs of the lower linear component has twelve 1s, so the minimum depth circuit computing it is not a complete binary tree. Some of the inputs to this circuit from the nonlinear component can be at higher depth than others, still allowing a depth 16 circuit. However, if the wrong inputs are combined first, the total depth can become 17.

To summarize, with the See-Saw Method, we alternate between improving the upper and lower linear components, updating the values for the goal depths and the input depths after each improvement. After the first iteration, the goal depth has been achieved, so the goal is to reduce the number of gates.

3.3 Paar's Algorithm

The heuristic DCLO used within the See-Saw Method uses ideas from a well-known algorithm due to Paar [15]. Paar's technique keeps a list of the variables already computed, which is initially only the inputs. Then it repeatedly determines which two variables, XORed together, occur in most outputs. One such pair is selected and XORed together. This result is added as a new variable which appears in all outputs where both variables previously appeared. This is repeated until all required outputs have been computed. Paar's technique is implemented by starting with the initial matrix with input columns corresponding to the inputs and rows corresponding to the outputs which the circuit should calculate. The algorithm adds columns corresponding to the new variables which are computed. When a new column is added, this corresponds to adding two existing variables, u and v. In all rows in the matrix which currently have a one in both of the columns corresponding to u and v, those two 1s are changed to 0s, and a one is placed in the corresponding row of the new column. All other values in the new column are set to 0. This operation, adding a new column to the matrix, corresponding to a new XOR gate in the circuit, is called a *Paar-like operation*. The 1s in a row indicate which variables still need to be added together to produce that output. The algorithm terminates when all rows have Hamming weight 1.

Cancellation occurs in a circuit when the inputs to an XOR gate are of the form $(f + g, h + g)$. The XOR gate in that case computes the function $f + h$, "cancelling" the term g. In addition to being oblivious to depth, Paar's algorithm incurs a significant cost in size due to the fact that it can only produce circuits which do not allow cancellation.

Proposition 1.Paar-like operations are cancellation-free.

Proof.: We show by induction that the 1s in any row represent sums modulo 2 of disjoint sets of variables. Initially, each 1 represents a single input variable, and they are all distinct. When a Paar-like operation occurs, 1s representing two disjoint sets are removed and the new column represents the union of those two sets. Since the two sets were disjoint, no cancellation occurs in that operation, and the new set is disjoint from all other sets represented by 1s in that row. ┼

It is shown in [4] that non-cancellation can increase the size of a circuit by a factor $\Omega(n/\log^2 n)$. The techniques described in the following sections allow both depth restriction and cancellation.

3.4 Randomized construction heuristic, RAND-GREEDY-ALG

RAND-GREEDY-ALG, our first algorithm for finding small programs with depth restrictions, is a randomized version of Paar's algorithm. It keeps track of the depths of the gates and only adds gates if the global depth restrictions can be satisfied.

Recall that the input to the algorithm is a matrix representing the linear combinations (outputs) to be computed in a linear component, plus the input depths of each of the inputs and the goal depths of each output. For each column, c, let $v(c)$ be a bit vector indicating which variables are present in the linear combination computed by column c. Initially, each column represents a single input variable, and each vector has exactly one 1. When creating a new gate for adding columns c_1, c_2 in the matrix, a new column c_3 is created with $v(c_3)$ set to the symmetric difference of $v(c_1)$ and $v(c_2)$. For the subset of the rows where the new gate is used, the bits in positions c_1 and c_2 are flipped and the bit in position c_3 is set to 1 (for the other rows, it is set to 0). We call this an *update operation*. Note that in RAND-GREEDY-ALG, updates only occur if the bits in positions c_1 and c_2 are both 1, as in Paar's algorithm, so no cancellation occurs. In DCLO, this is not the case.

The following invariant holds for the matrix input and will hold while running RAND-GREEDY-ALG and DCLO:

> **Row-sum invariant:** For any row r, the linear combination to be computed is the sum of all $v(i)$ for which position (r, i) in the matrix is 1.

As in Paar's algorithm, when a gate is added, RAND-GREEDY-ALG updates each row by changing the entries corresponding to the two inputs from 1 to 0 and placing a 1 in the new column corresponding to that gate. There is, however, a *feasibility requirement*: An update is only applied to a row r if, after doing the update to row r, it would still be possible to produce the output for that row in its goal depth. See Subsection 3.5 for a description of how feasibility is checked.

In order to choose two columns, c_1 and c_2, as inputs for the next gate, RAND-GREEDY-ALG determines how many of the output rows could *benefit* from having that gate, i.e., how many rows have 1's in columns c_1 and c_2, both of which can be flipped while maintaining feasibility. For each possible next gate, the number of output rows which would benefit from

the gate is calculated, giving an *improvement*. The inputs to the new gate are chosen at random from those that give the largest improvement. The algorithm is shown in Figure 2. The function FEASIBLE-UPDATE is shown in Figure 4.

3.5 Computing the feasibility of a gate

The algorithm, RAND-GREEDY-ALG, adds one gate at a time to the circuit. For each candidate for the next gate, RAND-GREEDY-ALG checks how many output rows can benefit from the candidate. We require that a row have 1s in the columns for the two inputs to the candidate gate in order to benefit from that gate. In addition, it must still be possible to calculate the row within its goal depth after using this gate. If both of these conditions hold, then we say the candidate is *feasible* for that output. For example, suppose two required outputs for a linear component are $x_1 \oplus x_2 \oplus x_3 \oplus x_4$ with goal depth 2 and $x_1 \oplus x_2 \oplus x_3 \oplus x_5 \oplus x_7$ with goal depth 3. A candidate gate computing $((x_1 \oplus x_2) \oplus x_3)$ at depth two would not be feasible for the first output, but would be feasible for the second.

In earlier work [5], computing a depth-16 circuit for the AES S-Box, feasibility of the gates was maintained by working in phases, never using gates produced in the current phase within that same phase. Thus, there was an upper bound on the depth of the gates produced in the same phase. Here, instead of using phases, we check feasibility explicitly, giving more freedom as to which candidate gates can be chosen.

First, we define a function FEASIBLE, which is used to check if a row is currently feasible. The depth of a column c is $d(c)$; it is the input depth for the inputs and is $1 + \max\{d(c_1), d(c_2)\}$ for a column created using columns c_1 and c_2. For any row r, the depths of columns with a 1 in that row form a multiset $Depths(r)$. FEASIBLE is a function of $Depths(r)$ and $goal\ depth(r)$. If there is only one value d in $Depths(r)$, then FEASIBLE($Depths(r)$, $goal\ depth(r)$) is true if and only if d is less than or equal to the goal depth for row r, $goal\ depth(r)$.

If there is more than one value in $Depths(r)$, then

$$\text{FEASIBLE (Depths } (r), goal\_depth(r)) = \text{ FEASIBLE } \left(\left(Depths(r) - \left\{ d_1, d_2 \right\} \right) \oplus \left\{ d_2 + 1 \right\}, goal\_depth(r) \right)$$

where d_1 is the smallest (d_2 the second smallest) value in $Depths(r)$ (note that they can be equal).

This can be accomplished by storing $Depths(r)$ in a priority queue containing keys which are the depths corresponding to all the 1s in that row. Then, FEASIBLE() is calculated as follows: while there is more than one depth value in the priority queue, find and delete the two lowest values, say d_1 and d_2, and insert the value $1 + \max(d_1, d_2)$. At the end, if the only depth value remaining is no larger than the goal depth for that output, the row is feasible. Otherwise, it is not. Pseudocode for this calculation can be found in Figure 3.

Assuming that H is a priority queue (min-heap order) containing the current depths of the variables to be XORed for row r, the algorithm, FEASIBLE, correctly determines whether or not there exists a circuit which computes the function for row r in depth gd.

Lemma 1.—The algorithm, FEASIBLE, for determining feasibility returns 1 if and only if there exists a circuit computing the XOR of the set of variables available at the depths in the priority queue, H, within goal depth gd.

Proof.: Each iteration of the while loop defines a new gate produced from two variables at depths d_1 and d_2. The depth of the new gate is $\max\{d_1,d_2\} + 1$, so inductively, it correctly defines a circuit with output at depth d. FEASIBLE only returns 1 if that depth d is at most gd and H has one element. Thus, if FEASIBLE returns 1, there exists a feasible circuit.

FEASIBLE determines if repeatedly taking the two columns with lowest weight for the next gate results in a circuit having no more depth than the goal depth for that output. The algorithm is thus correct if for any linear combination $F = x_{i1} \oplus x_{i2} \oplus \cdots \oplus x_{ik}$, where the x_{ij} could be original inputs to the component or outputs of XOR gates produced earlier, and any given depths for these k inputs, no circuit implementing this linear combination with the given depth constraints has lower depth than any circuit, C, produced in this way.

We consider any minimum depth circuit C' for F. Note that we can assume without loss of generality that all gates in C' are cancellation-free, since the inputs where cancellation occurs can be computed (for example, by essentially copying the computation of the original inputs, but removing the variables the inputs have in common from both subcircuits) in a cancellation-free manner without increasing the depth. Since a cancellation-free XOR circuit for k inputs has $k - 1$ gates, C' has $k - 1$ gates. C also has $k - 1$ gates, since each iteration of the while loop decreases the number of remaining elements in the priority queue by 1. Assume without loss of generality that the lowest depth of any input is 0.

We show that for any depth d, the number of gates at depth at most d in C is at least as large as the number of gates at depth at most d in C'. This clearly holds for all gates at depth 1. Note that in C at most one input at depth 0 is not an input to a gate at depth 1. More generally, at depth $d' + 1$, all but at most one of the inputs and gates at depth at most d', which have not yet been inputs to any gate, become inputs to gates at depth $d' + 1$, pairwise. This is the maximum possible number of gates at level $d' + 1$ given the gates which have already been produced. To see that no other circuit could do better by choosing different gates at this level or having chosen fewer gates at earlier levels, consider n_d the number of gates at level d in circuit C' and $S_d = \sum_{i=1}^{d} n_i$. Since no input to F and no gate in C' can be input to more than one gate in C', the number of gates is bounded by $n_d \leq \frac{1}{2}$ ((number of inputs at level less than d) − (number of gates at level less than d)).

$$S_d = n_d + S_{d-1} \leq \left\lfloor \frac{1}{2}\left(\left(\text{number of inputs at level less than } d\right) - S_{d-1}\right)\right\rfloor + S_{d-1}$$

which is an increasing function of S_{d-1}. Inductively, C has at least as many gates at level at most d as C'.

If feasibility holds for all rows, there exists a circuit which evaluates all required outputs within the given depth constraints. By Lemma 1, we can say that feasibility holds for a row,

r, if and only if FEASIBLE returns 1 for *r*. In a valid DCLO initial matrix, all goal depths are initially feasible. Feasibility is the second invariant in our algorithm:

Feasibility invariant: For any row r, the goal depth remains feasible.

We ensure this invariant by explicitly testing for feasibility before each matrix update.

To calculate if an update (candidate gate) is feasible for an output corresponding to row r and columns c_1, c_2 at depths d_1, d_2, we need to calculate

$$\text{FEASIBLE}\left(\left(Depths(r) - \{d_1, d_2\}\right) \oplus \left\{\max\{d_1, d_2\} + 1\right\},\ \text{goal depth}\ (r)\right)$$

For RAND-GREEDY-ALG, we also require that the entries in columns c_1 and c_2 of row r are both 1s.

We can now show that RAND-GREEDY-ALG runs in polynomial time.

Theorem 1.—RAND-GREEDY-ALG is correct. Let M be an m × n 0–1 matrix containing H 1s. Suppose that, every row in M is feasible, according to the function FEASIBLE initially. The running time of RAND-GREEDY-ALG is $O(tm(t^2+n \log n))$, where t is the final number of columns and is at most $H+n-m \leq mn+n-m$.

Proof.: By Lemma 1, no input or gate is considered as possible inputs to a new gate for a particular row unless that row can still be computed within its goal depth with that new gate. The algorithm continues as long as any row has Hamming weight greater than 1. As long as it continues, by the Feasiblity Invariant, there is a candidate gate which would be feasible for some row. Since the updates of the matrix are Paar-like operations and the depths are calculated correctly, the algorithm is correct.

Within FEASIBLE, at most two Insert and DeleteMin priority queue operations are performed for each 1 in row *r*, since the number of elements in the priority queue is decreased by one each time through the while loop. The number of 1s in any row is at most the initial number of columns in the matrix, *n*. Thus, the running time of FEASIBLE is $O(n \log n)$. The for each loop in RAND-GREEDY-ALG takes the most time within the outer while loop. Each pair of the first $s-1$ columns is considered, $O(t^2)$ pairs. For each pair, only constant work is done in a row by FEASIBLE-UPDATE unless there are 1s in that row for both columns. If there are 1s for both columns, FEASIBLE is called. Thus, the running time for each row, in one iteration of the while loop, is $O(t^2 + n \log n)$.

Since the while loop is executed once for each new gate, it is executed at most *t* times. There are *m* rows to process, so RAND-GREEDY-ALG runs in time $O(tm(t^2 + n \log n))$. Since there are at most *n* 1s in every row initially, each row will be computed using at most $n-1$ XORs, and all *m* rows will be computed with at most $m(n-1)$ XORs. There are *n* columns initially, so in all $t \leq H+n-m \leq mn+n-m$.

3.6 Improvements to RAND-GREEDY-ALG: DCLO

In the following, we introduce three new techniques that were added to the simplified algorithm, RAND-GREEDY-ALG, to give the algorithm we used to produce the improved circuits for several functions useful for cryptography that are mentioned in this paper, with straightline programs for the circuits available at [16]. We refer to the algorithm with these three improvements as DCLO.

Reducing the size of the matrix — preprocessing—Using the See-Saw Method, particularly after the first iteration, the upper linear component may have some outputs which can be computed at a larger depth than some others, without affecting the total depth of the circuit. In the case of the AES S-Box, and presumably in some other applications, there are cases where an output g can be computed as the sum of exactly two of the other outputs, and both of those other outputs have to be computed at a lower depth than is allowed for g. Thus, there is no reason for the row representing output g to be included in the input matrix to RAND-GREEDY-ALG. Adding one extra gate at the end of the computation, adding those two other outputs, will suffice. This extra gate at the end will typically, but not necessarily, have some cancellation. In our experiments with the upper linear component of the AES S-Box, there were actually 5 such outputs which could be automatically computed in this manner, yielding the the 27-gate circuit for the upper linear component. The preprocessing is never relevant for the lower linear component, since all outputs are given the same goal depths.

Finding these triples of outputs is relatively straight-forward, checking all triples of rows in the matrix, checking that one of the three rows has a larger goal depth than the other two, and checking that the bitwise XOR of the three rows is 0. This preprocessing can be done each time, before RAND-GREEDY-ALG is run. The gates found are saved and then added to the end of the straight-line program for the circuit.

Allowing more cancellation – the Generalized Paar Operation—The simplified algorithm, RAND-GREEDY-ALG, without the preprocessing of the matrix, produces cancellation-free circuits since it only does Paar-like operations. Recall that a Paar-like operation updates a row when a new gate is created based on the inputs to that gate. If the inputs come from columns i and j, the row in question must have 1s in both columns i and j. The 1s must be changed to 0s, and a 1 must be placed in the new column for that gate. No other changes are made to the row.

In the following example, every cancellation-free circuit is suboptimal. Consider running Paar's algorithm on the following matrix:

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

Assume the first two columns are the first air of columns chosen. The matrix becomes:

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

If the third and fifth columns are chosen next, the matrix becomes:

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

Now, choosing the fourth and sixth columns yields

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

At this point Paar's algorithm would require two more additions. Instead, we allow a simple no-cost rewrite of the matrix:

Note that a flip uses (or omits using) the gate defined by column c in row r and preserves the row-sum invariant. We do a flip if the Hamming weight of the row decreases as a result.[3] The flip operation introduces the possibility of cancellation.

In our example, note that $v(7) = (1, 1, 1, 1) = x_1 + x_2 + x_3 + x_4$ A flip on row 4, column 7 yields

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

and only one more addition completes the circuit. The cost is only four gates, which is not possible without cancellation.

Allowing second best pairs—In order to choose a pair of columns for the update, RAND-GREEDY-ALG counts for each possible pair the number of rows where it is feasible, computes the maximum of these counts, and chooses randomly among those pairs with the maximum count. Clearly, this greedy approach of choosing among those pairs with the maximum count is intuitively reasonable. However, it may be that this greedy approach sometimes gives a suboptimal solution overall. Thus, with probability 2 %, in DCLO a random choice is made among the column pairs with the second to largest count, rather than the maximum count.

[3] We relax this requirement in our code.

3.7 The See-Saw Method applied to the AES S-Box, an example

The algorithm DCLO, which we applied to obtain our new, small, low-depth circuits, was RAND-GREEDY-ALG, plus the improvements of the last three subsections, the preprocessing, the Generalized Paar Operations, and allowing the second best pairs. We applied the See-Saw Method to the AES S-Box (in the forward direction) and obtained a circuit of size 125 and depth 16. This required very few iterations of DCLO.

With each iteration, we ran RAND-GREEDY-ALG 10 000 times and chose one of the circuits produced with the smallest number of gates.

In the forward direction, there are four outputs of the AES S-Box which are negated. These negations were ignored until the end. We started with the middle nonlinear component found in [5]. This had 63 gates and was used in all of our circuits. For the upper linear and lower linear components, we used the original Paar algorithm [15], with no regard for depth, always choosing the first of the pairs of columns where 1s occurred in the most rows. The circuit we created had 27 gates for the upper linear component, 34 for the lower linear, for a total of 124 gates and depth 19.

Then we created a new upper linear component with our algorithm, setting the goal depths for all outputs to the minimum possible; for a row with Hamming weight h, this minimum is $\lceil \log_2(h) \rceil$. Since all rows have weight less than 8, the values ranged from 1 to 3. This resulted in a component with 29 gates, increasing the total circuit size to 126, but decreasing the total depth to 18. Note that for a function, such as the AES S-Box, consisting only of an upper linear component, a middle nonlinear component and a lower linear component, initially creating an upper linear circuit with minimum goal depths, will make it possible to obtain a minimum depth circuit (assuming the nonlinear component is fixed) in the next iteration. This holds since the See-Saw Method next applies DCLO to the lower linear component with input depths determined by the current circuit and goal depths all equal to the optimal depth (given the middle nonlinear component being used).

Then we ran the algorithm on the lower linear component, with goal depths of 16 (15 is impossible given the middle nonlinear component used; three of the outputs of the lower linear component could not be computed in depth 15, though the others could). The smallest circuit found had size 35, giving us a circuit with 127 gates and depth 16. One generalized Paar operation was used.

This lower linear component was then used to try to get a smaller upper linear component, but still depth 16. The goal depths were relaxed as much as possible and the preprocessing was used. The smallest number of gates found was 27, giving us our final circuit of depth 16 with only 125 gates in all. One of the outputs was at depth 15 and the others were all at depth 16. (In our second try for a lower linear component of size 35, we found one giving depth 16 for all outputs and chose this instead. Our final circuit has depth 16 for all outputs. Five preprocessing gates were used.)

Working on the inverse AES S-Box, starting with the middle nonlinear component found in [5] and minimum goal depths for the top linear component, we got a component with 29

gates. Working on the lower linear component with goal depths of 16, we got 35 gates, but 10 000 times was not enough and we changed it to 100 000 times. Introducing the slack goal depths for the upper linear component gave us 28 gates using 5 preprocessing gates. Thus, the total size is 126 gates. Note that this uses some XNOR gates in the upper linear component, while the forward direction only uses them on outputs that need to be negated. For the inverse, the inputs corresponding the negated outputs from the forward direction need to be negated. This effect can be achieved by computing the desired circuit without the negations and XNORs and then changing some XORs to XNORs when exactly one of the inputs should have been negated.

The depth cannot be reduced without changing the circuit for the middle nonlinear component. Of course, if the logical base is expanded, one could probably decrease the sizes slightly. For example, if NAND gates are used in the circuit for inversion in $GF(2^4)$, it is not hard to reduce the number of gates by two without increasing the depth (see Appendix A). Since there are only 256 possible inputs, we verified the circuits fully against the specifications in [13].

4 Rows with Hamming weight 2

Previous work, also obtaining a small depth-16 circuit for the AES S-Box [5], was less automated, obtained a slightly worse result, and had a minor error in the algorithm. In this section, we explain that error. The algorithm works in phases: During phase $i \geq 0$, no row in the current matrix has Hamming weight more than 2^{k-i} and only inputs or gates already produced at depth i or less are considered as possible inputs to gates in phase i. Thus, the depth of gates in phase i is at most $i + 1$. At the beginning of each phase of that algorithm, there is a check to see if there are any rows with Hamming weight 2. If so, the algorithm created the final gate for any such rows at that point. At first glance, it seems as if this can only help the algorithm, since that gate would need to be produced at some point anyway. However, there are a couple of problems with this strategy.

If handling a row with Hamming weight 2 only takes place at the beginning of a phase, there is no conflict with the definition of a phase, since the columns chosen must have been created in the previous phase, so their depth would be acceptable for the new phase. However, if such handling had been allowed in the middle of a phase, one of the columns chosen could be from the current phase and the depth of the new gate could be one too large to be used for another row in the current phase. For example, suppose we have the following matrix:

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

This can clearly be implemented in depth 2. However, if we choose the last two columns first, the matrix becomes:

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

If we now choose the last row, because it has Hamming weight 2, and if we use that gate for the other rows, the matrix becomes:

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

Now we need depth 3 to finish.

There are also examples where choosing a row with Hamming weight 2 does not increase the depth, but leads to a larger number of gates:

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

If one chooses the first row, because it is the first with Hamming weight 2, the matrix becomes:

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

Next one chooses the fourth row, because it is the first with Hamming weight 2, where both 1s have depth 0, giving:

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

One needs to choose the first two columns before choosing any columns at depth 1, so six more gates are necessary.

If instead, we started with the other row with Hamming weight 2, which happens to be the one which will decrease our total Hamming weight most, our second matrix would become:

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

Then, choosing the first and fourth columns gives:

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

Now only five more gates are necessary to finish. Thus, one cannot arbitrarily choose a row with Hamming weight 2 and assume that this cannot have a negative affect on the number of gates used.

5 Circuits

This work concentrated on optimizing the linear components of circuits. In [5], the search technique in [2], to find circuits for nonlinear components with few AND gates, was modified to reject candidate gates with too large depth. This decreased the depth of the $GF(2^4)$ inversion from 9 to 4 while only increasing the number of gates from 16 to 17, changing 5 AND gates and 11 XOR gates to 7 AND gates and 10 XOR gates.

The techniques presented in this paper and in [5] appear, not surprisingly, to lead to a trade-off between size and depth. To illustrate this trade-off, we list first the depths and sizes of some of the circuits for the AES S-Box which we obtained earlier:

- depth 28; size 115: [2]
- depth 27; size 113: published on [16].
- depth 23; size 116: inserting the new middle component into [2]
- depth 22; size 117
- depth 21; size 118
- depth 20; size 122
- depth 16; size 128: final result in [5]

In this research we obtained the following:

−depth 19; size 124: new middle component with Paar result for lower and upper linear components

−depth 18; size 126: calculating lower linear component with minimum depths for all outputs

−depth 16; size 125: final result

We applied RAND-GREEDY-ALG to the multiplication of binary polynomials of degree 9, starting from the straight-line program given by Bernstein on http://binary.cr.yp.to/m.html. Running RAND-GREEDY-ALG on the lower linear component of that circuit, we obtained the same size as Bernstein with 155 gates, but reduced the depth from 9 to 6. Cenk and Hasan [6] report the same number of gates, but depth 8. Find and Peralta [8] report size 154 and depth 9, but have a different nonlinear component. Running RAND-GREEDY-ALG on the lower linear component of their circuit also gave 154 gates, but depth 7. Depth 6 was not feasible, given their nonlinear component. No Generalized Paar Operations were used, and of course no preprocessing of the matrix, since all goal depths were 7.

We applied RAND-GREEDY-ALG to computing the product of degree 12 polynomials over $GF(2)$. Bernstein has a circuit with depth 9 and 256 gates. Our techniques, starting with the straight-line program given on his homepage http://binary.cr.yp.to/m.html, gave depth 8 and 255 gates, which is the same result obtained by Cenk and Hasan [6]. We have no reason to believe that RAND-GREEDY-ALG would not produce similar results for computing the products of binary polynomials of other degrees, but did not pursue this since it did not seem to reduce the number of gates.

The tower field construction (see, for example, [11]) for a Galois Field with 2^{2^k} elements lends itself well to the methods described here. We built circuits for multiplication and inversion in $GF(2^k)$ for $k = 2, 4, 8, 16$. We constructed $GF(2^{2k})$ from an optimized circuit for $GF(2^k)$. Then we optimized the new circuit. The results for $GF(2^8)$ and $GF(2^{16})$ are of wide applicability in cryptography. The sizes and depths obtained are as follows:

−$GF(2^8)$ multiplication: size 106 and depth 6.

−$GF(2^8)$ inversion: size 98 and depth 18.

−$GF(2^{16})$ multiplication: size 381 and depth 8.

−$GF(2^{16})$ inversion: size 387 and depth 31.

These results are much better than anything previously known. However, the depths of the inversion circuits seem much bigger than what is likely possible. This may be due to the specific inversion formulas that are natural in tower fields (see Appendix B). We may attempt to improve on these depths in future research.

Inversion in $GF(2^{16})$ is the basis of the 16-bit S-Box proposed by Kelly et al. in [9]. The paper quotes a size of 1382 gates (1238 XOR gates and 144 AND gates, no depth is given). This size is derived from the work in [18, 19]. If instead we use the tower field representation of Appendix B, the resulting S-Box has 446 gates (113 AND gates, 324 XOR gates, and 9 XNOR gates) and depth 35. This is not exactly the same S-Box as the one by

Kelly et al. A quick way to derive the latter S-Box from the tower field S-Box is by doing a change of basis. Without further optimization, the resulting circuit has 537 gates.

The circuit for inversion in $GF(2^{16})$ was verified using an automatically generated circuit for multiplication and then using this circuit to verify that x multiplied by x^{-1} is 0 for $x = 0$ and the identity element of the field for all other values of x. We saw it necessary to add this verification method after an anonymous referee determined that a previous circuit was incorrect. We are grateful for his/her contribution.

The circuits described here have been posted at [16].

6 Conclusion

Automated techniques for finding small, low-depth circuits for cryptographic functions were presented. The See-Saw Method and the algorithm, DCLO, used within it were successful in finding better results for the AES S-Box and other functions.

A Inversion in $GF(2^4)$

Figure 6 demonstrates that if NAND gates are allowed in addition to AND and XOR gates, then there is a circuit with depth 4 and only 15 gates computing inversion in $GF(2^4)$. This is the same depth, but two fewer gates than we used for this work. It also has one less gate than was used in [2], where depth 9 was acceptable.

B Tower field construction up to $GF(2^{16})$

In the following, bases will be defined for each of the finite fields. Each base (b_1, b_2) will be such that $b_1 + b_2 = 1$. This identity can be verified by repeated squaring of the defining irreducible polynomial and adding a telescoping sequence (verify $GF(2^k)$ before $GF(2^{2k})$). For each k, the irreducible polynomial for $GF(2^{2k})$ was found using the circuits for multiplication and addition in $GF(2^k)$ to compute the range of $x^2 + x$. Then $x^2 + x + a$ is irreducible for any a not in the range of $x^2 + x$.

$GF(2^2)$ is built from $GF2$ by adjoining a root W of $x^2 + x + 1$.

A basis for $GF(2^2)$ is (W, W^2)

$GF(2^4)$ is built from $GF(2^2)$ by adjoining a root Z of $x^2 + x + W^2$.

A basis for $GF(2^4)$ is (Z^2, Z^8).

$GF(2^8)$ is built from $GF(2^4)$ by adjoining a root V of $x^2 + x + WZ^2$.

A basis for $GF(2^8)$ is (V, V^{16}).

$GF(2^{16})$ is built from $GF(2^8)$ by adjoining a root T of $x^2 + x + WZ^2V$.

A basis for $GF(2^{16})$ is (T, T^{256}).

B.1 Multiplication and inversion in *GF* (2^{16})—Let $\Theta = WZ^2V$. Multiplication is given by

$$\left(aT + bT^{256}\right)\left(cT + dT^{256}\right) = (ac + \Theta(a+b)(c+d))T + (bd + \Theta(a+b)(c+d))T^{256}$$

We now derive efficient equations for inversion in GF(2^{16}). The identity element is $1 \cdot T + 1 \cdot T^{256}$.

From the multiplication formulas we get

$$1 = ac + \Theta(a+b)(c+d) \quad 1 = bd + \Theta(a+b)(c+d)$$

Setting $\mu = \Theta(a + b)$ and summing yields

$$1 = c(a+\mu) + d\mu \quad 0 = ac + bd$$

Equate the c coefficients

$$a = ca(a+\mu) + da\mu \quad 0 = ac(a+\mu) + bd(a+\mu)$$

Summing them

$$a = d(b(a+\mu) + a\mu) \quad d = (b(a+\mu) + a\mu)^{-1}a$$

Yields

$$c = bda^{-1} = b(b(a+\mu) + a\mu)^{-1} \quad d = a(b(a+\mu) + a\mu)^{-1}$$

and

$$c = b\left(ba + (a+b)^2\Theta\right)^{-1} \quad d = a\left(ba + (a+b)^2\Theta\right)^{-1}$$

The operation $(a+b)^{2\Theta}$ is usually referred to as “square-scaling”. Both square-scaling and inversion in the equations for c, d are operations in the lower field GF(2^8).

Acknowledgments

The first author was supported in part by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014–00041. The second author participated in this research while a guest researcher at the National Institute of Standards and Technology during 2015–2016.

References

1. Bernstein DJ. Optimizing linear maps modulo 2. Available at http://cr.yp. to/papers.html#linearmod2.

2. Boyar J, Matthews P, and Peralta R. Logic minimization techniques with applications to cryptology. J. of Cryptology, 26(2):280–312, 2013.
3. Boyar J, Peralta R, and Pochuev D. On the multiplicative complexity of Boolean functions over the basis (∧, ⊕, 1). Theoretical Computer Science, 235:43–57, 2000.
4. Boyar Joan and Magnus Gausdal Find. Cancellation-free circuits in unbounded and bounded depth. Theor. Comput. Sci, 590:17–26, 2015.
5. Boyar Joan and Peralta Ren´e. A small depth-16 circuit for the AES s-box. In Gritzalis Dimitris, Furnell Steven, and Theoharidou Marianthi, editors, Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, volume 376 of IFIP Advances in Information and Communication Technology, pages 287–298. Springer, 2012.
6. Cenk Murat and Anwar Hasan M. Some new results on binary polynomial multiplication. J. Cryptographic Engineering, 5(4):289–303, 2015.
7. Courtois N, Hulme D, and Mourouzis T. Solving circuit optimisation problems in cryptography and cryptanalysis. IACR Cryptology ePrint Archive, 2011:475, 2011 Appears in electronic proceedings of 2nd IMA Conference Mathematics in Defense, UK, Swindon, 2011, www.ima.org.uk/ db/ documents/Courtois.pdf.
8. Magnus Find and Rene Peralta. personal communication, 2017.
9. Kelly Matthew, Kaminsky Alan, Kurdziel Michael T., Lukowiak Marcin, and Radziszowski Stanislaw P.. Customizable sponge-based authenticated encryption using 16-bit s-boxes. In 34th IEEE Military Communications Conference, MILCOM 2015, Tampa, FL, USA, October 26–28, 2015, pages 43–48, 2015.
10. Lupanov OB. A method of circuit synthesis. Izvestia V.U.Z. Radiofizika, 1:120– 140, 1958.
11. Moradi Amir, Poschmann Axel, Ling San, Paar Christof, and Wang Huaxiong. Pushing the Limits: A Very Compact and a Threshold Implementation of AES, pages 69–88. Springer Berlin Heidelberg, 2011.
12. Nechiporuk EI. On the complexity of schemes in some bases containing nontrivial elements with zero weights (in Russian). Problemy Kibernetiki, 8:123–160, 1962.
13. NIST. Advanced Encryption Standard (AES) (FIPS PUB 197). National Institute of Standards and Technology, 11 2001.
14. Nogami Y, Nekado K, Toyota T, Hongo N, and Morikawa Y. Mixed bases for efficient inversion in $F(((2^2)^2)^2)$ and conversion matrices of SubBytes of AES In Mangard S and Standaert F-X, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 234–247. Springer, 2010.
15. Paar . Optimized arithmetic for Reed-Solomon encoders. In 1997 IEEE International Symposium on Information Theory, page 250, 1997.
16. Peralta R. Circuit minimization work http: // cs-www. cs. yale. edu/ homes/ peralta/ CircuitStuff/ CMT. html , (accessed March 10, 2018).
17. Shannon CE. The synthesis of two-terminal switching circuits. Bell System Tech. J, 28:59–98, 1949.
18. Wood Christopher A.. Large substitution boxes with efficient combinational implementations. Rochester Institute of Technology, 2013.
19. Wood Christopher A., Radziszowski Stanislaw P., and Lukowiak Marcin. Constructing large s-boxes with area minimized implementations. In Military Communications Conference, MILCOM 2015–2015 IEEE, pages 49–54. IEEE, 2015.

$$\left(\begin{array}{c|c|c|c|c} & x_1:0 & x_2:2 & x_3:1 & x_4:0 \\ \hline y_1:2 & 1 & 0 & 1 & 1 \\ \hline y_2:3 & 0 & 1 & 1 & 1 \\ \hline y_3:4 & 1 & 1 & 1 & 1 \\ \hline y_4:3 & 1 & 1 & 0 & 1 \end{array}\right)$$

Fig. 1.
Sample DCLO problem

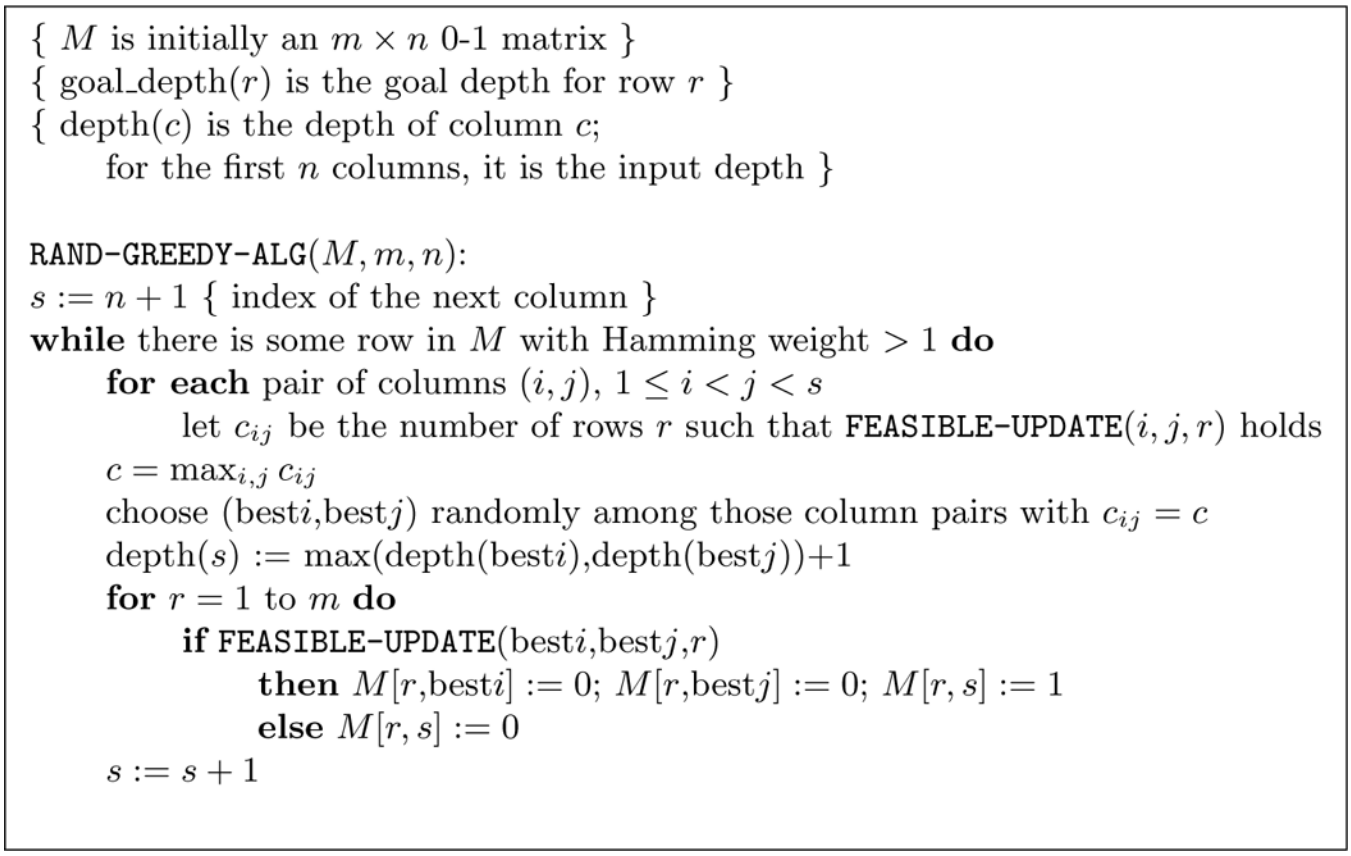

Fig. 2.
Algorithm for creating a low-depth circuit for linear components

```
{ H is a min-heap ordered priority queue containing the depths of inputs
(Depths(r) for row r). H is non-empty. }
{ gd is the goal depth (goal_depth(r) for row r). }

FEASIBLE (H, gd):
{ Checks if the XOR of the set of input variables available
at the depths in H can be computed within goal depth gd.
    Returns 1 if it can and 0 otherwise. }
    while H has more then one entry do
    { Choose the wires with the lowest depth for the next gate }
        d_1 :=DeleteMin(H)
        d_2 :=DeleteMin(H)
        Insert(H,1 + max(d_1, d_2))
    { The depth of the output gate is now the only depth in H }
    d :=DeleteMin(H)
    { Determine if the new depth is feasible }
    if (d ≤ gd) then
        return(1)
    return(0)
```

Fig. 3.
Algorithm for determining feasibility of a row

```
{ M is an m × (s − 1) 0-1 matrix }
{ goal_depth(r) is the goal depth for row r }
{ depth(c) is the depth of column c;
      for the first n columns, it is the input depth }

FEASIBLE-UPDATE (c1, c2, r):
{ Checks if an XOR gate with inputs from columns c1, c2 can be used for row r.
      Returns 1 if it can and 0 otherwise. }
if M[r, c1] = M[r, c2] = 1 then
{ Only return 1 if both c1 and c2 are necessary for row r }
      H := empty priority queue
      for 1 ≤ c < s, c ∉ {c1, c2} do
      { Put all depths of wires which need to be XORed into priority queue H }
            if M(r, c) = 1 then
                  Insert(H,depth(c))
      { Put depth of potential new gate in H }
      Insert(H,1 + max(depth(c1),depth(c2))
      return(FEASIBLE(H, goal_depth(r))
return(0)
```

Fig. 4.
Algorithm for determining feasibility of a candidate gate

Assume the first two columns are the first pair of columns chosen. The matrix becomes:

$$\begin{pmatrix} 0\,0\,0\,0\,1 \\ 0\,0\,1\,0\,1 \\ 0\,0\,1\,1\,1 \\ 0\,1\,1\,1\,0 \end{pmatrix}$$

If the third and fifth columns are chosen next, the matrix becomes:

$$\begin{pmatrix} 0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,1 \\ 0\,0\,0\,1\,0\,1 \\ 0\,1\,1\,1\,0\,0 \end{pmatrix}$$

Now, choosing the fourth and sixth columns yields

$$\begin{pmatrix} 0\,0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,0\,1 \\ 0\,1\,1\,1\,0\,0\,0 \end{pmatrix}$$

At this point Paar's algorithm would require two more additions. Instead, we allow a simple no-cost rewrite of the matrix:

> Complement a bit in column c and row r along with all bits in row r that are 1s in the vector $v(c)$.

Fig. 5.
The *flip* operation

$$
\begin{array}{lll}
t_1 = x_2 + x_3 & t_2 = x_2 \times x_0 & t_3 = x_1 + t_2 \\
t_4 = x_0 + x_1 & t_5 = x_3 + t_2 & t_6 = t_5 \times t_4 \\
t_7 = t_3 \times t_1 & t_8 = x_0 \; NAND \; x_3 & t_9 = t_4 \times t_8 \\
t_{10} = x_2 \; NAND \; x_1 & t_{11} = t_1 \times t_{10} & y_0 = t_2 + t_{11} \\
y_1 = x_3 + t_7 & y_2 = t_2 + t_9 & y_3 = x_1 + t_6
\end{array}
$$

Fig. 6.

Inversion in $GF(2^4)$ using NAND gates. Input is (x_0,x_1,x_2,x_3) and output is (y_0,y_1,y_2,y_3).