Logic Minimization Techniques with Applications to Cryptology[*]

Joan Boyar [†] Philip Matthews [‡] René Peralta [§]

January, 2011

Abstract

A new technique for combinational logic optimization is described. The technique is a two-step process. In the first step, the non-linearity of a circuit – as measured by the number of non-linear gates it contains – is reduced. The second step reduces the number of gates in the linear components of the already reduced circuit. The technique can be applied to arbitrary combinational logic problems, and often yields improvements even after optimization by standard methods has been performed. In this paper we show the results of our technique when applied to the S-box of the Advanced Encryption Standard (AES [13]).

We also show that in the second step, one is faced with an NP-hard problem, the Shortest Linear Program (SLP) problem, which is to minimize the number of linear operations necessary to compute a set of linear forms. In addition to showing that SLP is NP-hard, we show that a special case of the corresponding decision problem is Max SNP-Complete, implying limits to its approximability.

Previous algorithms for minimizing the number of gates in linear components produced cancellation-free straight-line programs, i.e., programs in which there is no cancellation of variables in GF(2). We show that such algorithms have approximation ratios of at least 3/2 and therefore cannot be expected to yield optimal solutions to non-trivial inputs. The straight-line programs produced by our techniques are not always cancellation-free. We have experimentally verified that, for randomly chosen linear transformations, they are significantly smaller than the circuits produced by previous algorithms.

Keywords: Circuit complexity; multiplicative complexity; linear component minimization; Shortest Linear Program; cancellation; AES; S-box.

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

[*]Some of this work appeared in [4] and some appeared in [6].

[†]Department of Mathematics and Computer Science, University of Southern Denmark, joan@imada.sdu.dk. Partially supported by the Danish Natural Science Research Council (FNU). Some of this work was done while visiting the University of California, Irvine.

[‡]This work was done while at the University of Aarhus.

[§]Information Technology Laboratory, NIST, rene.peralta@nist.gov.

and one Boolean output (there are 2^{256} such functions). As an example of this, consider multiplicative complexity, the number of GF(2) multiplications necessary and sufficient to compute a function. The multiplicative complexity of the Boolean function E^8_4, which is true if and only if exactly four of its eight input bits are true, is unknown [5].

In practice, we build circuit implementations of functions using a variety of heuristics. Many of these heuristics have exponential time complexity and thus can only be applied to small components of a circuit being built. This works reasonably well for functions that naturally decompose into repeated use of small components. Such functions include arithmetic functions (which we often build using full adders), matrix multiplication (which decomposes into multiplication of small submatrices), and more complex functions such as cryptographic functions (which are commonly based on multiple iterations of an algorithm containing linear steps and one or more non-linear steps).

This work presents a new technique for logic synthesis and circuit optimization. The technique can be applied to arbitrary functions, and yields improvements even on programs/circuits that have already been optimized by standard methods. We apply our technique to the S-box of AES[1], which, in addition to being used in AES, has been used in several proposals for a new hash function standard[2]. The result is, as far as we know, the smallest circuit yet constructed for this function. The circuit contains 32 AND gates and 83 XOR/XNOR gates for a total of 115 gates. We have also applied these techniques to the logic embedded in the non-linear components of several candidates to the SHA-3 competition. The improvements in software performance were significant.

Our circuits are over the basis $\{\oplus, \wedge, 1\}$. This basis is logically complete: any Boolean circuit can be transformed into this form using only local replacements. The circuit operations can be viewed either as performing Boolean logic or arithmetic modulo 2 (when viewing it the latter way, we will write outputs to be computed as polynomials with multiplication replacing $\wedge$ and addition replacing $\oplus$). The number of $\wedge$ gates is called the *multiplicative complexity* of the circuit. Connected components of the circuit containing $\wedge$ gates are called *non-linear*. Components free of $\wedge$ gates are called *linear*. Circuits and programs for computing Boolean functions can be defined using straight-line programs, where each statement defines the operation of a gate or a line in a program. The examples in Fig. 1, define two different circuits, and their corresponding straight-line programs, for computing the majority function of three inputs, a, b, and c.

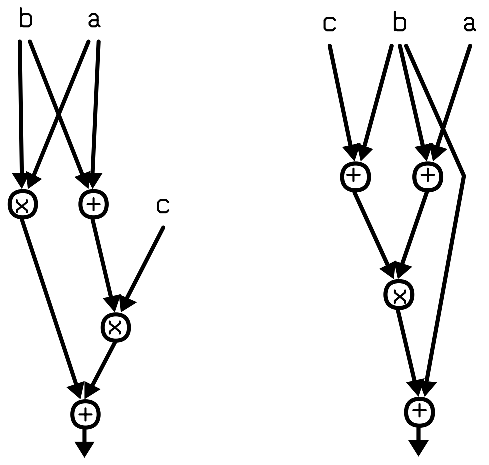
1.1 Combinational Circuit Optimization

The techniques described here would generally be applied to subcircuits of a larger circuit, such as an S-box in a cryptographic application, which have relatively few inputs and outputs connecting them to the remainder of the circuit. The key observation that led us to our techniques is that circuits with low multiplicative complexity will naturally have large sections which are purely linear (i.e. contain only $\oplus$ gates). Thus

> *it is plausible that a two-step process, which first reduces multiplicative complexity and then optimizes linear components, leads to small circuits.*

[1]Our circuit for the AES S-box has already been used as the basis of a software bitsliced implementation of AES in counter mode [18].

[2]See http://csrc.nist.gov/groups/ST/hash/sha-3/index.html



t_1	=	$a \wedge b$;	t_2	=	$a \oplus b$;	t_3	=	$t_2 \wedge c$;	t_4	=	$t_1 \oplus t_3$;
u_1	=	$a \oplus b$;	u_2	=	$b \oplus c$;	u_3	=	$u_1 \wedge u_2$;	u_4	=	$u_3 \oplus b$;

Figure 1: Two circuits and corresponding straight-line programs for MAJ(a, b, c).

We have, of course, no way of proving this hypothesis. But the experiments reported here support it. Additionally, we have successfully applied the heuristics described in this paper to a number of circuit optimization problems of interest to cryptology. These include finite-field arithmetic and binary multiplication. New records are periodically posted at http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html.

1.1.1 First step

The first step of our technique consists of identifying non-linear components of the subcircuit to be optimized and reducing the number of $\wedge$ gates. This reduction is not easy to do. For example, it is not obvious how to algorithmically transform one of the two equivalent circuits defined in Fig. 1 into the other.

Classic results by Shannon [26] and Lupanov [19] show that almost all predicates on n bits have Boolean circuit complexity about $\frac{2^n}{n}$. Analogous to the Shannon-Lupanov bound, it was shown in [8] 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 functions is significantly smaller than their Boolean complexity.

Finding circuits with minimum multiplicative complexity is, in all likelihood, a highly intractable problem. However, recent work on multiplicative complexity contains an arsenal of reduction techniques that in practice yield circuits with small, and often optimal, multiplicative complexity [5]. That work focuses exclusively on symmetric functions (those whose value depends only on the Hamming weight of the input).

In this paper we use ad-hoc heuristics to construct a circuit with low multiplicative complexity for inversion in $GF(2^4)$. (In general, $GF(2^n)$ is the field with 2^n elements.) The technique is described in Section 2.1.

1.1.2 Second step

The second step of our technique consists of finding maximal linear components of the circuit and then minimizing the number of XOR gates needed to compute the target functions computed in these linear components. A new heuristic for this computationally intractable problem is described in Section 3.1.

1.2 The Shortest Linear Program Problem

We argue below that minimizing the number of XOR gates in the second step is equivalent to the Shortest Linear Program problem over GF(2).

Let $\mathbb{F}$ be an arbitrary field and let

$$\begin{gathered}\alpha_{1,1}x_1 + \alpha_{1,2}x_2 + \ldots + \alpha_{1,n}x_n \\ \alpha_{2,1}x_1 + \alpha_{2,2}x_2 + \ldots + \alpha_{2,n}x_n \\ \ldots \\ \alpha_{m,1}x_1 + \alpha_{m,2}x_2 + \ldots + \alpha_{m,n}x_n\end{gathered}$$

be a set of linear forms where the $\alpha_{i,j}$'s are constants from $\mathbb{F}$ and the x_i's are variables over $\mathbb{F}$.

Suppose a subcircuit for a linear component in a circuit has x_is as inputs and y_js as outputs.[3] The y_j are linear functions of the x_is in the field $GF(2)$, so the subcircuit is an algorithm for computing the linear forms (the functions the y_js represent) given the x_i's as input, in the special case where $\mathbb{F} = GF(2)$.

We consider this question in the model of computation known as *linear straight-line programs.* A linear straight-line program is a variation on a straight-line program which does not allow multiplication of variables. That is, every line of the program is of the form $u := \lambda v + \mu w$; where λ, μ are in $\mathbb{F}$ and v, w are variables. Some of the lines are output lines; these are the lines where the linear forms in the set are produced. For brevity, we will use the terms *linear programs* or simply *programs* to refer to linear straight-line programs. The *length* of the program is the number of lines it contains, and is equal to the number of XOR gates in a subcircuit computing these forms. A program is *optimal* if it is of minimum length.

The linear straight-line program model (see [9] for a discussion of linear complexity) has the advantage of being very structured, but is nevertheless optimal to within a constant factor as compared to arbitrary straight-line programs when the computation is over an infinite field. Over finite fields the optimality of linear straight-line programs is unknown[4], but we restrict our attention to this form and consider minimizing the length of the program.

The standard algorithm for computing the linear forms $A\mathbf{x}$, where A is an $m \times n$ matrix containing entries from a set of size r, requires $m(n-1)$ operations. Savage [25], however,

[3]We consider circuits without negations only. There is no loss of generality in doing so because negations can be treated as standard XOR gates via $(\neg X = (X \oplus 1))$.

[4]It is not known if multiplication of variables can ever be used to reduce program length when the program outputs only linear functions.

showed that $O(mn/\log_r m)$ operations are sufficient in many cases, including computations over $GF(2)$ if $m \geq 4$. Williams [28] improved this to $O(n^2/\log^2 n)$ on a RAM with word length $\Theta(n)$ for n by n matrices over finite semirings. In contrast, Winograd [29] has shown that most sets of linear forms have a non-linear complexity in the straight-line program model; in fact, for a "random" $m \times n$ matrix A the probability is high that its complexity is $\Omega(mn)$ (for infinite fields). However, there are non-trivial matrices which can be computed considerably faster than this.

Over $GF(2)$, finding the shortest linear straight-line program is equivalent to our original goal of finding a circuit with only XOR gates and minimizing the number used. Linear forms have many applications, especially to problems in scientific computation, and there has been considerable success in finding efficient algorithms for computing them in special cases. The best known example is the Fast Fourier Transform, an $O(n \log n)$ algorithm, discovered by Cooley and Tukey in 1965 [12].

In section 3.1.1 we show that finding the shortest linear straight-line program is NP-hard. This can be seen in relation to Håstad's result [16] showing that tensor rank is NP-hard and thus finding the minimum bilinear program for computing bilinear forms is NP-hard.

In section 3.1.2 the NP-hardness result is used to prove a special case of the problem MAX SNP-Complete [23] (and also APX-Complete). This means there are no ϵ-approximation algorithms for the problem unless P=NP [1].

A linear straight-line program over GF(2) is said to be a *cancellation-free straight-line program* if, for every line of the program $u := v + w$, none of variables in the expression for v are also present in the expression for w, i.e., there is no cancellation of variables in the computation. A small example showing that the optimal linear program is not always cancellation-free over $GF(2)$ is:

$$x_1 + x_2; \quad x_1 + x_2 + x_3; \quad x_1 + x_2 + x_3 + x_4; \quad x_2 + x_3 + x_4.$$

It is not hard to see, by exhaustive search, that the optimum cancellation-free straight-line program has length 5. A solution of length 4 which allows cancellations is

$$v_1 = x_1 + x_2; \quad v_2 = v_1 + x_3; \quad v_3 = v_2 + x_4; \quad v_4 = v_3 + x_1.$$

In subsection 3.1.3 we show that the approximation ratio for cancellation-free techniques is at least 3/2. This discovery led us to create the heuristic in section 3.1, allowing cancellations, for minimizing linear straight-line programs and the corresponding circuits.

2 First Step

We will illustrate the first step of the circuit minimization using AES's S-box as an example. The non-linear operation in AES's S-box is to compute an inverse in the field $GF(2^8)$. A recursive method for building a circuit for inverses in $GF(2^{mn})$, given a circuit for inverses in $GF(2^m)$, is due to Itoh and Tsujii [17]. The circuits produced by this method are said to have a *tower fields architecture*. Since there are multiple possible representations for Galois fields, several authors have concentrated on finding representations that yield efficient circuits under the tower fields architecture. We use the same general technique for the reduction from inversion in $GF(2^8)$ to $GF(2^4)$ inversion, but we use a completely different technique for computing the inversion in $GF(2^4)$. We then place the optimized circuit for $GF(2^4)$ inversion in its appropriate place in AES's S-box and, in the second step, apply a novel optimization technique to the linear parts of the resulting circuit.

2.1 $GF(2^4)$ Inversion – A Non-Linear Component.

The tower fields architecture for inversion in $GF(2^8)$ has (non-trivial) easily identifiable non-linear components corresponding to inversion in subfields. The first step in our method is to focus on one of these components and derive a circuit that uses few $\wedge$ gates. The component for inversion in $GF(2^2)$ is too small for us to benefit significantly from optimizing it. Instead we focus on inversion in $GF(2^4)$. There are many representations of $GF(2^4)$. We construct

- $GF(2^2)$ by adjoining a root W of $x^2 + x + 1$ over $GF(2)$;
- $GF(2^4)$ by adjoining a root Z of $x^2 + x + W^2$ over $GF(2^2)$.

Following Canright [10], we represent $GF(2^2)$ using the basis (W, W^2) and $GF(2^4)$ using the basis (Z^2, Z^8). Thus, an element $\delta \in GF(2^4)$ is written as $\delta_1 Z^2 + \delta_2 Z^8$, where $\delta_1, \delta_2 \in GF(2^2)$. Similarly, an element γ in $GF(2^2)$ is written as $\gamma_1 W + \gamma_2 W^2$, where $\gamma_1, \gamma_2 \in GF(2)$. Since Z satisfies $x^2 + x + W^2 = 0$ and W satisfies $x^2 + x + 1 = 0$, one can calculate that $Z^4 = Z^2 + W$, $Z^8 = Z^2 + 1$ $(1 = Z^8 + Z^2)$, $Z^{10} = Z^4 + Z^2$, $Z^{16} = Z^8 + W$, $W^3 = W^2 + W$, $W^4 = W$, and $W^5 = W^2$. These equations can be used to reduce expressions to check equalities.

Using this representation, an element of $GF(2^4)$ can be written as $\Delta = (x_1 W + x_2 W^2)Z^2 + (x_3 W + x_4 W^2)Z^8$, where $x_1, x_2, x_3, x_4 \in GF(2)$. The inverse of this element, $\Delta' = (y_1 W + y_2 W^2)Z^2 + (y_3 W + y_4 W^2)Z^8$, can then be calculated using the following polynomials over $GF(2)$:

- $y_1 = x_2x_3x_4 + x_1x_3 + x_2x_3 + x_3 + x_4$
- $y_2 = x_1x_3x_4 + x_1x_3 + x_2x_3 + x_2x_4 + x_4$
- $y_3 = x_1x_2x_4 + x_1x_3 + x_1x_4 + x_1 + x_2$
- $y_4 = x_1x_2x_3 + x_1x_3 + x_1x_4 + x_2x_4 + x_2$

The fact that Δ' is the inverse of Δ can be verified by multiplying the two elements together and reducing using the equations mentioned above (along with $x^2 = x$ and $x + x = 0$). The symbolic result is $(QW + QW^2)Z^2 + (QW + QW^2)Z^8$, where $Q = x_1x_2x_3x_4 + x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4 + x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4 + x_1 + x_2 + x_3 + x_4$. The fact that the value of Q is 1 unless all four variables have the value 0, when it is 0, can be seen by observing that it is the symmetric function $\Sigma^4_4 + \Sigma^4_3 + \Sigma^4_2 + \Sigma^4_1$. If exactly four variables are set, then the first term gives the value 1 (and the others 0); if three are set, then the second, third and fourth terms give the value 1; if exactly two are set, then only the third gives the value 1; and if only one is set, then only the last gives the value 1. Hence, the result is 1, except for the zero input.[5]

Thus the task at hand is to construct a circuit with four inputs and four outputs that calculates the above system of equations using as few $\wedge$ gates as possible. Currently, our heuristic search programs can handle functions with one output and up to eight inputs. (Since they are heuristics, one is not certain that an output is optimal, so they cannot be used, for example, to determine a tight lower bound for the multiplicative complexity of E^8_4.) This means that we can directly construct optimal circuits for each of the four

[5]A circuit for finite field inversion must have some output for the non-invertible zero element. In the following constructions we follow the AES convention that the output on input zero is zero.

$$\begin{array}{lclllclllcll}
t_1 & = & x_1 + x_2 & & t_2 & = & x_1 \times x_3 & & t_3 & = & x_4 + t_2 & \\
t_4 & = & t_1 \times t_3 & & y_4 & = & x_2 + t_4 & (*) & t_5 & = & x_3 + x_4 & \\
t_6 & = & x_2 + t_2 & & t_7 & = & t_6 \times t_5 & & y_2 & = & x_4 + t_7 & (*) \\
t_8 & = & x_3 + y_2 & & t_9 & = & t_3 + y_2 & & t_{10} & = & x_4 \times t_9 & \\
y_1 & = & t_{10} + t_8 & (*) & t_{11} & = & t_3 + t_{10} & & t_{12} & = & y_4 \times t_{11} & \\
y_3 & = & t_{12} + t_1 & (*) & & & & & & & &
\end{array}$$

Figure 2: Inversion in $GF(2^4)$.

equations individually, but not for the system itself. For the full system we took the following approach:

1. pick an equation and construct an efficient circuit for it;
2. store intermediate functions computed in the previous steps for possible use in constructing a circuit for the next equation to be tackled;
3. iterate until all equations have been computed.

The first step is non-trivial even for predicates on few inputs. The heuristic we used is inspired by methods from automatic theorem proving [7]: consider an arbitrary predicate f on n inputs. We refer to the last column of the truth table for f as the *signal* of f. The columns in the truth table corresponding to each of the inputs to f are *known* signals. A search for a circuit for f starts with this set S of known signals. If u, v are known signals for functions g, h respectively, then the bit-wise XOR (AND) of u and v is the signal for the predicate $g \oplus h$ $(g \wedge h)$. We can *grow* the set S by adding the XOR of randomly chosen signals. We call this step an *XOR round.* The analogous step where the AND of signals is added to S is called an *AND round.* Each round is parameterized by the number of new signals added and the maximum number of AND gates allowed. In either an XOR round or an AND round, two signals are not combined if doing so creates a signal with more AND gates than is allowed. The heuristic alternates between XOR and AND rounds until the target signal is found or the set S becomes too large. In the latter case, since this is a randomized procedure, we start again. Various enhancements and optimizations have been implemented. Their description is outside the scope of this paper. We can report, however, that we succeeded in determining the multiplicative complexity of all 2^{16} predicates on four bits. It turns out that 3 multiplications are enough to compute any predicate on four variables.[6] This is of interest to designers of cryptographic functions since many constructions have been proposed which use 4x4 S-boxes. We have not yet been able to do the same for all predicates on 5 bits.

We performed the three steps above for each of the twenty-four orderings of $\{y_1, y_2, y_3, y_4\}$. The ordering (y_4, y_2, y_1, y_3) gave the best results. The resulting circuit, expressed as a straight-line program over GF(2), is shown in Figure 2 (outputs are indicated by an (*)).

This circuit contains 5 $\wedge$ gates and 11 $\oplus$ gates. It is a significant improvement over previous constructions, e.g. Paar's construction [21] has a gate count of 10 $\wedge$ gates and 15

[6]Lest the reader think this trivial, he/she may attempt to compute the function $f(x_1, x_2, x_3, x_4) = x_1x_2x_3x_4 + x_1x_2x_3 + x_1x_2x_4 + x_2x_3x_4 + x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_3x_4$ using only three multiplications.

$\oplus$ gates for the same function. It is harder to compare to Canright's construction [10]. In his original, he had 9 $\wedge$ gates (and NAND gates) and 14 $\oplus$ gates (and XNOR gates), but he optimized, allowing NOR gates. After this, he had 8 NAND gates, 2 NOR gates, and 9 XOR/XNOR gates.

Under the given representation for $GF(2^4)$, the multiplicative complexity of inversion is 5. This can be argued as follows: the upper bound is given by the construction. The four outputs that have to be computed all have degree 3. One $\wedge$ is needed to compute a polynomial of degree 2. Then, an additional $\wedge$ is necessary to produce each of the four linearly independent polynomials, since each is of degree 3.

2.2 A View of the Structure of AES's S-Box.

In the previous section, using the tower fields architecture, we identified and optimized (with respect to multiplicative complexity) a major non-linear component in an implementation of the AES S-box. That completes the first step of our technique for circuit optimization, but in other circuits, one may be able to identify more non-linear components with few enough inputs that they can also be optimized before continuing. At this point, we replaced the $GF(2^4)$ inversion subcircuit, in Canright's [10] (already optimized) circuit, with the subcircuit in Figure 2. As expected, the resulting circuit contained large linear connected components. In fact, from a cryptanalyst's point of view, the topology of the resulting circuit is potentially of interest: the S-box of AES consists of an initial linear expansion U from 8 to 22 bits, followed by a non-linear contraction F from 22 to 18 bits, and ending with a linear contraction B from 18 to 8 bits. The U and B matrices are given below. AES's S-box is $S(\mathbf{x}) = B \cdot F(U \cdot \mathbf{x}) + [11000110]^T$, where $\cdot$ is matrix multiplication and $\mathbf{x}$ is the 8-bit S-box input. We do not know if there are any cryptanalytic implications to the structure of these matrices. The first row and last columns of U should raise an eyebrow, as should the 12^{th} and the last three columns of B. Note that the initial linear expansion and the linear contraction were defined to contain as much of the circuit as possible while still being linear, increasing the portion of the circuit which could be further optimized by concentrating on the linear components. Thus, the portion of the circuit defined by U, for example, overlaps with the $GF(2^8)$ inversion. Also included in these linear components is the linear transformation to change bases, before computing the inverse in $GF(2^8)$, plus the linear transformation to change back to the original basis, followed by the affine transformation which is the final operation in the S-box.

$$U = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 0
\end{bmatrix}$$

$$B = \begin{bmatrix}
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0
\end{bmatrix}$$

3 Second Step

The second step is to optimize the linear components in the circuit. One method of finding the nonlinear components to be optimized in the first step was to find maximal linear components of the circuit, remove them, and look at the remaining nonlinear components. Whether this was done or not, after the optimized nonlinear components are inserted into their appropriate places in the circuit, the beginning of the second step should be to find maximal linear components in this new circuit (since after optimization, some of the nonlinear portions may contain $\oplus$ gates which can be included in the "old" linear parts, as in the case of the U and B matrices from AES's S-box).

These maximal components define linear components of the circuit which should be minimized in Step 2. In the case of the AES S-box, the top-linear component corresponds to the matrix U, and the bottom-linear component corresponds to the matrix B. No other significant linear components were found. After finding these, the next step was to minimize the circuits for computing U and B.

3.1 Minimizing Linear Components

First, we show that the problem of linear circuit minimization, or equivalently, Shortest Linear Program (SLP), is NP-hard.

3.1.1 NP-Hardness

The problem SHORTEST LINEAR PROGRAM (SLP) is as follows: Given a set of linear forms E over a field F, find a shortest linear program to compute E.

In order to prove NP-hardness, we consider the corresponding decision problem, SLPd: Given a set of linear forms E over a field F and a positive integer k, determine if there exists a straight-line linear program with at most k lines which computes E.

We will prove SLPd NP–hard, even if the constants in the set of linear forms to be computed are only zeros and ones. Furthermore, if the field F is finite, then SLPd is easily seen to be in NP, so SLPd is NP–complete over finite fields.[7]

The interest of this section is not just in the final result that SLP is NP–hard, but also in the method used to prove it. In particular, most of this section is devoted to the proof of Lemma 1, which gives the exact complexity for sets of linear forms of a certain simple type. This proof is *algorithmic* in form, and its algorithmic nature can be exploited to prove a further result in subsection 3.1.2.

In order to show NP-hardness, we reduce from VERTEX COVER. A *vertex cover* of a graph $G = (V, E)$ is a subset V' of V such that every edge of E is incident with at least one vertex of V'. VERTEX COVER is defined as follows: Given a graph $G = (V, E)$ and an integer k, determine if there exists a vertex cover of size at most k.

The following polynomial-time reduction f transforms an arbitrary graph, $G = (V, E)$, and a bound, k, to a set of linear forms with another bound, $\bar{k}$. The input variables are $X = V \cup \{z\}$, where z is a distinguished variable not occurring in V. The linear forms are $\bar{E} = \{\, z + a + b \mid (a, b) \in E \,\}$, and the program length we ask about is $\bar{k} = k + |\bar{E}|$. This is an instance of SLPd, and it is clear that $f(G, k) = (\bar{E}, X, \bar{k})$ can be produced in polynomial time. We call a set of linear expressions in this restricted form, $z + x_i + x_j$, a set of **z-expressions**.

Before we proceed, we illustrate with an example:

The graph, G, in Figure 3 has a vertex cover of size $k = 3$: $\{a, c, e\}$. The corresponding instance of SLPd, $f(G, 3)$ is $\bar{E} = \{z + a + b, z + b + c, z + c + d, z + d + e, z + e + f, z + a + f, z + c + g, z + e + g\}$, $X = \{z, a, b, c, d, e, f, g\}$, and $\bar{k} = 3 + 8$. A linear program for this of size 11 is

$$\begin{array}{llll} v_1 := z + a; & v_2 := z + c; & v_3 := z + e; & v_4 := v_1 + b; \\ v_5 := v_2 + b; & v_6 := v_2 + d; & v_7 := v_3 + d; & v_8 := v_3 + f; \\ v_9 := v_1 + f; & v_{10} := v_2 + g; & v_{11} := v_3 + g; & \end{array}$$

[7]We avoid the discussion of models for dealing with infinite fields, such as in [27] or [3], by proving NP-hardness when the constants in the forms are only zeros and ones and showing that a shortest linear straight-line program for the forms considered can be created with only zeros and ones as constants.

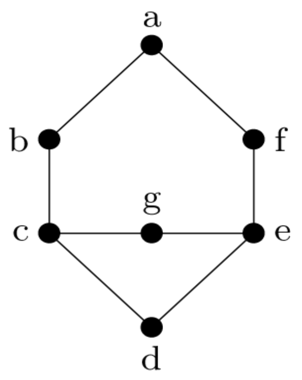


Figure 3: Graph with 8 edges and cover size 3.

where the computation of v_1, v_2, and v_3 corresponds to the vertex cover in the graph G, and the remaining operations produce the eight forms in $\bar{E}$. The variables $v_4, \ldots, v_{11}$ are called *output variables*.

A *cover* for a set $\bar{E}$ of z-expressions is a subset W of $X - \{z\}$ such that every expression in $\bar{E}$ contains at least one variable in W. Note that if $(\bar{E}, X, \bar{k}) = f(G, k)$, a cover for $\bar{E}$ trivially defines a vertex cover for the graph G and vice versa.

Lemma 1 *Let $(\bar{E}, X)$ be a set of z-expressions without repetitions; that is, $\bar{E}$ is a set of expressions of the form $z + x_i + x_j$, where x_i, x_j are distinct variables in X, z is a distinguished variable in X, and no two of these z-expressions contain exactly the same variables. There is a cover of $\bar{E}$ of size at most k if and only if there is a linear straight-line program P for $\bar{E}$ of length $\bar{k} = k + |\bar{E}|$. In addition, given a linear straight-line program P for $\bar{E}$, a cover for $\bar{E}$ of size at most $|P| - |\bar{E}|$ can be computed in polynomial time.*

Proof. We will refer to the elements of $X - \{z\}$ as "the variables" and z as "the symbol", although as an element of a linear program, z is also an input variable.

Given a cover W of size k for $\bar{E}$, a (cancellation-free) linear straight-line program for $\bar{E}$ can be created consisting of $z + w_i$ for each $w_i \in W$, followed by linear expressions computing each output, created by adding a second variable to the appropriate $z + w_i$. This program has length $k + |\bar{E}|$.

It remains to be shown that, given a linear straight-line program P for $\bar{E}$, we can efficiently find a cover, W, for $\bar{E}$ of size no more than $|P| - |\bar{E}|$. This cover is computed by associating elements of $X - \{z\}$ with some non-output lines of the program—W will then be the union of all those variables so associated. Since we will assign at most one element of $X - \{z\}$ to each non-output line, the cover is of size at most $|P| - |\bar{E}|$. (Note that the W computed may not be minimum if the program is not optimal).

Let $F^{(i)}$ be the linear function computed at line i; the result there is assigned to v_i. It will be convenient to use the notation $F^{(i)}$ to refer both to the function and to the minimal formal expression $\sum_j \beta_{i,j} x_j$ where the x_j's are distinct and the $\beta_{i,j}$'s are non-zero field elements. The association of variables with lines of the program will be denoted by a mapping $m : I\!N \rightarrow X \cup \{\lambda\}$. Initially, we set $m(i) = \lambda$ for all lines i. At any point in time, the current partial cover W is the set of all variables that are assigned to some $m(i)$.

$$W = \{x \in X \mid x = m(j) \text{ for some } 1 \leq j \leq |P|\}.$$

```
$W \leftarrow \lambda$
for $i = 1$ to $|P|$ do
    $m(i) \leftarrow \lambda$
    if $F^{(i)}$ is not an output then
        if $\exists$ a variable $x$ in $F^{(i)}$, but not in $W$ then
            choose $x$
            $m(i) \leftarrow x$
    else { $F^{(i)}$ is an output}
        if $F^{(i)}$ contains more than one variable not in $W$ then
            Fix-up$(i, i)$
```

Figure 4: Computing the cover W

The algorithm works as follows. Starting at the first line of the linear straight-line program P, the algorithm associates with each non-output line i a variable in $X - \{z\}$ which occurs in the formal expression computed at line i and which is currently unassociated (if there is no such variable, the line is assigned the null symbol, λ). When an output is reached, the algorithm checks if the set W of all variables currently assigned covers that output, i.e. if there is some variable in W which occurs in the formal expression computed at that output line. If this is not the case, then a fix-up procedure is invoked. This fix-up procedure changes some of the associations until all the output expressions up to that point are covered. After the algorithm has terminated, all the output expressions will be covered, so W is the desired cover, and $|P| \geq |W| + |\bar{E}|$. If the straight-line program P is restricted to being cancellation-free, the fix-up procedure will never be necessary; it is only called if an output line was produced as a linear combination of two lines, where at one of those lines a cancelled variable was added to the cover, W.

The remainder of the proof first establishes the precise conditions under which the fix-up procedure is called, and then describes the action taken. We first define the two properties that the algorithm seeks to establish for each line l of the program.

Property 1 If line l is not an output, either all variables in $F^{(l)}$ are in W, or some variable in both W and $F^{(l)}$ is associated uniquely with line l.

Property 2 There is at most one variable in $F^{(l)}$ which is not in W.

In terms of these two properties, the algorithm in Figure 4 can be described as follows. Given that Properties 1 and 2 hold for lines 1 to $i-1$, establish Property 1 for line i and check if Property 2 holds for line i. If not, the fix-up procedure will be called.

Claim 1 *If Properties 1 and 2 hold for lines 1 through $l-1$, after line l is processed, if $F^{(l)}$ is not an output, then Properties 1 and 2 hold for line l.*

Proof. This holds by induction. We define $v_0 = z$ and $v_{-i} = x_i$ for each variable x_i and note that Property 2 holds for these initial lines of the program. Line 1 of P contains at most two variables and cannot be an output. Thus, one of these variables is assigned to $m(1)$ and W, so Properties 1 and 2 hold for line 1. Suppose line i has the form $v_i := \lambda \cdot v_{i'} + \mu \cdot v_{i''}$. By assumption, Property 2 holds for $F^{(i')}$ and $F^{(i'')}$, so there are at most two variables in $F^{(i)}$ but not in W before line i is processed. If there is at most one such variable, we are done. If there are exactly two such variables, then $m(i)$ is assigned some variable not in

```
**Fix-up**$(i, l)$
{ $i$ is the current line being fixed; $l$ is original line being fixed }
{ line $i$, $v_i := \lambda v_{i'} + \mu v_{i''}$, produces the expression $z + a + b$,
   $a$ is not present in line $i'$ and $b$ is not present in $i''$}
{ line $i'$ is not an output and $m(i') = c$}
set $m(i') \leftarrow b$
$W \leftarrow (W \cup \{b\}) \setminus \{c\}$
for $j \leftarrow 1$ to $l$ do
    if $F^{(j)}$ is not an output then
        if $m(j) = \lambda$ and $c \in F^{(j)}$ then
            $m(j) \leftarrow c$
            $W \leftarrow W \cup \{c\}$
            break { exit for loop }
for $j \leftarrow 1$ to $l$ do
    if $F^{(j)}$ is an output then
        if $|F^{(j)} \setminus W| > 1$ then Fix-up$(j, l)$
```

Figure 5: The Fix-up procedure

W. Thus, Property 1 holds. Since that variable is also added to W, $F^{(i)}$ has at most one variable that does not occur in W and Property 2 also holds. □

Suppose the Fix-up procedure is called for line i:

$$v_i := \lambda \cdot v_{i'} + \mu \cdot v_{i''},$$

which produces the output expression $z+a+b$. If both a and b are present in the expression for line i' or both are in the expression for line i'', then at least one of a or b is in W. Since neither is there, we may assume, without loss of generality, that a is not present in line i' and b is not present in i''. All other variables present in line i' must also be present in line i''. In addition, at least one of those two lines is not an output, since otherwise one contains $z+a$ and the other contains $z+b$ and no linear combination is an output. Assume that line i' is not an output. Since it contains b, which is not in W, $m(i') \neq \lambda$. Suppose $m(i') = c$.

The fix-up procedure, as defined in Figure 5, backs up to line i', changes the mapping m to put b into W instead of c, and then scans to ensure, first, that Property 1 still holds, and then that Property 2 still holds. Since this change in the mapping may upset lines where these properties held previously, this adjustment of the mapping may occur more than once.

Since $b \notin W$, there is no problem in setting $m(i')$ to b. The only way this can cause Property 1 to fail is that a line j might have $m(j) = \lambda$, even though the expression there involved a c which is no longer in W. The first "for" loop in "Fix-up" corrects this.

The removal of c from W may also cause Property 2 to fail. Note that this can only happen for an output line; Claim 1 still holds. Some of the failures at outputs may be rectified by the adjustment fixing Property 1. "Fix-up" is called recursively to fix the others.

We turn to the proof of termination.

Let $k_1, k_2, \ldots$ be the sequence of line numbers for output lines which require a call to the fix-up procedure, and let $W_1, W_2, \ldots$ be the corresponding values of W, the covers just before the fix-up procedure is called for the corresponding lines. Let $k_0 = i$ and define W_0 to be the value of the cover when the fix-up procedure is first called. Note that no two adjacent members of $k_0, k_1, k_2, \ldots$ are equal.

Let j be an index for which $k_j < k_{j+1}$ (if no such index exists, the sequence is clearly finite and this terminates). We claim that $|W_j| < |W_{j+1}|$. The size of the cover never decreases as the only operations done to change it are swaps and additions, so the claim follows if we show that a variable is added to the cover by the fix-up procedure when going from line k_j to k_{j+1}.

Consider how the fix-up procedure operates between the calls at lines k_j and k_{j+1}. Suppose that line k_j is

$$v_{k_j} := \lambda \cdot v_{k'_j} + \mu \cdot v_{k''_j}$$

We know that $k'_j < k''_j < k_j < k_{j+1}$. Suppose the formal expressions computed at these lines are

$$\begin{array}{rclrcl} F^{(k'_j)} & = & (b-c-V)/\lambda; & F^{(k''_j)} & = & z+a+c+V; \\ F^{(k_j)} & = & z+a+b; & F^{(k_{j+1})} & = & \ldots, \end{array}$$

where V is a sum of some variables (not including z, a, b, c). (We will assume that k' played the role of i' in “Fix-up”, but the same argument holds if k'' was, with a and b switching roles.) For line k_j to have caused a call to “Fix-up”, neither a nor b could have been in the cover W_j. Thus the algorithm first visited line k'_j and changed the mapping $m(k'_j)$ from c to b, then executed the first “for” loop, correcting lines not satisfying Property 1, and finally moved down the program, checking each line for Property 2, until reaching line k_{j+1}. But this means that Property 2 held at line k''_j and this could only have happened if a or c was in the cover (if line k''_j is not an output, it might have been added there). Since neither of them were in the cover immediately after the swap of b for c at line k'_j, one of them must have been added by the fix-up procedure at one of the lines in between. Thus $|W_j| < |W_{j+1}|$.

Hence for each j where $k_j < k_{j+1}$, the size of the cover increases. Moreover, since k is always positive, there can be at most n^2 lines visited between these increases in the cover size (where n is the length of the program). And since $|W| < |X| \leq n$, it follows that the whole algorithm requires at most $O(n^3)$ time. (The fact that the execution time is polynomial is irrelevant for the purposes of showing NP-hardness, but will be important later.) This completes the proof of Lemma 1. □

The following theorem follows immediately, since we have given a polynomial time reduction from VERTEX COVER, which is NP-complete.

Theorem 1 *For any field $\mathbb{F}$, SHORTEST LINEAR PROGRAM is NP-hard.*

For finite fields, it is easy to see that SLPd $\in$ NP. Thus we have

Theorem 2 *For any finite field $\mathbb{F}$, the decision version of SHORTEST LINEAR PROGRAM is NP-Complete.*

Note that in the proof of Lemma 1, if the straight-line program P had been restricted to be cancellation-free, the proof would have been easier, because the fix-up procedure would never be necessary; it is only called if an output line was produced as a linear combination of two lines, where at one of those lines a cancelled variable was added to the cover, W. This immediately gives us the following:

Theorem 3 *For any finite field $\mathbb{F}$, SHORTEST LINEAR PROGRAM is NP-Complete even if the programs produced are restricted to being cancellation-free.*

3.1.2 Limits to Approximation

The major result of the previous subsection is that it is NP–hard to find an optimal linear program for computing a set of linear forms. Thus, it is natural to turn our attention to approximation algorithms for this problem. Here we concentrate entirely on polynomial-time approximation algorithms with provable performance guarantees.

We show that SHORTEST LINEAR PROGRAM has no ϵ–approximation scheme unless P=NP. Recall that these are families of algorithms, one for each $\epsilon > 0$, which are polynomial time and achieve an approximation ratio of $1+\epsilon$. We use a concept called Max SNP–completeness, which was introduced by Papadimitriou and Yannakakis [23]. Arora et.al. [1] have shown that no Max SNP–complete problem has an ϵ–approximation scheme unless P=NP. We show that BOUNDED Z-EXPN (defined below), is Max SNP–complete, showing that there is no ϵ–approximation scheme for SHORTEST LINEAR PROGRAM unless P=NP, since it is a generalization of BOUNDED Z-EXPN.

Max SNP is a complexity class of optimization problems. It is contained within NP in the sense that the decision versions of the problems are all in NP. Papadimitriou and Yannakakis [23] proved that many problems are Max SNP–complete, including the following: BOUNDED VERTEX COVER: Given a graph with maximum vertex degree bounded by a constant b, find the smallest vertex cover.

To talk about completeness for this class, we need a notion of reduction. The reductions Papadimitriou and Yannakakis defined, called **L-reductions**, preserve the existence of ϵ-approximation schemes. The following definitions and propositions are taken directly from the original paper.

Let Π and Π' be two optimization (maximization or minimization) problems, and let f be a polynomial-time transformation from problem Π to problem Π'. We say that f is an **L-reduction** if there are constants $\alpha, \beta > 0$ such that for each instance I of Π, the following two properties are satisfied:

(a) The optima of I and $f(I)$, written OPT(I) and OPT$(f(I))$ respectively, satisfy the relation $\text{OPT}(f(I)) \leq \alpha \text{OPT}(I)$.

(b) For any solution of $f(I)$ with cost c', we can find in polynomial time a solution of I with cost c such that $|c - \text{OPT}(I)| \leq \beta |c' - \text{OPT}(f(I))|$.

The constant β will usually be 1. The following two propositions, stated in [23], follow easily from the definition.

Proposition 1 *L-reductions compose.*

Proposition 2 *If Π L-reduces to Π' and if there is a polynomial-time approximation algorithm for Π' with worst-case error ϵ, then there is a polynomial-time approximation algorithm for Π with worse-case error $\alpha\beta\epsilon$.*

BOUNDED Z-EXPN is the following problem: Given a set of z-expressions (as defined in Theorem 1) in which each non-z variable appears at most b times (b is a fixed constant), generate an optimal linear program for computing the expressions (over some fixed field F).

Theorem 4 *BOUNDED Z-EXPN is* MAX *SNP–complete.*

Proof. First, we will show that BOUNDED Z-EXPN is in MAX SNP. To show membership in MAX SNP, we will exhibit an L-reduction of BOUNDED Z-EXPN to Bounded Vertex Cover, a problem in MAX SNP.

For every non-z variable x_i, we associate a vertex $\bar{x}_i$. The L-reduction f maps z-expressions to edges as follows: $f(\text{“}z + x_i + x_j\text{”}) = \text{“edge } (i,j)\text{”}$. Since variable occurrences are bounded by b in BOUNDED Z-EXPN, the vertex degrees will by bounded by b in the graph.

We proved in the previous section that a set of z-expressions can be optimally computed by first computing $z+x_i$ for those x_i which are in the minimum vertex cover, and then using these intermediate results to compute the z-expressions. Thus $\mathrm{OPT}(f(I)) + |E| = \mathrm{OPT}(I)$ where $|E|$ is both the number of z-expressions and the number of edges in the graph.

We claim that this reduction is an L-reduction. Property (a) is satisfied because the equation above implies that $\mathrm{OPT}(f(I)) \leq \mathrm{OPT}(I)$. Property (b) is satisfied because, from a vertex cover, we can build a linear program which computes the z-expressions in the manner described above. This gives $c = \mathrm{OPT}(I) + [c' - \mathrm{OPT}(f(I))]$.

To show that the problem is MAX SNP–hard we reverse the reduction so that it goes from Bounded Vertex Cover to Bounded Z-EXPN. The function f now maps “edge (i,j)” into “$z + x_i + x_j$”.

Proof of Property (a): By Lemma 1 we have that $\mathrm{OPT}(I) + |E| = \mathrm{OPT}(f(I))$. Since the maximum degree in the graph is bounded by b and every edge must be adjacent to at least one vertex of the cover, there can be at most $b \cdot \mathrm{OPT}(I)$ edges, of the cover. Thus $\mathrm{OPT}(f(I)) \leq (b+1)\mathrm{OPT}(I)$.

Proof of Property (b): The proof of Lemma 1 gave a polynomial-time procedure for converting any linear program computing a set of z-expressions into a vertex cover for the corresponding graph. By inspecting this procedure, one sees that $c = \mathrm{OPT}(I) + [c' - \mathrm{OPT}(f(I))]$. □

The fact that BOUNDED Z-EXPN is complete for the class MAX SNP implies that there is no ϵ-approximation scheme for it unless P=NP. In fact, Clementi and Trevisan [11] have shown that BOUNDED VERTEX COVER is not approximable within $16/15 - \epsilon$ for sufficiently large maximum degree. By Proposition 2, this means that there is no $1+(1/15-\epsilon)/\alpha\beta = 1 + (1/15 - \epsilon)/(1 + b)$-approximation algorithm for SLP unless P=NP. The fact that BOUNDED Z-EXPN is *in* the class MAX SNP means that there is an approximation algorithm for it with a constant approximation ratio. In fact, it is obvious that Z-EXPN, even without the boundedness constraint, has an approximation algorithm with a constant approximation. The straight-forward linear straight-line program for computing the $|E|$ forms only requires $2|E|$ lines, and every straight-line program for E must contain at least $|E|$ lines (assuming no repetitions within the set E). Thus, the straight-forward algorithm comes within a factor of 2 of optimal. Moreover, since there is an approximation algorithm for vertex cover which comes within a factor of two of optimal, we can do even better for Z-EXPN. Since the optimal linear program contains $|W|+|E|$ steps, where W is the minimum vertex cover, by Lemma 1, there is an algorithm which takes $2|W| + |E|$ steps. Since $|W| < |E|$, the ratio $(2|W|+|E|)/(|W|+|E|)$ is at most $3/2$, so there is a $(3/2)$-approximation algorithm for Z-EXPN. There are, however, no known approximation algorithms which obtain a constant ratio for the general SHORTEST LINEAR PROGRAM problem.

3.1.3 Cancellation can yield smaller circuits

Thus, unless P=NP, this problem does not even have efficient ϵ-approximation schemes, so our goal in this research is restricted to improving on known heuristics. As far as we know, the most successful heuristics are variations on a greedy algorithm due to Paar [22]. We report significant improvements over the latter methods. Paar's algorithm gives non-cancelling results. It keeps a list of variables 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 previous appeared. This can be repeated until everything has been computed. One possible variant of this was presented in the same article [22]: When there is more than one most frequently occurring pair, instead of selecting one, try all possibilities, using recursion. The original algorithm is very fast; the variant is not.

A different technique is due to Bernstein [2]. Bernstein's algorithm has the advantages of using less storage and functioning better on two-operand platforms, i.e., where $a := a \oplus b$ is an allowed operation, but $a := b \oplus c$ is not. However, experiments mentioned in [2] indicate that Bernstein's algorithm usually produces results with more gates than Paar's.

Previous work on circuit minimization for AES S-boxes (e.g. [21, 24, 10]) only consider cancellation-free straight-line programs for producing a set of linear forms over GF(2). Canright [10] even does an exhaustive search to find an optimal cancellation-free straight-line program. This does not, however, necessarily imply that Canright has found the optimal linear straight-line program. Some authors appear to make the incorrect assumption that there always exists a cancellation-free optimal linear program over GF(2).

As mentioned in the introduction, restricting the search for optimal straight-line programs for computing linear forms over GF(2) to cancellation-free programs can lead to sub-optimal solutions. In our counter-example, the optimal cancellation-free program has length $\frac{5}{4}$ times that of the true shortest program. It is natural to ask how close to optimal cancellation-free programs can get as the number of variables increases. In this subsection we show that the best cancellation-free straight-line programs are not guaranteed to even have length within a factor 3/2 that of the shortest straight-line linear program.

The following construction uses two integer parameters k and n, which can be made large to make the 3/2 inapproximability result hold asymptotically. The parameter k is the number of variables in a *block*, and n is the number of distinct blocks. Blocks have disjoint sets of variables: Block i, where $0 \leq i \leq n-1$, is the linear form $b_i = x_{ik+1} + x_{ik+2} + ... + x_{(i+1)k}$. The construction produces a linear straight-line program which is not cancellation-free. All intermediate linear forms (the linear forms produced at each line of the program) computed by this straight-line linear program will belong to the set of required outputs. The first part of the linear straight-line program will produce sums of consecutive pairs of blocks $s_i = b_i + b_{i+1}$, for $0 \leq i \leq n-2$, mixing the variables in the two blocks in such a way that also producing a single block alone would require extra additions compared to the program here. Then, pairs of these consecutive sums are computed, $p_i = s_i + s_{i+1}$, for $0 \leq i \leq n-3$. Each p_i is computed with only one further addition, but the two s_i's added share a common block which is cancelled, so $p_i = b_i + b_{i+2}$. We express this linear program, denoted P, using **for** loops in Figure 6, but for any fixed k and n it is a straight-line program of length $k(n-1)+(k-1)(n-1)+n-2 = 2kn-2k-1$.

We claim that an optimal cancellation-free program (for computing all the linear forms

```
for $i = 1$ to $k(n-1)$ do
    $u_i := x_i + x_{i+k}$
for $i = 0$ to $n-2$ do
    $s_i := u_{ik+1} + u_{ik+2}$
    for $j = 1$ to $k-2$ do
        $s_i := s_i + u_{ik+j+2}$
for $i = 0$ to $n-3$ do
    $p_i := s_i + s_{i+1}$
```

Figure 6: Straight-line program with cancellations

which are the result of some line in this program) does at least enough additional operations to compute each of the blocks, and this would require at least $n(k-1)$ additional lines. Let F denote the set consisting of the first $(2k-1)(n-1)$ lines of P, and let L denote the set of the last $n-2$ lines. All of the $2kn-2k-1$ lines output by the above straight-line program are linear forms which must be output. The lines in L are the only ones with cancellations. None of the results from the lines in F can be used to compute the lines in P, because, for any two lines $f \in F$ and $l \in L$, f contains at least one variable which is not present in the form calculated by l. It is conceivable that some of the non-output results computed in the process of producing the outputs in L could be used in computing those in F, but, since they are all outputs, at least one extra operation is needed to produce each output from F. Thus, we can consider computing the outputs in L independently from those in F.

Blocks b_2 through b_{n-3} each appear in two of the outputs from L, but there is no other overlap between the outputs in L. Thus, the only reuse of forms computed which is possible is within the blocks. An optimal way to compute the forms in L is to first compute each of the n blocks, using $k-1$ additions for each. After this, each form in L can be created by adding two blocks together, using one addition for each, as in P. The computation of the blocks gives $n(k-1)$ extra additions, for a total of $3kn-2k-n-1$ additions. Asymptotically, the ratio $\frac{3kn-2k-n-1}{2kn-2k-1}$ is $3/2$ for large n and k.

Theorem 5 *Any algorithm for computing short straight-line linear programs, which only produces cancellation-free straight-line programs, has an approximation ratio of at least* $3/2$.

Thus, even optimal cancellation-free circuits can be far from optimal in the unrestricted model. The heuristic we present below is not restricted to producing cancellation-free circuits. Furthermore, there appears to be little reason for restricting the search to cancellation-free circuits, as we have shown that finding an optimal cancellation-free circuit is also NP-hard in subsection 3.1.1.

A New Heuristic.

Let S be a set of linear functions. For any linear predicate f, we define the distance $\delta(S, f)$ as the minimum number of additions of elements from S necessary to obtain f.

The problem is to find a short linear program that computes $f(\mathbf{x}) = M\mathbf{x}$ where M is an $m \times n$ matrix over GF(2). The heuristic is as follows. We keep a "base" S of "known" functions. Initially S is just the set of variables $x_1, \ldots, x_n$. We maintain the vector $Dist[]$ of distances from S to the linear functions given by the rows of M. That is, $Dist[i] = \delta(S, f_i)$ where f_i is the i^{th} row of M multiplied by the input vector $\mathbf{x}$. Initially, $Dist[i]$ is just one less than the Hamming weight of row i. We then perform the following loop

- pick a new base element by adding two existing base elements;
- update $Dist[]$;

until $Dist[i] = 0$ for all i.

The current criterion for picking the new base element is

- pick one that minimizes the sum of new distances;
- resolve ties by *maximizing* the Euclidean norm of the vector of new distances.

This tie resolution criterion, which we term "Norm", may seem counter-intuitive. The basic idea is that we prefer a distance vector like 0,0,3,1 to one like 1,1,1,1. In the latter case, we would need 4 more gates to finish. In the former, 3 might do it.

The bulk of the time of the heuristic is spent on picking the new base element. Our experiments show that the following "pre-emptive" choice usually improves running time without increasing the size of the output circuit:

- if any two bases $S[i], S[j]$ are such that $S[i] \oplus S[j]$ is a row in M, then pick this sum as the new base element.

The tie resolution criterion is a critical part of the heuristic. It does well on most matrices we have tried, but we have found specific matrices for which other decision rules do better. Intuitively, no one simple rule should work for all matrices. The effectiveness of the heuristic most likely depends on the topology of the digraph represented by the input matrix. We have not pursued this line of inquiry. We have, however tested our heuristic with various tie resolution methods against Paar's algorithm [22]. On random matrices, our heuristic gives significant improvements under Norm as well as under three other tie-breaking rules (see Section 5),

The distance vector in our heuristics is computed by exhaustive search. The reason the heuristic is practical for moderate-size matrices is that the distance can only decrease. In fact, it can only decrease by 1. So when a new base is being considered, if a distance is d, then only combinations of exactly $d - 1$ old base elements and the new base element need to be considered.

A Small Example Using the Heuristic.

Suppose we need a circuit that computes the system of equations defined in Fig. 7, which is equivalent to finding a circuit for multiplication by the 6×5 matrix, M, given in the figure.

The *target signals* to be computed are simply the rows of M. The initial base is $\{x_0, x_1, x_2, x_3, x_4\}$, which corresponds to

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

$$\begin{aligned} y_0 &= x_0 + x_1 + x_2 \\ y_1 &= x_1 + x_3 + x_4 \\ y_2 &= x_0 + x_2 + x_3 + x_4 \\ y_3 &= x_1 + x_2 + x_3 \\ y_4 &= x_0 + x_1 + x_3 \\ y_5 &= x_1 + x_2 + x_3 + x_4 \end{aligned} \qquad M = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 7: Example sequence of equations and corresponding matrix.

Step 1 : $t_5 = x_1 + x_3$. (found signal = [0 1 0 1 0]). New D : [2 1 3 1 1 2].
Step 2 : $t_6 = x_2 + t_5$ (found target signal y_3 = [0 1 1 1 0]). New D : [2 1 3 0 1 1].
Step 3 : $t_7 = x_4 + t_6$ (found target signal y_5 = [0 1 1 1 1]). New D : [2 1 2 0 1 0].
Step 4 : $t_8 = x_0 + x_1$ (found signal = [1 1 0 0 0]). New D : [1 1 1 0 1 0].
Step 5 : $t_9 = x_0 + t_5$ (found target signal y_4 = [1 1 0 1 0]). New D : [1 1 1 0 0 0].
Step 6 : $t_{10} = x_2 + t_7$ (found target signal y_1 = [0 1 0 1 1]). New D : [1 0 1 0 0 0].
Step 7 : $t_{11} = x_2 + t_8$ (found target signal y_0 = [1 1 1 0 0]) . New D : [0 0 1 0 0 0].
Step 8 : $t_{12} = t_7 + t_8$ (found target signal y_2 = [1 0 1 1 1]). New D : [0 0 0 0 0 0]. (DONE!)

Figure 8: Example running heuristic for minimizing linear components.

The initial distance vector is

$$D = \begin{bmatrix} 2 & 2 & 3 & 2 & 2 & 3 \end{bmatrix}.$$

The heuristic must find two base vectors whose sum, when added to the base, minimizes the sum of the new distances. It turns out the right choice is to calculate $x_1 + x_3$. So the new base S is expanded to contain the signal

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The new distance vector is

$$D = \begin{bmatrix} 2 & 1 & 3 & 1 & 1 & 2 \end{bmatrix}.$$

The full run of the program is shown in Figure 8. The tie breaking criteria is used in Step 3. If one had chosen $x_0 + x_1$ instead of $x_4 + t_6$, the new distance vector would be $[\ 1\ \ 1\ \ 2\ \ 0\ \ 1\ \ 1\ ]$, which has norm $\sqrt{8}$, while the one found has norm $\sqrt{10}$. Note that there is cancellation in steps 6 and 8.

Thus, after the x_i, which may be nonlinear functions of other variables, are computed, the y_i are computed by following the algorithm produced and, in this case, letting $y_0 = t_{11}$, $y_1 = t_{10}$, $y_2 = t_{12}$, $y_3 = t_6$, $y_4 = t_9$, $y_5 = t_7$.

4 A Circuit for the S-Box of AES

Our techniques yield a circuit for the AES S-box composed of 115 gates in three parts: a "top" linear transformation, U; a middle non-linear part; and a "bottom" linear transformation, B. The linear transformations are defined by the matrices U and B of section 2.2.

$y_{14} = x_3 + x_5$	$y_{13} = x_0 + x_6$	$y_9 = x_0 + x_3$
$y_8 = x_0 + x_5$	$t_0 = x_1 + x_2$	$y_1 = t_0 + x_7$
$y_4 = y_1 + x_3$	$y_{12} = y_{13} + y_{14}$	$y_2 = y_1 + x_0$
$y_5 = y_1 + x_6$	$y_3 = y_5 + y_8$	$t_1 = x_4 + y_{12}$
$y_{15} = t_1 + x_5$	$y_{20} = t_1 + x_1$	$y_6 = y_{15} + x_7$
$y_{10} = y_{15} + t_0$	$y_{11} = y_{20} + y_9$	$y_7 = x_7 + y_{11}$
$y_{17} = y_{10} + y_{11}$	$y_{19} = y_{10} + y_8$	$y_{16} = t_0 + y_{11}$
$y_{21} = y_{13} + y_{16}$	$y_{18} = x_0 + y_{16}$	

Figure 9: Top linear transformation: Inputs are $x_0, x_1, ..., x_7$. Outputs to the next level are $x_7, y_1, y_2, ..., y_{21}$.

For the matrix U, the smallest circuits we found had 23 $\oplus$ gates. Among the many such circuits, the shortest ones have depth 7. It is worthwhile to note that if 24 $\oplus$ gates are allowed, circuits with depth 4 exist for U. Figure 9 shows a circuit of size 23 and depth 7. The circuit maps inputs $x_0 \dots x_7$ to outputs $x_7, y_1 \dots y_{21}$.

Figure 10 shows the non-linear middle part of the S-box circuit. It is a function from 22 to 18 bits. The circuit contains 32 $\wedge$ gates and 30 $\oplus$ gates. It maps inputs $x_7, y_1 \dots y_{21}$ to outputs $z_0 \dots z_{17}$.

For matrix B, the randomized version of our heuristic yields many circuits with 30 $\oplus$ gates. The heuristic is fast enough that we are able to pick a circuit which is both small and short. Figure 11 shows a circuit of depth 6. The circuit maps inputs $z_0 \dots z_{17}$ to outputs $s_0 \dots s_7$.

As mentioned earlier, our circuit was based on Canright's [10]. Our non-linear middle part corresponds fairly closely to his, except that his subcircuit for inversion in $GF2^4$ was replaced by ours. He does not consider all of the top linear transformation as one unit, but he uses 29 XOR/XNOR gates to compute the entire transformation. Similarly, he uses 31 XOR/XNOR gates to compute what corresponds to our bottom linear transformation. After optimizations, his circuit has a total of 80 XOR/XNOR gates, 34 NANDs, and 6 NORs. We did not attempt to use NOR gates to further reduce the size of our circuit.

5 Experiments with Different Tie–Breaking Methods

In order to compare the effects of using different tie-breakers, we tested our heuristics on matrices generated as follows

- We first chose a size (for example, 10×20 matrices, which represent 10 linear forms on 20 distinct variables);
- We then picked a *bias* ρ between 0 and 1;
- For each entry of the matrix, we set the bit to 1 with probability ρ and to 0 with probability $1 - \rho$. Thus ρ is the expected fraction of variables that appears in each linear form.
- Matrices with rows which are all zeros were discarded, as were matrices containing duplicate rows.

t_2	$=$	$y_{12} \times y_{15}$	t_3	$=$	$y_3 \times y_6$	t_4	$=$	$t_3 + t_2$
t_5	$=$	$y_4 \times x_7$	t_6	$=$	$t_5 + t_2$	t_7	$=$	$y_{13} \times y_{16}$
t_8	$=$	$y_5 \times y_1$	t_9	$=$	$t_8 + t_7$	t_{10}	$=$	$y_2 \times y_7$
t_{11}	$=$	$t_{10} + t_7$	t_{12}	$=$	$y_9 \times y_{11}$	t_{13}	$=$	$y_{14} \times y_{17}$
t_{14}	$=$	$t_{13} + t_{12}$	t_{15}	$=$	$y_8 \times y_{10}$	t_{16}	$=$	$t_{15} + t_{12}$
t_{17}	$=$	$t_4 + t_{14}$	t_{18}	$=$	$t_6 + t_{16}$	t_{19}	$=$	$t_9 + t_{14}$
t_{20}	$=$	$t_{11} + t_{16}$	t_{21}	$=$	$t_{17} + y_{20}$	t_{22}	$=$	$t_{18} + y_{19}$
t_{23}	$=$	$t_{19} + y_{21}$	t_{24}	$=$	$t_{20} + y_{18}$			
t_{25}	$=$	$t_{21} + t_{22}$	t_{26}	$=$	$t_{21} \times t_{23}$	t_{27}	$=$	$t_{24} + t_{26}$
t_{28}	$=$	$t_{25} \times t_{27}$	t_{29}	$=$	$t_{28} + t_{22}$	t_{30}	$=$	$t_{23} + t_{24}$
t_{31}	$=$	$t_{22} + t_{26}$	t_{32}	$=$	$t_{31} \times t_{30}$	t_{33}	$=$	$t_{32} + t_{24}$
t_{34}	$=$	$t_{23} + t_{33}$	t_{35}	$=$	$t_{27} + t_{33}$	t_{36}	$=$	$t_{24} \times t_{35}$
t_{37}	$=$	$t_{36} + t_{34}$	t_{38}	$=$	$t_{27} + t_{36}$	t_{39}	$=$	$t_{29} \times t_{38}$
t_{40}	$=$	$t_{25} + t_{39}$						
t_{41}	$=$	$t_{40} + t_{37}$	t_{42}	$=$	$t_{29} + t_{33}$	t_{43}	$=$	$t_{29} + t_{40}$
t_{44}	$=$	$t_{33} + t_{37}$	t_{45}	$=$	$t_{42} + t_{41}$	z_0	$=$	$t_{44} \times y_{15}$
z_1	$=$	$t_{37} \times y_6$	z_2	$=$	$t_{33} \times x_7$	z_3	$=$	$t_{43} \times y_{16}$
z_4	$=$	$t_{40} \times y_1$	z_5	$=$	$t_{29} \times y_7$	z_6	$=$	$t_{42} \times y_{11}$
z_7	$=$	$t_{45} \times y_{17}$	z_8	$=$	$t_{41} \times y_{10}$	z_9	$=$	$t_{44} \times y_{12}$
z_{10}	$=$	$t_{37} \times y_3$	z_{11}	$=$	$t_{33} \times y_4$	z_{12}	$=$	$t_{43} \times y_{13}$
z_{13}	$=$	$t_{40} \times y_5$	z_{14}	$=$	$t_{29} \times y_2$	z_{15}	$=$	$t_{42} \times y_9$
z_{16}	$=$	$t_{45} \times y_{14}$	z_{17}	$=$	$t_{41} \times y_8$			

Figure 10: The middle non-linear section: Inputs are $x_7, y_1, y_2, ..., y_{21}$. Outputs to the next level are $z_0, z_1, ..., z_{17}$. Note that the computation of t_{25} through t_{40} is the inversion in $GF(2^4)$.

t_{46}	$=$	$z_{15} + z_{16}$	t_{47}	$=$	$z_{10} + z_{11}$	t_{48}	$=$	$z_5 + z_{13}$
t_{49}	$=$	$z_9 + z_{10}$	t_{50}	$=$	$z_2 + z_{12}$	t_{51}	$=$	$z_2 + z_5$
t_{52}	$=$	$z_7 + z_8$	t_{53}	$=$	$z_0 + z_3$	t_{54}	$=$	$z_6 + z_7$
t_{55}	$=$	$z_{16} + z_{17}$	t_{56}	$=$	$z_{12} + t_{48}$	t_{57}	$=$	$t_{50} + t_{53}$
t_{58}	$=$	$z_4 + t_{46}$	t_{59}	$=$	$z_3 + t_{54}$	t_{60}	$=$	$t_{46} + t_{57}$
t_{61}	$=$	$z_{14} + t_{57}$	t_{62}	$=$	$t_{52} + t_{58}$	t_{63}	$=$	$t_{49} + t_{58}$
t_{64}	$=$	$z_4 + t_{59}$	t_{65}	$=$	$t_{61} + t_{62}$	t_{66}	$=$	$z_1 + t_{63}$
s_0	$=$	$t_{59} + t_{63}$	s_6	$=$	t_{56} XNOR t_{62}	s_7	$=$	t_{48} XNOR t_{60}
t_67	$=$	$t_{64} + t_{65}$	s_3	$=$	$t_{53} + t_{66}$	s_4	$=$	$t_{51} + t_{66}$
s_5	$=$	$t_{47} + t_{65}$	s_1	$=$	t_{64} XNOR s_3	s_2	$=$	t_{55} XNOR t_{67}

Figure 11: Bottom linear transformation: Inputs are $z_0, z_1, ..., z_{17}$. Outputs are $s_0, s_1, ..., s_7$.

The testing was performed with a C++ program, compiled with g++ -O3, on a quadcore x86_64, running Ubuntu 9.10, with Intel Xenon 5150 processors running at 2.66 GHz, with 8 GB memory. There were no other users on the machine. The programs and matrices used can be found at www.imada.sdu.dk/∼joan/xor/, though minor changes are necessary to run the programs with different files as input or to change the matrix size and bias for the matrix generator. We compared the different heuristics on sets of one hundred random matrices with different sizes and densities. The experiments showed that the heuristics were slower when the bias was larger. This was expected, since the initial "distances" (number of operations on the base vectors to obtain the target vectors) were then larger on average when there were more ones in the matrices.

The tie-breakers we compared were the following:

- **Norm:** maximizing the Euclidean norm
- **Norm-largest:** maximizing the square of the Euclidean norm minus the largest distance
- **Norm-diff:** maximizing the square of the Euclidean norm minus the difference of the largest two distances
- **Random:** In processing the possible new base vectors, if the current possible new base vector has the same sum of distances as the previous best (current choice), then flip an unbiased coin. If heads, then keep the current choice. If tails, then apply the Norm criterion. This heuristic may end up choosing a pair with non-maximum Euclidean norm. On the other hand, it allows substitution of one optimum (by sum-of-distances and Euclidean norm) pair by another found later in the search.

In all cases, except the "Random" one, when there were still ties after applying the "tie-breaker", the first pair with both the minimum sum of distances and the optimal value for the tie-breaker was chosen. This was the base pair with lexicographically minimum indices (i, j). The exception to this is when there is a target with distance 1, meaning that using one extra gate will produce a target. Since it can never be wrong to use such a gate, a check is made for this case, by scanning the distances and choosing the first with distance 1 when such exists. This check is efficient, and when there is a target of distance one, it saves lengthy computations of new distances for each possible pair of bases.

Randomized tie-breaking allows running the heuristic several times and picking the best result. In our tests we ran the heuristic with "Random" tie-breaking three times.

We also compared these heuristics to Paar's heuristic [22] on the same matrices. Paar's heuristic repeatedly finds the most frequently occurring base pair and adds that as the next base pair. It is significantly faster than our heuristic, but it produces only cancellation-free circuits. Its performance, relative to the heuristics proposed here, decreases as the bias increases, using more than 30% extra gates when the bias is 3/4 (when the number of rows is at least 15) and 40% extra when the bias is 9/10.

Among the biases tried, the number of gates in the circuits found by our heuristics is similar with biases 1/2 and 3/4. It is not a strictly increasing function of the bias, since when nearly all of the variables are used in nearly all of the forms, the outputs from many of the gates can be reused for many targets. Thus, circuits with fewer gates were found when the bias was 9/10 than when it was 1/2 or 3/4. This was also true for Paar's heuristic, but less dramatically so.

All the tie resolution criteria performed fairly similarly, producing circuits of nearly the same size, with Random apparently doing slightly better (more often producing smaller circuits), presumably because it tries three different circuits and uses the best. Random also runs for about three times as long as the others. The results of these tests are presented in tables in the Appendix. In the tables, the column headings specify the matrix size and the bias. For each heuristic, and all matrix sizes and biases, 100 randomly chosen matrices were tested.

For each tie-breaker rule and Paar's heuristic, for each matrix size and bias, the average number of gates in the circuits found and the number of matrices where that heuristic did not obtain the minimum value of all of the heuristics was computed, along with the running time in seconds. The Paar heuristic was beaten by at least one of the other heuristics on all 700 matrices except for 17 of the 100 with bias 1/4 (and there was only one matrix on which Paar's heuristic beat any of the other heuristics). In fact, for the tests with bias larger than 1/4, Paar's heuristic did worse than any of the other heuristic on every one of the matrices; usually the values obtained for the newer heuristics were similar, with Random possibly being marginally better, but with the value for Paar's heuristic being significantly larger.

Paar's heuristic (and, for matrices between size 4 and 10, a variant which does at most one gate better on average in the data presented) was tested [22] on square matrices of sizes 4×4 through 16×16 and the average number of XOR gates is presented, along with the relative improvement over the straightforward implementation. These square matrices came from applying Mastrovito's [20] matrix description of multiplication in $GF(2^n)$ to constant multiplication. Paar tries all possible constants in $GF(2^n)$ for n between 4 and 16, giving these square matrices. Since our heuristics are so much slower and the matrices in the cryptographic applications we are interested in do not necessarily have this form, we have not tested on all of these restricted matrices of those sizes, but rather on random matrices with different biases. For 15×15 matrices, Paar gets an average of 52.9 gates. This is similar to our results for Paar's algorithm with 15×15 matrices with biases 1/2 and 3/4, where the Paar heuristic gets averages of 51.7 and 53.3 gates, respectively. For bias 1/2, our deterministic heuristics get average gate counts between 44.21 and 44.28, while Random gets 43.81. For bias 3/4, our deterministic heuristics all get average count 40.82, while Random gets 40.38. Thus, our relative improvement over the Paar heuristic is between 17% and 32% for these types of matrices. Paar's result of 52.9 gates for 15×15 matrices is a relative improvement of 45.5% over the straightforward approach.

The last row in each table in the Appendix shows the sums of the values which are the minimum of those calculated by the different heuristics for each matrix. This shows that for each of the tie-breakers, there are cases where it gets a worse result than at least one of the others.

6 Conclusions and Work in Progress

We developed and tested new techniques for decreasing circuit size. The techniques were applied to the extensively studied AES S-box. We obtained the smallest circuit yet constructed for this function. The circuit contains 32 AND gates and 83 XOR/XNOR gates for a total of 115 gates. As by-products of the experiment, we obtained very small circuits for inversion in $GF(2^4)$ and $GF(2^8)$.

The result that SHORTEST LINEAR PROGRAM is NP-hard indicates that using heuristic techniques is more realistic than expecting to find the smallest subcircuits for linear parts of a Boolean circuit. The result that a special case of SHORTEST LINEAR PROGRAM is MAX SNP-Complete indicates that there is a limit to how well these heuristic techniques can be guaranteed to do.

Since cancellation-free techniques can produce linear straight-line programs which are a factor 3/2 larger than the optimal, the heuristic developed here (in Step 2) is not restricted to cancellation-free operations.

The experiments with linear circuit optimization indicate that our techniques are likely to be superior to previous techniques which produced only cancellation-free circuits. We expect this to be particularly useful for cryptographic applications, both for hardware and software implementations, where many XOR operations are used, along with some AND operations to introduce nonlinearity.

It would be interesting to determine how close to optimal the circuits found by these techniques usually are and how much better they are than the optimal cancellation-free circuits. Finding even better techniques which are not restricted to finding cancellation-free circuits would also be very interesting.

Work on finding exact solutions using SAT-solvers has developed a technique which will quickly find a circuit with 23 gates, the same size we report here for our techniques, for the top linear transformation[14, 15]. They also prove that this cannot be achieved with 22 gates, so the number of gates used here for the top linear transformation is optimal.

Recent work has shown that the lower bound of 3/2 for the approximation ratio of cancellation-free straight-line programs can be improved to 2.

In practice, one would like to construct small low-depth circuits. This paper has discussed size only. However, it is plausible that a short circuit can be obtained by first minimizing size and then shortening the circuit along critical paths. Preliminary results using this technique are highly encouraging.

References

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the Association for Computing Machinery*, 45:501–555, 1998.

[2] D.J. Bernstein. Optimizing linear maps modulo 2. In *Workshop Record of SPEED-CC: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers*, http://cr.yp.to/papers.html#linearmod2.

[3] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bull. Amer. Math. Soc.*, 21:1–46, 1989.

[4] J. Boyar, P. Matthews, and R. Peralta. On the shortest linear straight-line program for computing linear forms. In *33nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2008)*, volume 5162 of *Lecture Notes in Computer Science*, pages 168–179, Springer, 2008.

[5] J. Boyar and R. Peralta. Tight bounds for the multiplicative complexity of symmetric functions. *Theoretical Computer Science*, 396(1-3):223–246, 2008.

[6] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In *9th International Symposium on Experimental Algorithms (SEA 2010)*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189, Springer 2010.

[7] J. Boyar and R. Peralta. Patent application number 61089998 filed with the U.S. Patent and Trademark Office. *A new technique for combinational circuit optimization and a new circuit for the S-Box for AES*, 2009.

[8] J. Boyar, R. Peralta, and D. Pochuev. On the multiplicative complexity of Boolean functions over the basis $(\wedge, \oplus, 1)$. *Theoretical Computer Science*, 235:43–57, 2000.

[9] P. Bürgisser, M. Clausen, and M.A. Shokrollahi. *Algebraic Complexity Theory.* Springer, 1997. chapter 13.

[10] D. Canright. A very compact Rijndael S-box. Technical Report NPS-MA-05-001, Naval Postgraduate School, 2005.

[11] A.E.F. Clementi and L. Trevisan. Improved non-approximability results for vertex cover with density constraints. In *Computing and Combinatorics*, pages 333–342, 1996.

[12] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[13] FIPS. *Advanced Encryption Standard (AES).* National Institute of Standards and Technology, 2001.

[14] C. Fuhs and P. Schneider-Kamp. Synthesizing shortest linear straight-line programs over GF(2) using SAT. In *13th International Conference on Theory and Applications of Satisfiability Testing*, volume 6175 of *Lecture Notes in Computer Science*, pages 71–84, Springer, 2010.

[15] C. Fuhs and P. Schneider-Kamp. Optimizing the AES S-Box using SAT. In *Proceedings of the 8th International Workshop on the Implementation of Logics*, 2010.

[16] Johan Håstad. Tensor rank is NP-Complete. *J. Algorithms*, 11(4):644–654, 1990.

[17] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.

[18] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17, 2009.

[19] O. B. Lupanov. A method of circuit synthesis. *Izvestia V.U.Z.*, Radiofizika(1):120–140, 1958.

[20] E. Mastrovito. *VLSI architectures for computation in Galois fields.* PhD thesis, Linköping University, Dept. Electr. Eng., Sweden, 1991.

[21] C. Paar. Some remarks on efficient inversion in finite fields, 1995. In 1995 IEEE International Symposium on Information Theory, page 58, Whistler, B.C. Canada.

[22] C. Paar. Optimized arithmetic for Reed-Solomon encoders. In *IEEE International Symposium on Information Theory*, page 250, 1997.

[23] C. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[24] A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A compact Rijndael hardware architecture with S-Box optimization. In *Advances in Cryptology - Proceedings of ASIACRYPT 01*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer-Verlag, 2001.

[25] J.E. Savage. An algorithm for the computation of linear forms. *SICOMP*, 3(2):150–158, 1974.

[26] C. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28:59–98, 1949.

[27] L.G. Valiant. Completeness classes in algebra. In *Proceedings of the 11th Annual ACM Symposium on the Theory of Computing*, pages 249–261, 1979.

[28] R. Williams. Matrix-vector multiplication in sub-quadratic time: (some preprocessing required). In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 995–1001, 2007.

[29] S. Winograd. On the number of multiplications necessary to compute certain functions. *Comm. Pure and Applied Math.*, 23:165–179, 1970.

Appendix: Experimental results on samples of 100 random matrices

	15×15 matrices, Bias=$\frac{1}{4}$			15×15 matrices, Bias=$\frac{1}{2}$		
Heuristic	Average	Not min	Seconds	Average	Not min	Seconds
Norm	29.65	16	12	44.21	48	125
Norm-largest	29.63	14	12	44.23	49	121
Norm-diff	29.65	15	11	44.28	51	119
Random	29.59	10	29	43.81	23	322
Paar	31.07	83	0.01	51.70	100	0.02
Minimum	29.48	0	-	43.50	0	-

	15×15 matrices, Bias=$\frac{3}{4}$			15×15 matrices, Bias=$\frac{9}{10}$		
Heuristic	Average	Not min	Seconds	Average	Not min	Seconds
Norm	40.82	47	291	30.28	31	388
Norm-largest	40.82	46	290	30.28	31	428
Norm-diff	40.82	46	292	30.29	32	388
Random	40.39	23	838	30.01	14	1145
Paar	53.27	100	0.03	43.11	100	0.02
Minimum	40.11	0	-	29.86	0	-

	20×20 matrices, Bias=$\frac{3}{4}$		
Heuristic	Average	Not min	Seconds
Norm	67.47	62	86,465
Norm-largest	67.43	60	82,597
Norm-diff	67.40	58	82,780
Random	66.87	30	234,815
Paar	90.86	100	0.11
Minimum	66.43	0	-

	20×10 matrices, Bias=$\frac{3}{4}$			10×20 matrices, Bias=$\frac{3}{4}$		
Heuristic	Average	Not min	Seconds	Average	Not min	Seconds
Norm	31.44	25	1.35	42.04	44	30,626
Norm-largest	31.43	24	1.38	42.08	44	30,490
Norm-diff	31.44	25	1.34	42.12	44	30,740
Random	31.23	11	4.08	41.76	22	84,540
Paar	43.32	100	0.02	50.02	100	0.02
Minimum	31.12	-	41.50	0	-	