Synthesizing Shortest Linear Straight-Line Programs over GF(2) using SAT*

Carsten Fuhs[1] and Peter Schneider-Kamp[2]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
fuhs@informatik.rwth-aachen.de
[2] IMADA, University of Southern Denmark, Denmark
petersk@imada.sdu.dk

Abstract. Non-trivial linear straight-line programs over the Galois field of two elements occur frequently in applications such as encryption or high-performance computing. Finding the shortest linear straight-line program for a given set of linear forms is known to be MaxSNP-complete, i.e., there is no ϵ-approximation for the problem unless $P = NP$.
This paper presents a non-approximative approach for finding the shortest linear straight-line program. In other words, we show how to search for a circuit of XOR gates with the minimal number of such gates. The approach is based on a reduction of the associated decision problem ("Is there a program of length k?") to satisfiability of propositional logic. Using modern SAT solvers, optimal solutions to interesting problem instances can be obtained.

1 Introduction

Straight-line programs over the Galois field of two elements, often denoted GF(2), have many practically relevant applications. The most prominent ones are probably in high performance computing (inversion of sparse binary matrices), networking and storage (error detection by checksumming), and encryption (hashing, symmetric ciphers).

In this paper, we focus on linear straight-line programs over GF(2) with applications in cryptography. The motivation behind this choice is that modern symmetric ciphers like AES can be implemented by lookup tables and addition in GF(2). Multiplication and addition in GF(2) correspond to the Boolean AND and XOR operations, respectively. In other words, we are looking at straight-line programs composed of array lookups and sequences of XOR operations.

The goal of this paper is, given a specification of a linear function from a number of inputs to a number of outputs, to find the shortest linear straight-line program over GF(2) that satisfies the specification. In other words, we show how to find a XOR circuit with the minimal number of gates that connects inputs to outputs. Finding such shortest programs is obviously interesting both for software and for hardware implementations of, for example, the symmetric cipher Advanced Encryption Standard (AES) [13].

* In *Proc. SAT'10*, LNCS, 2010. Supported by the G.I.F. grant 966-116.6 and the Danish Natural Science Research Council.

While there are heuristic methods for finding short straight-line linear programs [4] (see also [3] for the corresponding patent application), to the best of our knowledge, there is no feasible method for finding an optimal solution. In this paper, we present an approach based on reducing the associated decision problem ("Is there a program of length k?") to satisfiability of propositional logic. The reduction is performed in a way that every model found by the SAT solver represents a solution. Recent work [11] has shown that reductions to satisfiability problems are a promising approach for circuit synthesis. By restricting our attention to linear functions, we now obtain a polynomial-size encoding.

The structure of this paper is as follows. In Section 2, we formally introduce our optimization problem and show how linear straight-line programs can be used to compute a given set of linear forms. Section 3 presents a novel encoding for the associated decision problem to SAT. Then, we discuss in Section 4 how to tackle our optimization problem by reducing it to the associated decision problem using a customized search for k.

In Section 5 we present an empirical case study where we try to optimize an important component of AES. To prove optimality of the solution found, the case study prompts us to improve the performance of our encoding for the decision problem in the unsatisfiable case. For this, we discuss different approaches in Section 6. We conclude with a summary of our contributions in Section 7.

2 Linear Straight-Line Programs

In this paper, we assume that we have n inputs $x_1, \ldots, x_n$ and m outputs $y_1, \ldots, y_m$. Then the linear function to be computed can be specified by m equations of the following form:

$$\begin{aligned} y_1 &= a_{1,1} \cdot x_1 \oplus a_{1,2} \cdot x_2 \oplus \ldots \oplus a_{1,n} \cdot x_n \\ y_2 &= a_{2,1} \cdot x_1 \oplus a_{2,2} \cdot x_2 \oplus \ldots \oplus a_{2,n} \cdot x_n \\ &\ldots \\ y_m &= a_{m,1} \cdot x_1 \oplus a_{m,2} \cdot x_2 \oplus \ldots \oplus a_{m,n} \cdot x_n \end{aligned}$$

We call each equation a *linear form*. Note that each $a_{\ell,j}$ is a constant from GF(2) $= \{0, 1\}$, each x_j is a variable over GF(2), and $\oplus$ and $\cdot$ denote standard addition and multiplication on GF(2), respectively. In this paper, we always assume that the linear forms are pairwise different.

Our goal is to come up with an algorithm that computes these linear forms given $x_1, \ldots, x_n$ as inputs. More specifically, we would like to express this algorithm via a *linear straight-line program* (or, for brevity, just *program*). Here, every line of the program has the shape $u := e \cdot v \oplus f \cdot w$ with $e, f \in$ GF(2) and v, w variables. Some lines of the program will contain the output, i.e., assign the value of one of the desired linear forms to a variable. The *length* of a program is the number of lines the program contains. Without loss of generality, we perform write operations only to fresh variables, so no input is overwritten and no intermediate variable is written to twice. A program is *optimal* if there is no shorter program that computes the same linear forms.

Example 1. Consider the following linear forms:

$$\begin{aligned}
y_1 &= x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \\
y_2 &= x_1 \oplus x_2 \oplus x_3 \oplus x_4 \\
y_3 &= x_1 \oplus x_2 \oplus x_3 \oplus x_5 \\
y_4 &= x_3 \oplus x_4 \oplus x_5 \\
y_5 &= x_1 \oplus x_5
\end{aligned}$$

A shortest linear program for computing these linear forms has length 6. The following linear program is an optimal solution for this example.

$$\begin{aligned}
v_1 &= x_1 \oplus x_5 && [y_5] \\
v_2 &= x_2 \oplus v_1 \\
v_3 &= x_3 \oplus v_2 && [y_3] \\
v_4 &= x_4 \oplus v_3 && [y_1] \\
v_5 &= x_5 \oplus v_4 && [y_2] \\
v_6 &= v_2 \oplus v_5 && [y_4]
\end{aligned}$$

It is easy to check that for each output y_ℓ there is a variable v_i that contains the linear form for y_ℓ. In the above program, this mapping from intermediate variables to outputs is given by annotating the program lines with the associated output in square brackets.

Note that finding the shortest program over GF(2) is *not* an instance of the common subexpression elimination problem known from program optimization. The above shortest program makes extensive use of *cancellation*, i.e., of the fact that for all x in GF(2), we have $x \oplus x = 0$. For example, the output y_4 is computed by adding v_2 and v_5. These two variables are described by the linear forms $x_1 \oplus x_2 \oplus x_3 \oplus x_4$ and $x_1 \oplus x_2 \oplus x_5$, respectively. By adding these two linear forms, we obtain the desired $x_3 \oplus x_4 \oplus x_5$ since $x_1 \oplus x_1 \oplus x_2 \oplus x_2 = 0$ for all values of x_1 and x_2. Without cancellations, a shortest linear straight-line program has length 8, i.e., it uses 25% more XOR gates.

The goal that we are now pursuing in this paper is to synthesize an optimal linear straight-line program for a given set of linear forms both automatically and efficiently. Formally, this problem can be described as follows:

> Given n variables $x_1, \ldots, x_n$ over GF(2) and m linear forms $y_\ell = a_{\ell,1} \cdot x_1 \oplus \ldots \oplus a_{\ell,n} \cdot x_n$, find the shortest linear program that computes all y_ℓ.

Note that here we are aiming at a (provably) *optimal* solution. This is opposed to allowing approximations with more lines than actually necessary, which is currently the state of the art [2].

As a step towards solving this optimization problem, first let us consider the corresponding decision problem:

> Given n variables $x_1, \ldots, x_n$ over GF(2), m linear forms $y_\ell = a_{\ell,1} \cdot x_1 \oplus \ldots \oplus a_{\ell,n} \cdot x_n$ and a natural number k, decide if there exists a linear program of length k that computes all y_ℓ.

Of course, if the answer to this question is "Yes", we do not only wish to get this answer, but we would also like to obtain a corresponding program of length (at most) k. In line i, the variable v_i is defined as the sum of two other variables. Here, one may read from the variables $x_1, \ldots, x_n$ and also from the intermediate variables $v_1, \ldots, v_j$ with $j < i$, i.e., from those intermediate variables that have been defined before.

To facilitate the description of our encoding in the following section, we reformulate the problem via matrices over GF(2). Here, given a natural number k, we represent the given coefficients of the m linear forms over n inputs with $y_\ell = a_{\ell,1} \cdot x_1 \oplus a_{\ell,2} \cdot x_2 \oplus \ldots \oplus a_{\ell,n} \cdot x_n$ $(1 \leq \ell \leq m)$ as rows of an $m \times n$-matrix A. The ℓ-th row thus consists of the entries $a_{\ell,1} a_{\ell,2} \ldots a_{\ell,n}$ from GF(2).

Likewise, we can also express the resulting program via two matrices:

- A matrix $B = (b_{i,j})_{k \times n}$ over GF(2), where $b_{i,j} = 1$ iff in line i of the program the input variable x_j is read.
- A matrix $C = (c_{k,k})_{k \times k}$ over GF(2) where $c_{i,j} = 1$ iff in line i of the program the intermediate variable v_j is read.

To represent for example the program line $v_3 = x_3 \oplus v_2$ from Example 1, all $b_{3,j}$ except for $b_{3,3}$ and all $c_{3,j}$ except for $c_{3,2}$ have to be 0. Thus, the third row in B is $\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ while in C it is $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$.

Now, for the matrices B and C to actually represent a legal linear straight-line program, for any row i there must be exactly two non-zero entries in the combined i-th row of B and C. That is, the vector $\begin{pmatrix} b_{i,1} & \ldots & b_{i,n} & c_{i,1} & \ldots & c_{i,k} \end{pmatrix}$ must contain exactly two 1s.

Furthermore, for the represented program to actually compute our linear forms, we have to demand that for each desired output y_ℓ, there is a line i in the program (and the matrices) such that $v_i = y_\ell$ where $y_\ell = a_{\ell,1} \cdot x_1 \oplus \ldots \oplus a_{\ell,n} \cdot x_n$ and $v_i = b_{i,1} \cdot x_1 \oplus \ldots \oplus b_{i,n} \cdot x_n \oplus c_{i,1} \cdot v_1 \oplus \ldots \oplus c_{i,i-1} \cdot v_{i-1}$. Note that we only use the lower triangular matrix as a program may only read intermediate values that have already been written. To represent the mapping of intermediate variables to outputs, we use a function $f : \{1, \ldots, m\} \mapsto \{1, \ldots, k\}$.

Example 2. Consider again the linear forms from Example 1. They are represented by the following matrix A. Likewise, the program is represented by the matrices B and C and the function f.

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad C = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \quad f = \begin{cases} 1 \mapsto 4 \\ 2 \mapsto 5 \\ 3 \mapsto 3 \\ 4 \mapsto 6 \\ 5 \mapsto 1 \end{cases}$$

Obviously, all combined rows of B and C contain exactly two non-zero elements. Furthermore, by computing the v_i and the y_ℓ, we can see that each of the linear forms described by A is computed by the program represented by B and C.

3 Encoding to Propositional Logic

Now that the scenario has been set up and the matrix formulation has been introduced, we start by giving a high-level encoding of the decision problem as a logical formula in second order logic. Then we perform a stepwise refinement of that encoding where in each step we eliminate some elements that cannot directly be expressed by satisfiability of propositional logic.

For our first encoding, the carrier is the set of natural numbers, and we use predicates over indices to represent the matrices A, B, and C as well as the vectors x, y, and v. We also use a function f to map indices of outputs from y to indices of intermediate variables from v. Finally, we make use of cardinality constraints by predicates exactly_k that take a list of variables and check that the number of variables that are assigned 1 is exactly k.

First, we need to ensure that B and C represent a legal linear straight-line program. This is encoded by the following formula α_1:

$$\alpha_1 = \bigwedge_{1 \le i \le k} \text{exactly}_2(B(i,1), \ldots, B(i,n), C(i,1), \ldots, C(i,i-1))$$

Second, we demand that the values for the intermediate variables from v are computed by using the values from B and C:

$$\alpha_2 = \bigwedge_{1 \le i \le k} \left(v(i) \;\leftrightarrow\; \bigoplus_{1 \le j \le n} B(i,j) \wedge x(j) \;\oplus\; \bigoplus_{1 \le p < i} C(i,p) \wedge v(p) \right)$$

Third, we ensure that the value of the intermediate variable determined by f for the ℓ-th output actually takes the same value as the ℓ-th linear form:

$$\alpha_3(\ell) = v(f(\ell)) \leftrightarrow \bigoplus_{1 \le j \le n} A(\ell, j) \wedge x(j)$$

Here, $v(f(\ell))$ denotes the intermediate variable which stores the result of the linear form $y(\ell)$. In other words, the (existentially quantified) function f maps the index ℓ of the linear form y_ℓ to the index $i = f(\ell)$ of the variable v_i in v which contains the result of y_ℓ.

Now we can give our first encoding by the following formula α:

$$\alpha = \exists B. \exists C. \exists f. \forall x. \exists v.\ \alpha_1 \wedge \alpha_2 \wedge \bigwedge_{1 \le \ell \le m} \alpha_3(\ell)$$

Note that we indeed use the expressivity of second order logic as all our quantifications are over predicates and functions. Fortunately, all these only need to be defined on finite domains. In order not to have to deal with quantification over predicates representing matrices and vectors, we can just introduce a finite number of Boolean variables to represent the elements of the matrices and vectors and work on these directly. For example, for the $k \times n$ matrix B we introduce the $k \cdot n$ Boolean variables $b_{1,1} \ldots, b_{k,n}$.

Similarly, for the function f we introduce $m \cdot k$ Boolean variables $f_{\ell,i}$ that denote that the ℓ-th linear form is computed by the i-th intermediate variable. To make sure that these variables actually represent a function, we need to encode well-definedness: for each ℓ there must be exactly one i with $f_{\ell,i}$.

We obtain the refined overall constraint β, which is a formula from QBF:

$$\beta_1 = \bigwedge_{1 \le i \le k} \text{exactly}_2(b_{i,1}, \ldots, b_{i,n}, c_{i,1}, \ldots, c_{i,i-1})$$

$$\beta_2 = \bigwedge_{1 \le i \le k} \left(v_i \leftrightarrow \bigoplus_{1 \le j \le n} b_{i,j} \wedge x_j \oplus \bigoplus_{1 \le p < i} c_{i,p} \wedge v_p \right)$$

$$\beta_3(\ell) = \bigwedge_{1 \le i \le k} \left(f_{\ell,i} \rightarrow \left(v_i \leftrightarrow \bigoplus_{1 \le j \le n} a_{\ell,j} \wedge x_j \right) \right) \ \wedge \ \text{exactly}_1(f_{\ell,1}, \ldots, f_{\ell,k})$$

$$\beta = \exists b_{1,1} \ldots \exists b_{k,n} . \exists c_{1,1} \ldots \exists c_{k,k} . \exists f_{1,1} \ldots \exists f_{m,k} . \forall x_1 \ldots \forall x_n . \exists v_1 \ldots \exists v_k . \\ \beta_1 \ \wedge \ \beta_2 \ \wedge \bigwedge_{1 \le \ell \le m} \beta_3(\ell)$$

The above formula β is in prenex normal form and has a quantifier prefix of the shape $\exists^+ \forall^+ \exists^+$. This precludes us from using a SAT solver on β directly. For this, we would need to have a quantifier prefix of the shape $\exists^+$ alone. Thus, unless we want to use a QBF solver, we need to get rid of the $\forall^+ \exists^+$ suffix of the quantifier prefix of β. In other words, we need to get rid of the quantifications over $x_1, \ldots, x_n$ and $v_1, \ldots, v_k$.

We observe that β explicitly contains the computed values v_i of the intermediate variables. We can eliminate them by unrolling the defining equations of an intermediate variable v_i to be expressed directly via $x_1, \ldots, x_n$. In other words, we do not regard the intermediate variables for "computing" the result of the linear forms y_ℓ, but we directly use a closed expression that depends on the $b_{i,j}$ and the $c_{i,p}$. Here, we introduce the auxiliary formulae $\varphi(i)$ for $1 \le i \le k$ whose truth value should correspond to the value taken by the corresponding v_i:

$$\varphi(i) = \left(\bigoplus_{1 \le j \le n} b_{i,j} \wedge x_j \right) \oplus \left(\bigoplus_{1 \le p < i} c_{i,p} \wedge \varphi(p) \right)$$

We now reformulate β to obtain a refined encoding γ. Note that we do not need to redefine β_1 and we do not need an equivalent of β_2 as we unroll the definition of the v_i into γ_3 using $\varphi(i)$.

$$\gamma_3(\ell) = \bigwedge_{1 \le i \le k} \left(f_{\ell,i} \rightarrow \left(\varphi(i) \leftrightarrow \bigoplus_{1 \le j \le n} a_{\ell,j} \wedge x_j \right) \right) \ \wedge \ \text{exactly}_1(f_{\ell,1}, \ldots, f_{\ell,k})$$

$$\gamma = \exists b_{1,1} \ldots \exists b_{k,n} . \exists c_{1,1} \ldots \exists c_{k,k} . \exists f_{1,1} \ldots \exists f_{m,k} . \forall x_1 \ldots \forall x_n . \beta_1 \wedge \bigwedge_{1 \le \ell \le m} \gamma_3(\ell)$$

Note that it looks as though for each i we had obtained many redundant copies of the subformulae $\varphi(i)$, which would entail a blow-up in formula size. However, in

practical implementations it is beneficial to represent propositional formulae not as trees, but as directed acyclic graphs with sharing of common subformulae. This technique is also known as *structural hashing* [6]. We perform standard Boolean simplifications (e.g., $\varphi \wedge 1 = \varphi$), we share Boolean junctor applications modulo commutativity and idempotence (where applicable), and we use varyadic $\wedge$ and $\vee$. In contrast, the junctors $\leftrightarrow$ and $\oplus$ are binary and associate to the left.

Nevertheless, we still have universal quantification over the inputs as part of our encoding. This states that regardless of the input values for $x_1, \ldots, x_n$, our program should yield the correct result. Fortunately, we can now benefit from linearity of the operation $\oplus$ on GF(2), which means that the absolute positiveness criterion for polynomials [10] (a simple technique commonly used in automated termination provers, cf. e.g. [7]) is not only sound, but also complete. Essentially, the idea is that two linear forms compute the same function iff their coefficients are identical. In this way, we can now drop the inputs $x_1, \ldots, x_n$.

For $1 \leq j \leq n$ and $1 \leq i \leq k$, we introduce the auxiliary formulae $\psi(j, i)$, which should denote the dependence of the value for v_i with respect to x_j (i.e., whether the value of v_i toggles if x_j changes or not):

$$\psi(j,i) = b_{i,j} \oplus \bigoplus_{1 \leq p < i} c_{i,p} \wedge \psi(j,p)$$

We finally get an encoding δ in prenex normal form that can be used as input for a SAT solver (by dropping explicit existential quantification, encoding cardinality constraints using [5, 1], and performing Tseitin's transformation [14]).

$$\delta_3(\ell) = \bigwedge_{1 \leq i \leq k} \left(f_{\ell,i} \rightarrow \bigwedge_{1 \leq j \leq n} (\psi(j,i) \leftrightarrow a_{\ell,j}) \right) \wedge \text{ exactly}_1(f_{\ell,1}, \ldots, f_{\ell,k})$$

$$\delta = \exists b_{1,1} \ldots \exists b_{k,n} . \exists c_{1,1} \ldots \ldots \exists c_{k,k} . \exists f_{1,1} \ldots \exists f_{m,k} .\ \beta_1 \wedge \bigwedge_{1 \leq \ell \leq m} \delta_3(\ell)$$

For the implementation of δ we used the SAT framework in the verification environment AProVE [8] and the Tseitin implementation from SAT4J [12].

3.1 Size of the Encoding

Given a decision problem with an $m \times n$ matrix and a natural number k (where w.l.o.g. $m \leq k$ holds since for $m > k$, we could just set $\delta = 0$), our encoding δ has size $\mathcal{O}(n \cdot k^2)$ if the cardinality constraints are encoded in space linear in the number of arguments [5]. To see this, consider the following size estimation for δ where due to the use of structural hashing we can look at δ_3 and ψ separately.

$$|\delta| = \mathcal{O}(k \cdot n + k \cdot k + m \cdot k + |\beta_1| + m \cdot |\delta_3| + n \cdot k \cdot |\psi|)$$

For β_1 and δ_3 we obtain the following estimations where g is a function describing the size of the cardinality constraint:

$$|\beta_1| = \mathcal{O}(k \cdot g(n+k)) \qquad |\delta_3| = \mathcal{O}(k \cdot n + g(k))$$

For ψ we immediately obtain the size estimation $|\psi| = \mathcal{O}(k)$. Now, we can simplify the estimation for δ by using $m \leq k$:

$$\begin{aligned}|\delta| &= \mathcal{O}(k \cdot n + k \cdot k + m \cdot k + k \cdot g(n+k) + m \cdot (k \cdot n + g(k)) + n \cdot k \cdot k) \\ &= \mathcal{O}(n \cdot k^2 + k \cdot g(n+k) + m \cdot g(k))\end{aligned}$$

3.2 Tuning the Encoding

The models of the encoding δ from this section are all linear straight-line programs of length k that compute the m linear forms $y_1, \ldots, y_m$. The programs can be decoded from a satisfying assignment of the propositional formula by simply reconstructing the matrices B and C.

In this paper, we are interested in finding short programs. Thus, we can exclude many programs that perform redundant computations. We do so by adding further conjuncts that exclude those undesired programs. While we change the set of models, note that we do not change the satisfiability of the decision problem. That is, if there is a program that computes the given linear forms in k steps, we will find one which does not perform these kinds of redundant computation.

The first kind of redundant programs are programs that compute the same linear form twice, i.e., there are two different intermediate variables that contain the same linear form. We exclude such programs by demanding that for all distinct pairs of intermediate variables v_i and v_p, there is also some x_j that influences exactly one of the two variables:

$$\bigwedge_{1 \leq i \leq k} \bigwedge_{1 \leq p < i} \bigvee_{1 \leq j \leq n} (\psi(j,p) \oplus \psi(j,i))$$

The second kind of redundant programs are programs that compute the constant 0 or a linear form just depending on the value of one input variable. To exclude such programs, we add cardinality constraints stating that each compute linear form must depend on at least two input variables.

$$\bigwedge_{1 \leq i \leq k} \text{atLeast}_2(\psi(1,i), \ldots, \psi(n,i))$$

In fact, statements that compute linear forms that only depend on two input variables can be restricted not to use any other intermediate variables (as they could be computed in one step from the inputs).

$$\bigvee_{1 \leq j < i} c_{i,j} \rightarrow \bigwedge_{1 \leq i \leq k} \text{atLeast}_3(\psi(1,i), \ldots, \psi(n,i))$$

Apart from disallowing redundant programs, we additionally include implied conjuncts to further constrain the search space. In this way, the SAT solver becomes more efficient as unit propagation can be employed in more situations.

As stated in Section 2, we require that the input does not contain duplicate linear forms. Consequently, we may require f to be injective, i.e., any intermediate variable covers at most one linear form.

$$\bigwedge_{1 \leq i \leq k} \text{atMost}_1(f_{1,i}, \ldots, f_{m,i})$$

Often, CDCL-based SAT solvers are not very good at solving the pigeonhole problem. Additional constraints facilitate better unit propagation in these cases. Since f maps from $\{1, \ldots, m\}$ to $\{1, \ldots, k\}$, only at most k of the $f_{\ell,i}$ may become true.

$$\text{atMost}_k(f_{1,1}, \ldots, f_{m,k})$$

Similarly, we can even state that at least m of the $f_{\ell,i}$ need to become true as we have to compute all given (distinct) m linear forms.

$$\text{atLeast}_m(f_{1,1}, \ldots, f_{m,k})$$

4 From Decision Problem to Optimization

A simple approach for solving an optimization problem given a decision procedure for the associated decision problem is to search for the parameter to be optimized by repeatedly calling the decision procedure.

In our case, for minimizing the length k of the synthesized linear straight-line program, we start by observing that this minimal length must be at least the number of linear forms. At the same time, if we compute each linear form separately, we obtain an upper bound for the minimal length. More precisely, we know that the minimal length k_{min} is in the closed interval from m to $|A|_1 - m$ where $|\cdot|_1$ denotes the number of 1s in a matrix.

Without further heuristic knowledge about the typical length of shortest programs, the obvious thing to do is to use a bisecting approach for refining the interval. That is, one selects the middle element of the current interval and calls a decision procedure based on our encoding from Section 3 for this parameter. If there is a model, the interval is restricted to the lower half of the previous interval and we continue bisecting. If there is no model and δ is unsatisfiable, the interval is restricted to the upper half of the previous interval. When the interval becomes empty, the lower limit indicates the minimal parameter k_{min}.

The above approach requires a logarithmic number of calls to the decision procedure, approximately half of which will return the result "unsatisfiable". This approach is very efficient if we can assume that our decision procedure takes approximately the same time for a positive answer as for a negative answer. As we will see in the case study of the following section, though, for realistic problem instances the negative answers may require orders of magnitude more time.

Thus, to minimize the number of calls to the decision procedure resulting in a negative answer, we propose the following algorithm for refining the length k.

1. Start with $k := |A|_1 - m - 1$.
2. Call the decision procedure with k.
3. If `UNSAT`, return $k + 1$ and exit.
4. If `SAT`, compute used length k_{used} from B and C.
5. Set $k := k_{used} - 1$ and go to Step 2.

Here, the *used length* of a program is the number of variables that are needed directly or indirectly to compute the m linear forms. For given matrices B and C and a function f, the set of used variables *used* is the least set such that:

- if $f(\ell) = i$, then $v_i \in \mathit{used}$ and
- if $v_i \in \mathit{used}$ and $c_{i,j} = 1$, then $v_j \in \mathit{used}$.

The used length can then be obtained as the cardinality of the set *used*.

This algorithm obviously only results in exactly one call to UNSAT – directly before finding the minimal solution. The price we pay for this is that in the worst case we have to call the decision procedure a linear number of times. In practice, though, for $k > k_{min}$, there are many solutions and the solution returned by the SAT solver typically has $k_{used} < k$. Consequently, at the beginning the algorithm typically approaches k_{min} in rather large steps.

While it seems natural to use MaxSAT for our optimization problem instead of calling the SAT solver repeatedly, the decision problems close to the optimum are already so hard that solving these as part of a larger instance seems infeasible.

5 Case Study: Advanced Encryption Standard

As mentioned in the introduction, a major motivation for our work is the minimization of circuits for implementing cryptographic algorithms. In this section, we study how our contributions can be applied to optimize an important component of the Advanced Encryption Standard (AES) [13].

The AES algorithm consists of the (repeated) application of four steps. The main step for introducing non-linearity is the SubBytes step that is based on a so-called S-box. This S-box is a transformation based on multiplicative inverses in GF(2^8) combined with an invertible affine transformation. This step can be decomposed into two linear parts and a minimal non-linear part.

For our case study, we consider the first of the linear parts (called the "top matrix" in [4]) which is represented by the following 21×8 matrix A:

$$A = \begin{pmatrix}
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{pmatrix}
\quad
B = \begin{pmatrix}
0&0&1&0&0&0&0&1\\
1&0&0&0&0&1&0&0\\
0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0\\
0&1&0&0&0&0&0&0\\
1&0&0&0&0&0&1&0\\
1&0&0&1&0&0&0&0\\
0&0&0&0&0&0&1&0\\
0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&1&0&1&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0
\end{pmatrix}
\quad
C = \begin{pmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0
\end{pmatrix}$$

Here, the matrices B and C represent a solution with length $k = 23$. This solution was found in less than one minute using our decision procedure from

Section 3 with MiniSAT v2.1 as backend on a 2.67 GHz Intel Core i7. We strongly conjecture that $k_{\min} = 23$ and, indeed, the shortest known linear straight-line program for the linear forms described by the matrix A has length $k = 23$ [4]. This shows that our SAT-based optimization method is able to find very good solutions in reasonable time. The UNSAT case is harder, though. For $k = 20$ (which is trivially unsatisfiable due to the pigeonhole problem), without the tunings from Section 3 we cannot show unsatisfiability within 4 days. But with the tunings enabled we can show unsatisfiability in less than one second.

Unfortunately, proving the unsatisfiability for $k = 22$ proves to be much more challenging. Indeed, we have run many different SAT solvers (including but not limited to glucose, ManySat, MiniSat, MiraXT with 8 threads, OKsolver, PicoSAT, PrecoSAT, RSat, SAT4J) on the CNF file for this instance of our decision problem. Some of the more promising solvers for this instance were run for more than 40 days without returning either SAT or UNSAT.

In an effort to prove unsatisfiability of this instance and thereby prove optimality of the solution with $k = 23$, we have asked for and received a lot of support and good advice from the SAT community (see the Acknowledgements at the end of this paper). Still, to this day the unsatisfiability of this instance remains a conjecture. Using pre-processing techniques, the number of variables of this instance can be reduced from more than 45000 to less than 5000 in a matter of minutes. The remaining SAT problem seems to be very hard, though. [3]

To analyze how difficult these problems really are, we consider a small subset of the linear forms to be computed for the top matrix. The table to the right shows how the runtimes in seconds of the SAT solver are affected by the choice of k for the case that we consider only the first 8 out of 21 linear forms from A. In order to keep runtimes manageable we already incorporated the symmetry breaking improvement described in Section 6. Note that unsatisfiability for $k = 12$ is still much harder to show than satisfiability for $k_{min} = 13$.

To conclude this case study, we see that while finding (potentially) minimal solutions is obviously feasible, proving their optimality (i.e., unsatisfiability of the associated decision problem for $k = k_{min}-1$) is challenging. This observation confirms observations made in [11]. In the following section we present some of our attempts to improve the efficiency of our encoding for the UNSAT case.

k	result	time
8	UNSAT	0.4
9	UNSAT	0.5
10	UNSAT	1.2
11	UNSAT	5.0
12	UNSAT	76.8
13	SAT	1.0
14	SAT	3.4
15	SAT	2.8
16	SAT	1.5
17	SAT	4.3
18	SAT	2.7
19	SAT	2.5
20	SAT	3.0
21	SAT	3.0
22	SAT	3.5
23	SAT	3.6
24	SAT	5.5
25	SAT	5.9

6 Towards Handling the UNSAT Case

Satisfiability of propositional logic is an NP-complete problem and, thus, we can expect that at least some instances are computationally expensive. While

[3] The reader is cordially invited to try his favorite SAT solver on one of the instances available from: `http://aprove.informatik.rwth-aachen.de/eval/slp.zip`

SAT solvers have proven to be a Swiss army knife for solving practically relevant instances of many different NP-complete problems, our kind of program synthesis problems seems to be a major challenge for today's SAT solvers even on instances with "just" 1500 variables.

In this section we discuss three different approaches based on unary SAT encodings, on Pseudo-Boolean satisfiability, and on symmetry breaking.

6.1 Unary encodings

As remarked by [9], encoding arithmetic in unary representation instead of the more common binary (CPU-like) representation can be very beneficial for the performance of modern conflict-driven SAT solvers on the resulting instances. Unfortunately, encoding the computations not via XOR on GF(2), but rather in unary representation on $\mathbb{Z}$ with a deferred parity check turned out to be prohibitively expensive as the (integer) values for the i-th line are bounded only by $\mathcal{O}(\mathit{fib}(i))$ where fib is the Fibonacci function.

6.2 Encoding to Pseudo-Boolean Constraints

Instead of optimizing and tuning our encoding to SAT, we also implemented a straight-forward encoding to Pseudo-Boolean constraints. The hope was that, e.g., cutting plane approaches could be useful for showing unsatisfiability.

We experimented with MiniSat+, Pueblo, SAT4J, and SCIP but were not able to obtain any improvements for e.g. the first 8 linear forms of the top matrix.

6.3 Symmetry Breaking

In general, having many solutions is considered good for SAT instances as the SAT solver is more likely to "stumble" upon one of them. For UNSAT instances, though, having many potential solutions usually means that the search space to exhaust is very large.

One of the main reasons for having many solutions is symmetry. For example, it does not matter if we first compute $v_1 = x_1 \oplus x_2$ and then $v_2 = x_3 \oplus x_4$ or the other way around. Limiting these kinds of symmetries can be expected to significantly reduce the runtimes for UNSAT instances.

In our concrete setting, being able to reorder independent program lines is one of the major sources of symmetry. Two outputs in a straight-line programs are said to be *independent* if neither of them depends on the other (directly through the matrix C or indirectly).

Now, the idea for breaking symmetry is to impose an order on these lines: the line which computes the "smaller" linear form (w.r.t. a total order on linear forms, which can e.g. be obtained by lexicographic comparison of the coefficient vectors) must occur before the line which computes the greater linear form.

We can encode the direct dependence of v_i on v_p:

$$\bigwedge_{1 \le i \le k} \bigwedge_{1 \le p < i} c(i,p) \rightarrow \mathit{dep}(i,p)$$

Likewise, the indirect dependence of v_i on v_p can be encoded by transitivity:

$$\bigwedge_{1\le i\le k}\bigwedge_{1\le p<i}\bigwedge_{p<q<i} c(i,q)\wedge dep(q,p)\rightarrow dep(i,p)$$

We also need to encode the reverse direction, i.e.:

$$\bigwedge_{1\le i\le k}\bigwedge_{1\le p<i}\left(dep(i,p)\rightarrow\left(c(i,p)\ \vee\bigvee_{p<q<i}(c(i,q)\wedge dep(q,p))\right)\right)$$

Now we can enforce that for $i > p$, the output v_i depends on the output v_p or v_i encodes a greater linear form than v_p:

$$\bigwedge_{1\le i\le k}\bigwedge_{1\le p<i}(dep(i,p)\ \vee\ [\psi(1,i),\ldots,\psi(n,i)]>_{lex}[\psi(1,p),\ldots,\psi(n,p)])$$

Here lexicographic comparison of formula tuples is encoded in the usual way (see for example the encodings in [7, 5]).

While this approach eliminates some otherwise valid solutions of length k and thus reduces the set of admissible solutions, obviously there is at least one solution of length k which satisfies our constraints whenever solutions of length k exist at all. This way, we greatly reduce the search space by breaking symmetries that are not relevant for the result, but may slow down the search considerably.

Consider again the restriction of our S-box top matrix to the first 8 linear forms. With symmetry breaking, we can show unsatisfiability for the "hard" case $k = 12$ in 76.8 seconds. In contrast, without symmetry breaking, we cannot show unsatisfiability within 4 days.

7 Conclusion

In this paper we have shown how shortest linear straight-line programs for given linear forms can be synthesized using SAT solvers. To this end we have presented a novel polynomial-size encoding of the associated decision problem to SAT and a customized white-box method for again turning this decision procedure into an optimization algorithm.

We have evaluated the feasibility of this approach by a case study where we minimize an important part of the S-box for the Advanced Encryption Standard. This study shows that our SAT-based approach is indeed able to synthesize shortest-known programs for realistic problem settings within reasonable time.

Proving the optimality of the programs found by showing unsatisfiability of the associated decision problem leads to very challenging SAT problems. To improve the performance for the UNSAT case, we discussed three approaches based on unary encodings, on a port to Pseudo-Boolean satisfiability, and on symmetry breaking. We have shown that symmetry breaking significantly reduces runtimes in the UNSAT case.

In future work, we consider to apply our method to other problems from cryptography. Also, we plan to further enhance our encoding and specialize existing SAT solvers to further improve performance in the UNSAT case.

Acknowledgements

Our sincere thanks go to Erika Ábrahám, Daniel Le Berre, Armin Biere, Youssef Hamadi, Oliver Kullmann, Matthew Lewis, Lakhdar Saïs, and Laurent Simon for input on and help with the experiments. Furthermore, we thank Joan Boyar and René Peralta for providing us with information on their work and Michael Codish for pointing out similarities to common subexpression elimination.

References

1. R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In *Proc. Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *LNCS*, pages 167–180, 2009.
2. J. Boyar, P. Matthews, and R. Peralta. On the shortest linear straight-line program for computing linear forms. In *Proc. Mathematical Foundations of Computer Science (MFCS '08)*, volume 5162 of *LNCS*, pages 168–179, 2008.
3. J. Boyar and R. Peralta. A new technique for combinational circuit optimization and a new circuit for the S-Box for AES. Patent Application Number 61089998 filed with the U.S. Patent and Trademark Office, 2009.
4. J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In *Proc. International Symposium on Experimental Algorithms (SEA '10)*, volume 6049 of *LNCS*, 2010, To appear.
5. M. Codish, V. Lagoon, and P. Stuckey. Solving partial order constraints for LPO termination. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 5:193–215, 2008.
6. N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modelling and Computation (JSAT)*, 2(1–4):1–26, 2006.
7. C. Fuhs, J. Giesl, A. Middeldorp, R. Thiemann, P. Schneider-Kamp, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. Theory and Applications of Satisfiability Testing (SAT '07)*, volume 4501 of *LNCS*, pages 340–354, 2007.
8. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *LNAI*, pages 281–286, 2006.
9. O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *Proc. International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *LNAI*, pages 483–497, 2006.
10. H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning (JAR)*, 21(1):23–38, 1998.
11. A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev. Finding efficient circuits using SAT-solvers. In *Proc. Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *LNCS*, pages 32–44, 2009.
12. D. Le Berre and A. Parrain. SAT4J. `http://www.sat4j.org`.
13. Federal Information Processing Standard 197. The advanced encryption standard. Technical report, National Institute of Standards and Technology, 2001.
14. G. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. 1968. Reprinted in *Automation of Reasoning*, 2:466–483, 1983.