Determining the Multiplicative Complexity of Boolean Functions using SAT

Mathias Soeken
Microsoft, Switzerland

***Abstract*—We present a constructive SAT-based algorithm to determine the multiplicative complexity of a Boolean function, i.e., the smallest number of AND gates in any logic network that consists of 2-input AND gates, 2-input XOR gates, and inverters. In order to speed-up solving time, we make use of several symmetry breaking constraints; these exploit properties of XAGs that may be useful beyond the proposed SAT-based algorithm. We further propose a heuristic post-optimization algorithm to reduce the number of XOR gates once the optimum number of AND gates has been obtained, which also makes use of SAT solvers. Our algorithm is capable to find all optimum XAGs for representatives of all 5-input affine-equivalent classes, and for a set of frequently occurring 6-input functions.**

I. Introduction

We are considering the minimization of AND gates in XOR-AND graphs (XAGs), which are logic networks that can have 2-input AND gates and 2-input XOR gates. XAGs can represent all normal Boolean functions, also called 0-preserving functions, which are all Boolean functions f for which $f(0,\ldots,0) = 0$. A non-normal Boolean function $f(x)$ can be represented by an XAG for $\bar{f}(x)$ with its output being inverted. Note that the use of inverters cannot lead to a smaller number of AND gates in the logic network [1]. Inverters can always be propagated towards the outputs, since

$$\bar{x} \oplus \bar{y} = x \oplus y,\ \bar{x} \oplus y = \overline{x \oplus y},$$
$$\bar{x}y = xy \oplus y, \text{ and } \bar{x}\bar{y} = \overline{xy \oplus x \oplus y}. \quad (1)$$

Missing cases are covered by commutativity of the operations. In the remainder of this paper, we assume that all Boolean functions are normal, unless explicitly stated otherwise.

The multiplicative complexity of a Boolean function is the smallest number of AND gates in any XAG that represents the function [1]. Determining the multiplicative complexity is intractable. Find has shown that if one-way functions exist [2], no algorithm can compute the multiplicative complexity of an n-variable Boolean function in time $2^{O(n)}$ given as input the truth table of f [3]. Therefore, heuristic optimization techniques that minimize the number of AND gates in XAGs have been used as a tool to assess the multiplicative complexity of the function [4], [5], [6], [7], [8], since it provides an upper bound of the actual complexity.

The concept of multiplicative complexity and the optimization of AND gates in XAGs play an important role in cryptography and fault-tolerant quantum computing. In cryptography, the number of AND gates correlates to the degree of vulnerability of a circuit [9]. Further, the multiplicative complexity of a function directly correlates to the resistance of the function against algebraic attacks [10]. The number of AND gates also plays an important role in high-level cryptography protocols such as zero-knowledge protocols, fully homomorphic encryption (FHE), and secure multi-party computation (MPC) [11], [12], [6]. For example, the size of the signature in post-quantum zero-knowledge signatures based on "MPC-in-the-head" [13] depends on the multiplicative complexity in the underlying block cipher [12]. Moreover, the number of computations in MPC protocols based on Yao's garbled circuits [14] with the free XOR technique [15] is proportional to the number of AND gates. Regarding FHE, XOR gates are considered cheaper and less noisy compared to AND gates. In fault-tolerant quantum computing, the multiplicative complexity directly corresponds to the number of qubits and expensive operations (T gates), and therefore optimizing AND gates in an XAG can lead to more efficient quantum circuits [16].

In this paper, we present a SAT-based algorithm to determine the multiplicative complexity of a Boolean function. The algorithm is constructive and returns an XAG, in which the number of AND gates is minimum. The algorithm can only be applied to either small functions or to functions which have a small multiplicative complexity. It is inspired by SAT-based exact logic synthesis techniques [17], [18], [19]. While XAGs with minimum number of AND gates are known for all 6-input functions [20], our algorithms can also be used to enumerate multiple structurally different solutions, in order to minimize the number of XOR gates, while keeping the number of AND gates unchanged. For this purpose, we exploit an existing SAT-based algorithm to minimize the number of XOR gates in logic networks for linear functions [21]. Our algorithm is capable to find all optimum XAGs for representatives of all 5-input affine-equivalent classes, and for a set of frequently occurring 6-input functions. We further used the algorithm to find a new collection of optimum XAGs for 5-input functions, categorized with respect to representatives of a recently proposed affine equivalence classification algorithm [22].

II. Preliminaries

As general notation, we are using $[n] = \{1,\ldots,n\}$. We model an XAG for a Boolean function over variables $x_1,\ldots,x_n$ as a sequence of r steps, where each step has one of the two following forms:

$$x_i = x_{j_{1i}} \oplus x_{j_{2i}} \quad \text{or} \quad x_i = x_{j_{1i}} \wedge x_{j_{2i}} \quad (2)$$

for $n < i \leq n + r$. The values $1 \leq j_{1i} < j_{2i} < i$ point to primary inputs or previous steps in the network. The function

value is computed by the last step $f = x_{n+r}$. We assume that all logic functions represent Boolean functions that depend on all primary input variables. To represent the constant function, we define $x_0 = 0$.

Example 1: The *if-then-else* function $x_1 \,?\, x_2 : x_3 = x_1x_2 \oplus \bar{x}_1x_3 = x_1x_2 \oplus x_1x_3 \oplus x_3$ can be computed by the 4-step XAG

$$x_4 = x_1 \wedge x_2,\ x_5 = x_1 \wedge x_3,\ x_6 = x_4 \oplus x_5,\ x_7 = x_3 \oplus x_6.$$

The *multiplicative complexity of an XAG* is the number of AND gates it contains. It is an upper bound to the multiplicative complexity of the Boolean function, which is the smallest number of AND gates in any XAG that realizes the function.

Example 2: The multiplicative complexity of the XAG to realize the if-then-else function in Example 1 is 2, however, the function has multiplicative complexity 1, witnessed by the following alternative XAG:

$$x_4 = x_2 \oplus x_3,\ x_5 = x_1 \wedge x_4,\ x_6 = x_3 \oplus x_5$$

Let $S = \{i_1, \ldots, i_k\} \subseteq [n]$, then

$$L_S(x_1, \ldots, x_n) = x_{i_1} \oplus \cdots \oplus x_{i_k} \tag{3}$$

is a linear function over the variables indexed by S. We define $L_\emptyset = 0$, and omit brackets when explicitly writing the indexes, e.g., we write $L_{1,3}(x_1, x_2, x_3)$ instead of $L_{\{1,3\}}(x_1, x_2, x_3)$.

Since we are only interested in the number of AND gates in the XAG, we can describe the networks in a more general way such that the number of steps equals the number of AND gates. Each fanin of an AND gate is a multi-input XOR gate whose fanins in turn are either primary inputs or AND gates defined in previous steps. These multi-input XOR gates may only have a single input, in case the fanin of an AND gate directly connects to a primary input or another AND gate. Therefore, for a function $f(x_1, \ldots, x_n)$ a more abstract logic network, which we call abstract XAG in the following, consists of steps

$$x_i = L_{S_{1i}} \wedge L_{S_{2i}} \tag{4}$$

for $n < i \leq n+r$ with $S_{1i}, S_{2i} \subseteq [i-1]$. The function value is computed as a linear function over all primary inputs and AND gates $f = L_S$, with $S \subseteq [n+r]$. This abstract logic network has been defined in a similar way by the authors in [20]. In order to simplify some notation, we combine all index sets for linear functions by sets $\hat{S}_l$, where $\hat{S}_1 = S_{1(n+1)}, \hat{S}_2 = S_{2(n+1)}, \hat{S}_3 = S_{1(n+2)}, \hat{S}_4 = S_{2(n+2)}, \ldots, \hat{S}_{2r+1} = S$.

Example 3: The two XAGs for the if-then-else function from the previous examples can be represented by the abstract XAGs

$$x_4 = L_1 \wedge L_2,\ x_5 = L_1 \wedge L_3,\ f = L_{3,4,5},$$

and

$$x_4 = L_1 \wedge L_{2,3},\ f = L_{3,5},$$

respectively.

It is straightforward to translate an XAG into an abstract XAG by merging all the XOR gates between AND gates into single linear functions. The inverse direction, however, is more complicated. A naïve translation of an abstract XAG into an XAG may lead to a large number of XOR gates, if gate sharing and cancellations are not taken into account.

III. Finding Optimum Abstract XAGs

We solve the optimization problem of finding the multiplicative complexity of a normal Boolean function f, by solving a series of decision problems that ask whether there exists an abstract XAG for f with r gates. In that series of decision problems, we can either increment r, starting from some lower bound, or decrementing r, starting from an upper bound. A good lower bound is $d-1$, where d is the algebraic degree of f, which is the size of the largest monomial in its algebraic normal form [1]. A good upper bound is the number of AND-like gates in a logic network for f over binary gates.

A. SAT Encoding

We encode the decision problem as a SAT problem over variables s_{cij} for $c \in \{1, 2\}$, $n < i \leq n+r$ and $1 \leq j < i$ that are true, whenever $j \in S_{ci}$, and similarly, variables s_j that are true, whenever $j \in S$ for all $1 \leq j \leq n+r$. Further variables f_{ix} for $n < i \leq n+r$ and $0 < x < 2^n$ encode the function computed by AND gate i for input assignment $b_1, \ldots, b_n$ when $x = (b_n \ldots b_1)_2$. It is not necessary to consider the case $x = 0$, since f is normal. As notation we also introduce variables $\hat{s}_{lj}$ which correspond to variables s_{cij} and s_j with respect to the definition of the sets $\hat{S}_l$ above. Note that these are not additional variables for the SAT problem.

Clauses are added to relate the variables that encode the structure of the abstract XAG with the variables that encode the function of the abstract XAG. For each $x = (b_n \ldots b_1)_2$, we add the gate clauses

$$f_{ix} \leftrightarrow \bigwedge_{c \in \{1,2\}} \bigoplus_{j=1}^{i-1} (s_{cij} \wedge f_{jx}), \tag{5}$$

for all $n < i \leq n+r$. Here, $f_{jx} = b_j$ whenever $j \leq n$. We also add output clauses

$$f(b_1, \ldots, b_n) \leftrightarrow \bigoplus_{j=1}^{n+r} (s_j \wedge f_{jx}). \tag{6}$$

Note that $f(b_1, \ldots, b_n)$ evaluates to a constant value. All clauses are added to the SAT solver by making use of the Tseytin encoding [23].

B. Additional constraints and symmetry breaking

The clauses in the previous section are sufficient to find abstract XAGs with a maximum number of AND gates r for some given function f. In this section, we introduce additional constraints that help the SAT solver, e.g., by using redundant clauses, or by filtering out some solutions using symmetry breaking.

Non-constant linear fan-in: Unless f is the constant-0 function, all linear functions involved in the abstract XAG should contain at least one variable, i.e., $\hat{S}_l \neq \emptyset$ for all l, since they would otherwise evaluate to the 0 function. We can help the SAT solver by adding the clauses

$$(\hat{s}_{l1} \vee \cdots \vee \hat{s}_{l|S_l|}) \tag{7}$$

for all $1 \leq l \leq 2r+1$.

Commutativity: Since the AND operation is commutative, we can interchange S_{1i} and S_{2i} in any step without changing the function. We can force the SAT solver to break this symmetry by asserting that S_{1i} must be lexicographically smaller than S_{2i}, i.e., $\max\{S_{1i} \triangle S_{2i}\} \in S_{2i}$, where '$\triangle$' denotes the symmetric difference. This constraint can be enforced by adding the clauses

$$\bigwedge_{j=1}^{i-2} \big((\bar{s}_{1ij} \vee s_{2ij} \vee \bar{a}_{i(j-1)})(\bar{s}_{1ij} \vee a_{ij} \vee \bar{a}_{i(j-1)})(s_{2ij} \vee a_{ij} \vee \bar{a}_{i(j-1)})\big)$$
$$(\bar{s}_{1i(i-1)} \vee \bar{a}_{i(i-2)})(s_{2i(i-1)} \vee \bar{a}_{i(i-2)}),$$

for $n < i \leq n+r$, and $i-2$ auxiliary variables a_{ij}, where all literals $\bar{a}_{i0}$ are omitted [18].

Symmetric variables: A function $f(x_1, \ldots, x_n)$ is symmetric in two variables x_j and x_k, if swapping x_j with x_k in f does not change the function. In that case, we can enforce that x_j "is used before" x_k, by adding the clauses

$$\hat{s}_{lk} \rightarrow \bigvee_{1 \leq l' \leq l} \hat{s}_{l'j}$$

for $1 \leq l \leq 2r+1$.

All gates and all essential variables must be used: A variable x_i is essential in f if $f_{\bar{x}_i} \neq f_{x_i}$, where $f_{\bar{x}_i}$ and f_{x_i} are the negative and positive co-factors of f, which are obtained by setting x_i to 0 and 1, respectively. In an optimum solution each essential variable and each AND gate must be present in at least one of the index sets $\hat{S}_l$. This can be enforced by the clauses

$$\bigvee_{l \text{ s.t. } |\hat{S}_l| \geq j} \hat{s}_{li} \tag{8}$$

for all $i \in [n]$ such that x_i is an essential variable, and for all $n < i \leq n+r$.

Linear fan-ins are no subsets: It can be shown that there always exists a minimum abstract XAG in which $S_{1i} \not\subseteq S_{2i}$ and $S_{2i} \not\subseteq S_{1i}$ for all steps i. This property has also been used by the authors in [20] to reduce the number of topologies in the enumeration of minimum abstract XAGs for all 6-variable Boolean functions. The proof makes use of the following lemma:

Lemma 1: Let $S_1 \subseteq S_2$, then

$$L_{S_1} \wedge L_{S_2} = L_{S_1} \oplus (L_{S_1} \wedge L_{S_2 \setminus S_1}) \tag{9}$$

Proof: We start by expanding the left-hand side of the equality using the distributivity law:

$$L_{S_1} \wedge L_{S_2} = \bigoplus_{(i,j) \in S_1 \times S_2} x_i x_j$$

Splitting S_2 into the disjoint union $S_1 \cup (S_2 \setminus S_1)$ leads to

$$\bigoplus_{(i,j) \in S_1 \times S_1} x_i x_j \oplus \bigoplus_{(i,j) \in S_1 \times (S_2 \setminus S_1)} x_i x_j.$$

Since for each pair $(i,j) \in S_1 \times S_1$ with $i \neq j$, there also exists $(j,i) \in S_1 \times S_1$, all monomials of degree 2 on the left-hand side cancel, simplifying the expression to

$$\bigoplus_{i \in S_1} x_i \oplus \bigoplus_{(i,j) \in S_1 \times (S_2 \setminus S_1)} x_i x_j.$$

Now, it is easy to see that the term on the left-hand side is the linear function L_{S_1}, and the term on the right-hand side equals $L_{S_1} \wedge L_{S_2 \setminus S_1}$ by laws of distributivity. ∎

We can now formulate and prove the following theorem.

Theorem 1: Any abstract XAG can be expressed using steps in which $S_{1i} \not\subseteq S_{2i}$ and $S_{2i} \not\subseteq S_{1i}$ without changing the number of steps.

Proof: Assume we have an abstract XAG with r steps that does not fulfill the stated property. That is, there exists some step i for which $S_{1i} \subseteq S_{2i}$ or $S_{2i} \subseteq S_{1i}$. Without loss of generality, let us assume that $S_{1i} \subseteq S_{2i}$. Then due to Lemma 1, we can rewrite that step as

$$x_i = L_{S_{1i}} \wedge S_{S_{2i} \setminus S_{1i}},$$

and replace all $S' \in \{S\} \cup \{S_{1j}, S_{2j} \mid j > i\}$ by $S' \triangle S_{1i}$, whenever $i \in S'$. We repeat this process for all gates that do not fulfill the property, and since a change only affects gates with a larger index, this procedure eventually terminates. ∎

Based on this result, we add the clauses

$$\bigvee_{j=1}^{i-1} (s_{1ij} \wedge \bar{s}_{2ij}) \wedge \bigvee_{j=1}^{i-1} (\bar{s}_{1ij} \wedge s_{2ij}) \tag{10}$$

for all $n < i \leq n+r$ to the SAT formula, thereby ruling out abstract XAGs with the property in Theorem 1.

Multi-level subset relation: More symmetry breaking can be taken into account when considering subset relations among linear functions that are not from the same AND gate. For this purpose, we make use of the following lemma, which is proven in Appendix A.

Lemma 2: Let S, T, and U such that $S \subseteq T$, and $S \subseteq U$. Then

$$L_S \oplus (L_T \wedge L_U) = L_{T \setminus S} \oplus (L_T \wedge L_{T \triangle U}). \tag{11}$$

On the left-hand side, by definition $S \subseteq T$ and $S \subseteq U$. However, on the right-hand side, $T \setminus S \subseteq T$, but $T \setminus S \subseteq T \triangle U$, if and only if $T \cap U = S$. Therefore, we can restrict the number of solutions, by constraining that $S \subseteq T$ and $S \subseteq U$ implies that $T \cap U = S$, by adding the clauses

$$\bigwedge_{i=n+1}^{|\hat{S}_l|} \left(\Big(\hat{s}_{li} \wedge \bigwedge_{j=1}^{i-1} (\hat{s}_{lj} \rightarrow s_{1ij} s_{2ij}) \Big) \rightarrow \bigwedge_{j=1}^{i-1} (\hat{s}_{lj} \leftrightarrow s_{1ij} s_{2ij}) \right) \tag{12}$$

for all $1 \leq l \leq 2r+1$. In that case, the SAT solver, will rule out the left-hand side expression in (11), unless $T \cap U = S$.

C. Solving strategies

In this section we describe different solving strategies to solve the optimization problem of finding an XAG with the minimum number of AND gates for an n-variable Boolean function f. For this purpose, we make use of the following notation. Function $F_r(x)$ describes the conjunction of the main clauses (5) and (6) for some given number of steps r and input assignment x. The function A_r is a conjunction of additional and symmetry breaking constraints described in the previous section. Note that A_r does not necessarily need to contain all of these constraints. The value r_{low} is a lower bound for r, and is either set to $d-1$, where d is the algebraic degree of f, or an application specific value. The function solve solves a SAT formula using a SAT solver and returns either sat or unsat, and the function extract_xag extracts an XAG from the last satisfying SAT call.

Direct method: The direct method solves the optimization problem as follows:

```
set r ← r_low;
while solve(⋀_{x=1}^{2^n−1} F_r(x) ∧ A_r) = unsat do
    set r ← r + 1;
end
return extract_xag();
```

Starting from a lower bound, it constrains all input assignments (except $x = 0$), and returns one XAG for the first satisfying solution.

Counter-example guided abstraction refinement: In this method inspired by [24] we do not constrain all input assignments, but call the SAT solver incrementally, constraining new input assignments that are derived from counter-examples of wrong solutions:

```
set r ← r_low;
while true do
    set F ← A_r;
    while solve(F) = sat do
        set N ← extract_xag();
        set f' ← simulate(N);
        if f' = f then
            return N;
        else
            set x ← min_x{f(x) ≠ f'(x)};
            set F ← F ∧ F_r(x);
        end
    end
    set r ← r + 1;
end
```

After each satisfying SAT call, we extract a candidate XAG N and simulate it to extract its function f'. If $f' = f$, then N is a minimum XAG, otherwise we refine F by constraining the smallest input assignment in which f and f' differ.

Multiple solutions: Next, we assume that we have found some r, for which there exists an XAG that implements f.

```
set N ← ∅;
set F ← ⋀_{x=1}^{2^n−1} F_r(x) ∧ A_r;
while solve(F) = sat do
    set N ← N ∪ {extract_xag()};
    set F ← F ∧ block();
end
return N;
```

While F is satisfiable and solutions can be extracted, we iteratively block solutions, by adding a single large clause that contains all variables $\hat{s}_{lj}$ in a polarity opposite to their value in the last satisfying solution.

IV. Optimizing the Number of XOR Gates

There exist several differently structured abstract XAGs, all having the same number of steps, and all realizing the same Boolean function f. Further, for each of these abstract XAGs there exists several possible ways to transform it into an XAG without changing its structure, i.e., without changing the linear functions that fan-in to the AND gates. Finding the XAG with the smallest number of XOR gates for a given number of AND gates essentially requires to solve the SAT problem on the XAG structure rather than on the abstract XAG, which makes the problem much more complicated. Instead, in this section, we propose a method that (i) heuristically determines an abstract XAG that potentially leads to a small number of XOR gates in the XAG, and (ii) finds an XAG with the smallest number of XOR gates for that particular abstract XAG exactly using a SAT-based method.

A. Heuristic to find good abstract XAG as starting point

The number of XOR gates is possibly small, if the linear functions in the abstract XAG have small index sets. In other words, we aim to minimize the sum $|\hat{S}_1| + \cdots + |\hat{S}_{2r+1}|$. For this purpose, we add the constraint

$$\sum_{l=1}^{2r+1} \sum_{j=1}^{|\hat{S}_l|} \hat{s}_{lj} < p \tag{13}$$

to an otherwise satisfying instance to find an abstract XAG with r gates, where r is already known to be optimum. An initial value for p can be extracted from a known solution, and then incrementally made smaller until no more satisfying solution can be found. It is a common approach to implement such a search using a sorter network, which has as inputs all variables $\hat{s}_{lj}$. We can then constrain (13) by forcing the p^{th} most significant output of the sorter network to be 0 [25].

B. Finding the shortest linear network with SAT

Next we describe a SAT-based approach that finds the smallest XAG representation in terms of XOR gates for a given abstract XAG. For this purpose, we introduce multi-ouput linear networks. Let $L_{S_1}, \ldots, L_{S_m}$ be m linear functions over n variables. We can represent all functions by a single $m \times n$ Boolean matrix $A = (a_{lj})$ with $a_{lj} = [j \in S_l]$. A

linear network (or linear straight-line program) for m linear functions over n variables is a sequence of steps

$$x_i = x_{j_{1i}} \oplus x_{j_{2i}} \tag{14}$$

for $n < i \leq n+r$ and a mapping $f_l \in [n+r]$ for $1 \leq l \leq m$.

Given an abstract XAG over n variables and r steps, we can extract a linear network over $n+r$ variables and $2r+1$ linear functions, where variables $x_1, \ldots, x_{n+r}$ correspond to the original inputs and steps in the abstract XAG and the $2r+1$ linear functions are $S_{1(n+1)}, S_{2(n+1)}, \ldots, S_{1(n+r)}, S_{2(n+r)}, S$, in that order. From the linear network we can then construct an XAG, where the inputs $x_{n+1}, \ldots, x_{n+r}$ correspond to the outputs of an AND gate, and where all but the last linear function correspond to the inputs of an AND gate. To ensure that the XAG corresponds to a directed acyclic graph, we must control how steps are computed in the linear network. For example, in order to compute a function that corresponds to an input of AND gate x_i, we cannot use any variable with index i or higher.

For a single linear function L_S, where $S \neq \emptyset$, the shortest linear network requires $r = |S| - 1$ steps. Determining the smallest linear network for a set of linear functions is more complicated, since one needs to take step sharing and step cancellations into account. Boyar, Matthews, and Peralta have shown that finding a linear network with the smallest number of steps for some set of linear functions is MaxSNP-complete [26].

Fuhs and Schneider-Kamp have presented a SAT-based approach to find the shortest linear network for a given Boolean matrix A [21]. Similarly to the approach presented in this paper, they are solving a sequence of decision problems that ask whether there exists a linear network with r steps. Their SAT encoding for this decision problem consists of variables b_{ij} that encode whether step x_i uses input or step x_j for $n < i \leq n+r$ and $1 \leq j < i$, and variables f_{li} that encode whether step x_i computes output f_l for $n < i \leq n+r$ and $1 \leq l \leq m$. For simplicity, we assume that each row in A has at least two 1s, i.e., no output is a constant-0 or projection function. Then, the constraints

$$\sum_{j=1}^{i-1} b_{ij} = 2 \quad \text{for } n < i \leq n+r \tag{15}$$

ensure that each step has two inputs and the constraints

$$\sum_{i=n+1}^{n+r} f_{li} = 1 \quad \text{for } 1 \leq l \leq m \tag{16}$$

ensure that each output is computed by exactly one step. The main clauses

$$\bigwedge_{i=n+1}^{n+r} \left(f_{li} \rightarrow \bigwedge_{j=1}^{n} (\psi(j,i) \leftrightarrow a_{lj}) \right) \quad \text{for } 1 \leq j \leq m \tag{17}$$

ensure that if the function f_l is computed by step x_i, the linear function at that step matches the entries in row l of A. For this purpose, the recursive function

$$\psi(j,i) = b_{ij} \oplus \bigoplus_{i'=n+1}^{i-1} b_{ii'} \wedge \psi(j,i') \tag{18}$$

encodes whether variable x_j is in the linear function computed by step x_i for $1 \leq j \leq n$ and $n < i \leq n+r$. Further constraints to remove redundant solutions or break symmetries can speed up the SAT solver and are discussed in [21].

As described above, a linear network extracted from an abstract XAG must obey additional constraints to ensure that linear functions that correspond to input of gate x_i do not make use of variables that correspond to the output of a gate x_j with $j > i$. One possibility to ensure these constraints is to force the linear network to be cancellation-free [26], however, this is unnecessarily restrictive. A better way is to forbid using some inputs in the computation of an output. For this purpose, we extend the original SAT encoding to find the shortest linear network by the recursive function

$$\varphi(j,i) = b_{ij} \vee \bigvee_{i'=n+1}^{i-1} b_{ii'} \wedge \varphi(j,i'), \tag{19}$$

for $1 \leq j \leq n$ and $n < i \leq n+r$, which is similar to $\psi(j,i)$ in (18), but captures whether variable x_j has been used in any intermediate step to compute x_i. If we wish to enforce cancellation-free logic networks, we can add the constraints $\psi(j,i) \leftrightarrow \varphi(j,i)$, however, for our application it is sufficient to simply forbid that some outputs cannot use some inputs by constraining

$$f_{li} \rightarrow \bar{b}_{ij} \tag{20}$$

for all $n < i \leq n+r$, $1 \leq l < m$ and $n + \lceil \frac{l}{2} \rceil \leq j \leq n+r$.

V. Experimental Results

We have implemented the algorithm using the EPFL logic synthesis libraries [27]. In our experiments, we used the exact synthesis algorithms to verify known multiplicative complexities for Boolean functions with up to 6 variables [9], [20]. More precisely, we found optimum XAGs for affine-equivalent classes, since the multiplicative complexity is invariant for all functions in a class. As representative function, we chose the function determined by the classification algorithm in [22]. The experiment *affine*(n) finds optimum XMGs for all affine equivalent class representatives for n-variable Boolean functions, starting from a lower bound based on the function's algebraic degree. There are 8 and 48 classes for all 4- and 5-variable Boolean functions, respectively [28]. The experiment *practical6*(k) finds optimum XAGs for the k most occurring 6-variable Boolean functions found by enumerating cuts in all arithmetic and random control benchmarks of the EPFL logic synthesis benchmark suite [29]. Appendix B shows how these functions are obtained using ABC [30].

We report the total number of 2-input XOR gates, the total number of 2-input AND gates, and the overall run-time in

TABLE I
EXPERIMENTAL RESULTS

Experiment	mult.	XOR opt.	#XOR	#AND	Runtime
all-affine(4)			44	16	0.13
all-affine(4)	✓		9	16	0.93
all-affine(4)	✓	heur.	5	16	0.45
all-affine(4)	✓	SAT	5	16	0.48
all-affine(5)			508	162	125.98
all-affine(5)	✓		245	162	251.26
all-affine(5)	✓	heur.	179	162	412.26
all-affine(5)	✓	SAT	173	162	530.10
practical6(100)			1161	293	558.95
practical6(100)	✓		1022	293	663.76
practical6(100)	✓	heur.	981	293	3888.42
practical6(100)	✓	SAT	771	293	5881.60

seconds. We perform four runs for each experiment: The first run finds a single optimum XAG; the second run uses the solving strategy from Sect. III-C to find up to 50 optimum XAGs and then selects the one with the fewest number of XOR gates; the third run also finds up to 50 optimum XAGs but uses the XOR minimization heuristic based on sorter networks presented in Sect. IV-A; the fourth run is like the third run, but additionally optimizes XOR gates in the final solution using the SAT-based approach in Sect. IV-B.

We run all the experiments on a Microsoft Azure virtual machine, on a general purpose Standard D2 v3 size configuration, running on a Intel Xeon Platinum 8171M 2.60GHz CPU with 8 GiB memory and Ubuntu 18.04. We use Z3 [31] as a SAT solver. We also tried ABC's [30] modified implementation of MiniSAT [32], Glucose [33], and MapleSAT [34], which both performed slower in these experiments. We set conflict limits for the SAT solvers used in abstract XAG optimization and linear network optimization of 50,000 and 500,000 conflicts, respectively. These conflicts are only applied once the first optimum solution has been found.

We further used our exact abstract XAG algorithm together with the XOR minimization strategy to find small XAGs for all 48 affine equivalence classes for 5-variable Boolean functions, such that the class representative matches the function that is returned by the classification algorithm in [22]. Table II lists all XAGs together with the truth table representation of each function in hexadecimal notation and also lists the corresponding multiplicative complexity. Such a table can be used in AND minimization techniques such as the rewriting algorithm presented in [5].

VI. CONCLUSIONS

We presented a SAT-based algorithm to determine the multiplicative complexity of a Boolean function. The algorithm is only applicable to small Boolean functions. The multiplicative complexity is known for all Boolean functions up to 6 variables, however, the approaches that were used to determine these numbers are based on the exhaustive enumeration of all 150,357 equivalence classes. Instead, our approach can be used to find one or multiple solutions for one particular Boolean function of interest. Therefore, for functions with up to 6 variables, it is of interest when the goal is to explore different structures for the same function. For functions with more than 6 variables, it can find solutions, if the multiplicative complexity is low, or can prove that no solution with a small number of AND gates can exist.

Future insight in the structure of XAGs can be exploited as additional symmetry breaking rules in order to speed up solving time. Similarly, advances in SAT solvers have a positive impact on our proposed algorithm.

Acknowledgments: We like to thank Thomas Häner, Robin Kothari, René Peralta, Michael Miller, Bruno Schmitt, and Eleonora Testa for valuable feedback in the preparation of this manuscript.

REFERENCES

[1] C.-P. Schnorr, "The multiplicative complexity of Boolean functions," in *Int'l Conf. on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, 1988, pp. 45–58.

[2] L. A. Levin, "The tale of one-way functions," *Problems of Information Transmission*, vol. 39, no. 1, pp. 92–103, 2003.

[3] M. G. Find, "On the complexity of computing two nonlinearity measures," in *Int'l Computer Science Symposium in Russia*, 2014, pp. 167–175.

[4] J. Boyar, P. Matthews, and R. Peralta, "Logic minimization techniques with applications to cryptology," *Journal of Cryptology*, vol. 26, no. 2, pp. 280–312, 2013.

[5] E. Testa, M. Soeken, L. G. Amarù, and G. De Micheli, "Reducing the multiplicative complexity in logic networks for cryptography and security applications," in *Design Automation Conference*, 2019, p. 74.

[6] M. S. Riazi, M. Javaheripi, S. U. Hussain, and F. Koushanfar, "MPCircuits: Optimized circuit generation for secure multi-party computation," in *Int'l Symp. on Hardware-Oriented Security and Trust*, 2019, pp. 198–207.

[7] S. Cimato, V. Ciriani, E. Damiani, and M. Ehsanpour, "An OBDD-based technique for the efficient synthesis of garbled circuits," in *Int'l Workshop on Security and Trust Management*, ser. Lecture Notes in Computer Science, S. Mauw and M. Conti, Eds., vol. 11738. Springer, 2019, pp. 158–167.

[8] E. Testa, M. Soeken, H. Riener, and G. Luca Gaetano Amarù, De Micheli, "A logic synthesis toolbox for reducing the multiplicative complexity in logic networks," in *Design, Automation and Test in Europe*, 2020.

[9] M. S. Turan and R. Peralta, "The multiplicative complexity of Boolean functions on four and five variables," in *Int'l Workshop on Lightweight Cryptography for Security and Privacy*, ser. Lecture Notes in Computer Science, T. Eisenbarth and E. Öztürk, Eds., vol. 8898. Springer, 2014, pp. 21–33.

[10] D. H. Nicolas T. Courtois and T. Mourouzis, "Solving circuit optimisation problems in cryptography and cryptanalysis," Cryptology ePrint Archive, Report 2011/475, 2011, https://eprint.iacr.org/2011/475.

[11] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for MPC and FHE," in *Int'l Conf. on the Theory and Applications of Cryptographic Techniques*, 2015, pp. 430–454.

[12] M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha, "Post-quantum zero-knowledge and signatures from symmetric-key primitives," in *ACM SIGSAC Conf. on Computer and Communications Security*, 2017, pp. 1825–1842.

[13] I. Giacomelli, J. Madsen, and C. Orlandi, "ZKBoo: Faster zero-knowledge for Boolean circuits," in *USENIX Security Symposium*, 2016, pp. 1069–1083.

[14] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly compressed and scalable sequential garbled circuits," in *IEEE Symp. on Security and Privacy*, 2015, pp. 411–428.

[15] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *Int'l. Coll. on Automata, Languages, and Programming*, 2008, pp. 486–498.

[16] G. Meuli, M. Soeken, E. Campbell, M. Roetteler, and G. De Micheli, "The role of multiplicative complexity in compiling low T-count oracle circuits," in *Int'l Conf. on Computer-Aided Design*, 2019, pp. 1–8.
[17] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, "Finding efficient circuits using SAT-solvers," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2009, pp. 32–44.
[18] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.
[19] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Michaeli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 871–884, 2020.
[20] Ç. Çalik, M. S. Turan, and R. Peralta, "The multiplicative complexity of 6-variable Boolean functions," *Cryptography and Communications*, vol. 11, no. 1, pp. 93–107, 2019.
[21] C. Fuhs and P. Schneider-Kamp, "Synthesizing shortest linear straight-line programs over GF(2) using SAT," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2010, pp. 71–84.
[22] M. Soeken, E. Testa, and D. M. Miller, "A hybrid method for spectral translation equivalent Boolean functions," in *Pacific Rim Conference on Communications, Computers and Signal Processing*, 2019, pp. 1–6.
[23] G. S. Tseytin, "On the complexity of derivation in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics*, A. P. Slisenko, Ed. Springer, 1970, pp. 115–125.
[24] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.
[25] M. Codish and M. Zazon-Ivry, "Pairwise cardinality networks," in *Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 154–172.
[26] J. Boyar, P. Matthews, and R. Peralta, "On the shortest linear straight-line program for computing linear forms," in *Int'l Symp. on Mathematical Foundations of Computer Science*, 2008, pp. 168–179.
[27] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL logic synthesis libraries," *arXiv preprint arXiv:1805.05121v2*, 2018.
[28] M. A. Harrison, "On the classification of Boolean functions by the general linear and affine groups," *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, no. 2, pp. 285–299, 1964.
[29] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Int'l Workshop on Logic and Synthesis*, 2015.
[30] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.
[31] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.
[32] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.
[33] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Int'l Joint Conf. on Artificial Intelligence*, 2009, pp. 399–404.
[34] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2016, pp. 123–140.

APPENDIX

A. Proof of Lemma 2

We introduce a notation and identities for a subset of quadratic forms, which will be useful for the proof. Quadratic forms are algebraic normal forms, in which all monomials have degree 2. A subset of quadratic forms can be constructed for two disjoint sets S and T, i.e., $S \cap T = \emptyset$. We define

$$Q_{ST} = L_S \wedge L_T = \bigoplus_{(i,j) \in S \times T} x_i x_j. \tag{21}$$

Note that Q_{ST} contains of $|S| \times |T|$ monomials of degree 2, since no cancellation can take place. Also note that $Q_{ST} = Q_{TS}$. Let S_1, S_2, and T such that all three sets are pairwise disjoint. Then

$$Q_{S_1T} \oplus Q_{S_2T} = Q_{(S_1 \cup S_2)T}. \tag{22}$$

Further, for any two sets S and T, we have

$$L_S \oplus L_T = L_{S \triangle T} \tag{23}$$

and

$$L_S \wedge L_T = L_{S \cap T} \oplus Q_{(S \cap T)(S \triangle T)} \oplus Q_{(S \setminus T)(T \setminus S)} \tag{24}$$

We prove one non-trivial lemma, based on these identities:

Lemma 3: Given two sets T and U, we have $L_T \wedge L_{T \triangle U} = L_T \oplus (L_T \wedge L_U)$.

Proof: Since $T \cap (T \triangle U) = T \setminus U$, $T \triangle (T \triangle U) = U$, $T \setminus (T \triangle U) T \cap U$, and $(T \triangle U) \setminus T = U \setminus T$, by expanding the left-hand side of the lemma using (24), we get

$$L_{T \setminus U} \oplus Q_{(T \setminus U)U} \oplus Q_{(T \cap U)(U \setminus T)}.$$

We add twice the term $L_{T \cap U}$, which does not change the function due to cancellation, and also expand the first quadratic form on $U = (T \cap U) \cup (U \setminus T)$ using (22):

$$L_{T \setminus U} \oplus L_{T \cap U} \oplus L_{T \cap U} \oplus Q_{(T \setminus U)(T \cap U)} \oplus Q_{(T \setminus U)(U \setminus T)} \oplus Q_{(T \cap U)(U \setminus T)}.$$

Applying (23) to the first two linear forms, and (22) to the first and third quadratic form gives

$$L_T \oplus L_{T \cap U} \oplus Q_{(T \cap U)(T \triangle U)} \oplus Q_{(T \setminus U)(U \setminus T)},$$

which simplifies to $L_T \oplus (L_T \wedge L_U)$ after applying (24). ■

We can now prove Lemma 2, and show that $L_{T \setminus S} \oplus (L_T \wedge L_{T \triangle U}) = L_S \oplus (L_T \wedge L_U)$, if $S \subseteq T$ and $S \subseteq U$.

Proof:

$$L_{T \setminus S} \oplus (L_T \wedge L_{T \triangle U}) \overset{\text{Lemma 3}}{=} L_{T \setminus S} \oplus L_T \oplus (L_T \wedge L_U)$$

Since $S \subseteq T$, we have $(T \setminus S) \triangle T = S$, and applying (23) yields $L_S \oplus (L_T \wedge L_U)$. ■

B. Functions in *practical6*(n, k)

The following procedure generates the k practical 6-variable functions in *practical6*(n, k) using ABC [30] and the arithmetic and random control instances of the EPFL logic synthesis benchmarks [29].

```
$ abc
abc> arithmetic/adder.aig; cut -s
abc> arithmetic/bar.aig; cut -s
...
abc> random_control/router.aig; cut -s
abc> random_control/voter.aig; cut -s
abc> npnsave functions; quit
$ cat functions | grep -e "6$" | head -nk
                | cut -d ' ' -f1
```

TABLE II
OPTIMUM XAGS FOR ALL 48 AFFINE EQUIVALENT CLASS FOR 5-VARIABLE BOOLEAN FUNCTIONS

Class	Function	Logic network	MC
0	#00000000	$f = x_0$	0
1	#80000000	$x_6 = x_1 \wedge x_2, x_7 = x_3 \wedge x_4, x_8 = x_6 \wedge x_7, x_9 = x_5 \wedge x_8$	4
2	#80008000	$x_5 = x_1 \wedge x_2, x_6 = x_3 \wedge x_4, x_7 = x_5 \wedge x_6$	3
3	#00808080	$x_6 = x_1 \wedge x_2, x_7 = x_3 \wedge x_6, x_8 = x_4 \wedge x_7, x_9 = x_5 \wedge x_8, x_{10} = x_7 \oplus x_9$	4
4	#80808080	$x_4 = x_1 \wedge x_2, x_5 = x_3 \wedge x_4$	2
5	#08888000	$x_6 = x_1 \wedge x_2, x_7 = x_3 \wedge x_4, x_8 = x_5 \oplus x_7, x_9 = x_6 \wedge x_8$	3
6	#aa2a2a80	$x_6 = x_2 \wedge x_3, x_7 = x_4 \oplus x_6, x_8 = x_1 \wedge x_7, x_9 = x_4 \wedge x_8, x_{10} = x_1 \oplus x_9, x_{11} = x_5 \wedge x_{10}, x_{12} = x_8 \oplus x_{11}$	4
7	#88080808	$x_6 = x_1 \wedge x_2, x_7 = x_3 \wedge x_6, x_8 = x_4 \wedge x_7, x_9 = x_5 \wedge x_8, x_{10} = x_6 \oplus x_9, x_{11} = x_7 \oplus x_{10}$	4
8	#2888a000	$x_6 = x_2 \wedge x_5, x_7 = x_3 \wedge x_4, x_8 = x_6 \oplus x_7, x_9 = x_1 \wedge x_8$	3
9	#f7788000	$x_6 = x_3 \wedge x_4, x_7 = x_1 \wedge x_2, x_8 = x_5 \oplus x_6, x_9 = x_3 \oplus x_4, x_{10} = x_5 \oplus x_9, x_{11} = x_7 \oplus x_{10}, x_{12} = x_8 \wedge x_{11}, x_{13} = x_5 \oplus x_{12}$	3
10	#a8202020	$x_6 = x_4 \wedge x_5, x_7 = x_3 \oplus x_6, x_8 = x_2 \wedge x_7, x_9 = x_3 \oplus x_8, x_{10} = x_1 \wedge x_9$	3
11	#08880888	$x_5 = x_1 \wedge x_2, x_6 = x_3 \wedge x_5, x_7 = x_4 \wedge x_6, x_8 = x_5 \oplus x_7$	3
12	#bd686868	$x_6 = x_4 \wedge x_5, x_7 = x_2 \oplus x_6, x_8 = x_1 \wedge x_7, x_9 = x_2 \wedge x_3, x_{10} = x_3 \oplus x_8, x_{11} = x_1 \oplus x_9, x_{12} = x_{10} \wedge x_{11}, x_{13} = x_6 \oplus x_{12}$	4
13	#aa808080	$x_6 = x_2 \wedge x_3, x_7 = x_4 \wedge x_5, x_8 = x_6 \wedge x_7, x_9 = x_6 \oplus x_7, x_{10} = x_8 \oplus x_9, x_{11} = x_1 \wedge x_{10}$	4
14	#7e686868	$x_6 = x_4 \wedge x_5, x_7 = x_2 \oplus x_6, x_8 = x_1 \oplus x_6, x_9 = x_7 \wedge x_8, x_{10} = x_1 \wedge x_3, x_{11} = x_3 \oplus x_9, x_{12} = x_7 \oplus x_{10}, x_{13} = x_{11} \wedge x_{12}, x_{14} = x_6 \oplus x_{13}$	4
15	#2208a208	$x_6 = x_1 \wedge x_2, x_7 = x_3 \wedge x_6, x_8 = x_5 \wedge x_7, x_9 = x_4 \oplus x_7, x_{10} = x_6 \oplus x_9, x_{11} = x_1 \oplus x_8, x_{12} = x_{10} \wedge x_{11}$	4
16	#08888888	$x_6 = x_1 \wedge x_2, x_7 = x_3 \wedge x_6, x_8 = x_4 \wedge x_7, x_9 = x_5 \wedge x_8, x_{10} = x_6 \oplus x_9$	4
17	#88888888	$x_3 = x_1 \wedge x_2$	1
18	#ea404040	$x_6 = x_2 \wedge x_3, x_7 = x_4 \wedge x_5, x_8 = x_6 \oplus x_7, x_9 = x_1 \wedge x_8, x_{10} = x_6 \oplus x_9$	3
19	#2a802a80	$x_5 = x_2 \wedge x_3, x_6 = x_4 \oplus x_5, x_7 = x_1 \wedge x_6$	2
20	#73d28c88	$x_6 = x_1 \wedge x_2, x_7 = x_2 \oplus x_5, x_8 = x_6 \oplus x_7, x_9 = x_4 \wedge x_8, x_{10} = x_5 \oplus x_9, x_{11} = x_1 \oplus x_3, x_{12} = x_{10} \wedge x_{11}, x_{13} = x_6 \oplus x_9, x_{14} = x_{12} \oplus x_{13}$	3
21	#ea808080	$x_6 = x_2 \wedge x_3, x_7 = x_1 \oplus x_6, x_8 = x_4 \wedge x_5, x_9 = x_1 \oplus x_8, x_{10} = x_7 \wedge x_9, x_{11} = x_1 \oplus x_{10}$	3
22	#a28280a0	$x_6 = x_3 \wedge x_4, x_7 = x_5 \oplus x_6, x_8 = x_1 \oplus x_2, x_9 = x_7 \wedge x_8, x_{10} = x_3 \oplus x_9, x_{11} = x_1 \wedge x_{10}$	3
23	#13284c88	$x_6 = x_1 \wedge x_3, x_7 = x_1 \oplus x_6, x_8 = x_2 \wedge x_7, x_9 = x_4 \oplus x_6, x_{10} = x_5 \oplus x_8, x_{11} = x_2 \oplus x_{10}, x_{12} = x_9 \wedge x_{11}, x_{13} = x_8 \oplus x_{12}$	3
24	#a2220888	$x_6 = x_3 \wedge x_4, x_7 = x_2 \wedge x_6, x_8 = x_2 \oplus x_7, x_9 = x_5 \oplus x_8, x_{10} = x_1 \wedge x_9$	3
25	#aae6da80	$x_6 = x_2 \wedge x_3, x_7 = x_5 \oplus x_6, x_8 = x_1 \oplus x_2, x_9 = x_7 \wedge x_8, x_{10} = x_6 \oplus x_9, x_{11} = x_3 \oplus x_{10}, x_{12} = x_5 \wedge x_{11}, x_{13} = x_3 \oplus x_{12}, x_{14} = x_1 \oplus x_{13}, x_{15} = x_4 \wedge x_{14}, x_{16} = x_9 \oplus x_{15}, x_{17} = x_6 \oplus x_{16}$	4
26	#58d87888	$x_6 = x_1 \wedge x_2, x_7 = x_1 \oplus x_4, x_8 = x_3 \wedge x_7, x_9 = x_3 \oplus x_8, x_{10} = x_5 \wedge x_9, x_{11} = x_3 \oplus x_6, x_{12} = x_1 \oplus x_{10}, x_{13} = x_{11} \wedge x_{12}, x_{14} = x_8 \oplus x_{13}$	4
27	#8c88ac28	$x_6 = x_3 \wedge x_5, x_7 = x_2 \oplus x_6, x_8 = x_1 \wedge x_7, x_9 = x_2 \wedge x_4, x_{10} = x_3 \oplus x_8, x_{11} = x_1 \oplus x_9, x_{12} = x_{10} \wedge x_{11}, x_{13} = x_9 \oplus x_{12}$	4
28	#8880f880	$x_6 = x_1 \wedge x_2, x_7 = x_3 \oplus x_6, x_8 = x_4 \wedge x_7, x_9 = x_5 \wedge x_8, x_{10} = x_6 \oplus x_9, x_{11} = x_3 \wedge x_{10}, x_{12} = x_8 \oplus x_{11}$	4
29	#9ee8e888	$x_6 = x_1 \wedge x_2, x_7 = x_3 \oplus x_6, x_8 = x_4 \oplus x_7, x_9 = x_5 \wedge x_8, x_{10} = x_3 \wedge x_4, x_{11} = x_9 \oplus x_{10}, x_{12} = x_1 \oplus x_5, x_{13} = x_2 \oplus x_{12}, x_{14} = x_9 \oplus x_{13}, x_{15} = x_{11} \wedge x_{14}, x_{16} = x_6 \oplus x_{15}$	4
30	#4268c268	$x_6 = x_2 \wedge x_3, x_7 = x_5 \wedge x_6, x_8 = x_2 \oplus x_4, x_9 = x_6 \oplus x_8, x_{10} = x_1 \oplus x_7, x_{11} = x_9 \wedge x_{10}, x_{12} = x_3 \oplus x_{11}, x_{13} = x_1 \wedge x_{12}, x_{14} = x_6 \oplus x_{13}$	4
31	#16704c80	$x_6 = x_1 \wedge x_5, x_7 = x_2 \oplus x_6, x_8 = x_4 \wedge x_7, x_9 = x_1 \wedge x_2, x_{10} = x_5 \oplus x_9, x_{11} = x_3 \wedge x_{10}, x_{12} = x_8 \oplus x_{11}$	4
32	#78888888	$x_6 = x_3 \wedge x_4, x_7 = x_5 \wedge x_6, x_8 = x_1 \wedge x_2, x_9 = x_7 \oplus x_8$	3
33	#4966bac0	$x_6 = x_1 \wedge x_4, x_7 = x_2 \wedge x_3, x_8 = x_5 \oplus x_6, x_9 = x_1 \oplus x_7, x_{10} = x_8 \wedge x_9, x_{11} = x_3 \oplus x_5, x_{12} = x_4 \oplus x_6, x_{13} = x_2 \oplus x_{12}, x_{14} = x_{11} \wedge x_{13}, x_{15} = x_{10} \oplus x_{14}$	4
34	#372840a0	$x_6 = x_1 \wedge x_2, x_7 = x_1 \oplus x_6, x_8 = x_3 \wedge x_7, x_9 = x_2 \wedge x_3, x_{10} = x_8 \oplus x_9, x_{11} = x_5 \oplus x_{10}, x_{12} = x_4 \oplus x_6, x_{13} = x_{11} \wedge x_{12}, x_{14} = x_8 \oplus x_{13}$	4
35	#5208d288	$x_6 = x_1 \wedge x_2, x_7 = x_5 \wedge x_6, x_8 = x_4 \oplus x_7, x_9 = x_1 \oplus x_3, x_{10} = x_8 \wedge x_9, x_{11} = x_6 \oplus x_7, x_{12} = x_{10} \oplus x_{11}$	3
36	#7ca00428	$x_6 = x_1 \oplus x_4, x_7 = x_2 \wedge x_6, x_8 = x_5 \wedge x_7, x_9 = x_1 \oplus x_5, x_{10} = x_7 \oplus x_9, x_{11} = x_4 \wedge x_{10}, x_{12} = x_2 \oplus x_3, x_{13} = x_1 \oplus x_{11}, x_{14} = x_{12} \wedge x_{13}, x_{15} = x_8 \oplus x_{14}$	4
37	#f8880888	$x_6 = x_1 \wedge x_2, x_7 = x_5 \oplus x_6, x_8 = x_3 \wedge x_7, x_9 = x_4 \wedge x_8, x_{10} = x_6 \oplus x_9$	3
38	#2ec0ae40	$x_6 = x_3 \oplus x_4, x_7 = x_2 \wedge x_6, x_8 = x_4 \oplus x_7, x_9 = x_1 \wedge x_8, x_{10} = x_5 \wedge x_9, x_{11} = x_7 \oplus x_{10}, x_{12} = x_2 \wedge x_{11}, x_{13} = x_9 \oplus x_{12}$	4
39	#f888f888	$x_5 = x_1 \wedge x_2, x_6 = x_3 \wedge x_4, x_7 = x_5 \wedge x_6, x_8 = x_5 \oplus x_7, x_9 = x_6 \oplus x_8$	3
40	#58362ec0	$x_6 = x_3 \wedge x_5, x_7 = x_4 \oplus x_5, x_8 = x_6 \oplus x_7, x_9 = x_1 \oplus x_2, x_{10} = x_8 \wedge x_9, x_{11} = x_4 \oplus x_{10}, x_{12} = x_1 \wedge x_{11}, x_{13} = x_3 \oplus x_{12}, x_{14} = x_2 \oplus x_6, x_{15} = x_{13} \wedge x_{14}, x_{16} = x_{10} \oplus x_{15}$	4
41	#0eb8f6c0	$x_6 = x_3 \wedge x_5, x_7 = x_1 \wedge x_4, x_8 = x_5 \oplus x_7, x_9 = x_1 \oplus x_6, x_{10} = x_8 \wedge x_9, x_{11} = x_2 \oplus x_7, x_{12} = x_4 \oplus x_{11}, x_{13} = x_2 \oplus x_{10}, x_{14} = x_3 \oplus x_6, x_{15} = x_{13} \oplus x_{14}, x_{16} = x_{12} \wedge x_{15}, x_{17} = x_6 \oplus x_{16}, x_{18} = x_{11} \oplus x_{17}$	4
42	#567cea40	$x_6 = x_1 \wedge x_4, x_7 = x_5 \wedge x_6, x_8 = x_2 \oplus x_7, x_9 = x_1 \oplus x_8, x_{10} = x_3 \wedge x_9, x_{11} = x_5 \oplus x_{10}, x_{12} = x_2 \oplus x_3, x_{13} = x_{11} \wedge x_{12}, x_{14} = x_6 \oplus x_{10}, x_{15} = x_{13} \oplus x_{14}$	4
43	#f8887888	$x_6 = x_1 \wedge x_2, x_7 = x_3 \wedge x_4, x_8 = x_6 \wedge x_7, x_9 = x_5 \wedge x_8, x_{10} = x_6 \oplus x_9, x_{11} = x_7 \oplus x_{10}$	4
44	#78887888	$x_5 = x_1 \wedge x_2, x_6 = x_3 \wedge x_4, x_7 = x_5 \oplus x_6$	2
45	#e72890a0	$x_6 = x_2 \wedge x_5, x_7 = x_3 \oplus x_6, x_8 = x_1 \wedge x_7, x_9 = x_2 \oplus x_4, x_{10} = x_3 \wedge x_9, x_{11} = x_5 \oplus x_{10}, x_{12} = x_4 \wedge x_{11}, x_{13} = x_8 \oplus x_{12}$	4
46	#268cea40	$x_6 = x_2 \wedge x_3, x_7 = x_4 \oplus x_6, x_8 = x_1 \wedge x_7, x_9 = x_2 \wedge x_5, x_{10} = x_6 \oplus x_8, x_{11} = x_9 \oplus x_{10}$	3
47	#6248eac0	$x_6 = x_1 \wedge x_4, x_7 = x_3 \wedge x_6, x_8 = x_5 \oplus x_7, x_9 = x_1 \wedge x_8, x_{10} = x_3 \oplus x_9, x_{11} = x_2 \wedge x_{10}, x_{12} = x_6 \oplus x_{11}$	4