Reducing the Multiplicative Complexity in Logic Networks for Cryptography and Security Applications

Eleonora Testa
EPFL, Lausanne, Switzerland
eleonora.testa@epfl.ch

Luca Amarù
Synopsys Inc., Sunnyvale, CA, USA
luca.amaru@synopsys.com

Mathias Soeken
EPFL, Lausanne, Switzerland
mathias.soeken@epfl.ch

Giovanni De Micheli
EPFL, Lausanne, Switzerland
giovanni.demicheli@epfl.ch

ABSTRACT
Reducing the number of AND gates plays a central role in many cryptography and security applications. We propose a logic synthesis algorithm and tool to minimize the number of AND gates in a logic network composed of AND, XOR, and inverter gates. Our approach is fully automatic and exploits cut enumeration algorithms to explore optimization potentials in local subcircuits. The experimental results show that our approach can reduce the number of AND gates by 34% on average compared to generic size optimization algorithms. Further, we are able to reduce the number of AND gates up to 76% in best-known benchmarks from the cryptography community.

ACM Reference Format:

1 INTRODUCTION
Logic synthesis is considered one of the fundamental steps in the realization of competitive and leading-edge integrated circuits. In the last decades, both heuristics and exact methods have been proposed, together with new data structures, for the abstraction and manipulation of Boolean circuits. Classical data structures in logic synthesis work over the gate basis \{AND, OR, NOT\} [1], as they traditionally target CMOS-based applications. In recent years, new data structures based on majority function have also been considered for optimization of emerging nanotechnologies [2]. Despite using different data structures and algorithms, the optimization goals of logic synthesis are mainly area, delay, and power consumption of digital circuits. In this paper, we specifically extend logic synthesis to consider an alternative optimization objective for cryptography and security applications.

Reducing the number of AND gates also plays a crucial role in high-level cryptography protocols such as zero-knowledge protocols, fully homomorphic encryption (FHE) and secure multiparty computation (MPC) [8, 9]. In this scenario, AND gates are considered the “bottleneck” of the computation [8]. In particular, it has been demonstrated that in post-quantum zero-knowledge signatures based on “MPC-in-the-head” [10], the size of the signature is proportional to the number of AND gates used by the underlying blockcipher [9]. For MPC protocols based on Yao’s garbled circuits [11, 12] with the free XOR technique [13], the total number of computations depends on the multiplicative complexity. Further, for FHE, the XOR gates is considered much cheaper than the AND gates, and do not increase the noise during the computation. Further motivations for AND minimization also come from side-channel attacks. Indeed, in techniques to protect against side-channel attacks, the cost of general-purpose protections grows with the number of AND gates [6].

In view of all this, we propose logic synthesis for cryptographic applications, aiming at minimizing the number of AND gates in a circuit. We describe a cut rewriting algorithm to reduce the multiplicative complexity in Xor-And Graph (XAG). An XAG is a logic network consisting of AND gates, XOR gates, and inverters. While state-of-the-art methods rely heavily on manual decomposition and optimization strategies [14], our approach is fully automatic. Compared to generic size optimization algorithms [15], the proposed
method achieves lower numbers of AND gates for the EPFL combina-
tional benchmark suite [16]. On average, our proposed method re-
duces the number of AND gates by 34%. Moreover, we demon-
strate a significantly smaller number of AND gates in best-known
reported benchmarks in the context of MPC and FHE.

2 PRELIMINARIES

2.1 Xor-And Graphs and Multiplicative
Complexity

In many cryptographic applications, Boolean functions are usually
represented over the basis [AND, XOR, NOT] [3]. In analogy with
the data structures usually involved in logic synthesis, e.g., AND-
inverter graphs (AIGs, [1]), or majority-inverter graphs (MIGs, [17]),
in this work we represent logic networks from cryptographic appli-
cations in terms of XOR-AND graphs (XAGs). We define an XAG as
a logic network in which each gate corresponds to either an AND
or an XOR operator. Both regular and complemented edges can be
used to connect the gates, where a complemented edge indicates the
inversion of the signal. Fig. 1(a) shows an XAG representation of
the full adder, which uses two XOR gates, denoted by ⊕, and three
AND gates, denoted by ∧. Inversions are represented as dashed
lines. Previous works in logic synthesis have considered XOR-AND
logic networks, called XOR-AND Inverter Graphs (XAGs, [18]).
Even if our work and the one in [18] use the same data structure,
XAGs have been exploited to perform a different task. Indeed, the
work in [18] focuses on LUT mapping, and considers XOR and AND
gates to have the same cost.

A cut \( c \) of a node \( n \) in the logic network is a set of nodes, called
leaves, such that

- every path from node \( n \) to a primary input visits at least one
  leaf, and
- each leaf is contained in at least one path.

Node \( n \) is called the root of the cut and each cut represents a sub-
graph that includes the root \( n \) and some internal nodes, but has
the leaves as primary inputs. Fig. 1(b) shows in gray the subgraph
described by the cut for the output \( c_{\text{out}} \) with leaves \( a, b \) and \( c_{\text{in}} \). A cut is \( k \)-feasible (denoted here as \( k \)-cut), if it has at most \( k \) leaves.

All (or part of all) \( k \)-cuts of a logic network are found using cut
equation algorithms [19, 20].

As stated in the introduction, the multiplicative complexity of
a Boolean function is defined as the minimum number of AND
gates sufficient to implement it over the basis [AND, XOR, NOT] [4,
14]. More general, we also call the multiplicative complexity of a
logic network the actual number of AND gates to implement the
circuit [7].

2.2 Affine functions classification

This section reviews affine function classification, which is a strong
Boolean function classification technique based on affine opera-
tions.

Definition 2.1 (Affine operations [21]). The following set of five
affine operations on a Boolean function can be used to partition all
Boolean functions into equivalence classes [21].

1. Swapping two variables. From \( f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) \),
   one obtains \( g = f(x_1, \ldots, x_j, \ldots, x_i, \ldots, x_n) \) by swapping
   variables \( x_i \) and \( x_j \). We denote this operation as \( f \xrightarrow{\text{swap}} g \).
2. Complementing a variable. From \( f(x_1, \ldots, x_i, \ldots, x_n) \), one
   obtains \( g = f(x_1, \ldots, \neg x_i, \ldots, x_n) \) by complementing vari-
   able \( x_i \). We denote this operation as \( f \xrightarrow{\neg} g \).
3. Complementing the function. One obtains \( g = \bar{f} \) from \( f \) by
   complementing the whole function. We denote this operation as
   \( f \xrightarrow{\bar{}} g \).
4. Translational operation. One obtains \( g = f(x_1, \ldots, x_i \oplus
   x_j, \ldots, x_n) \) from \( f(x_1, \ldots, x_i, \ldots, x_n) \) by replacing \( x_i \) with
   \( x_i \oplus x_j \). We denote this operation as \( f \xrightarrow{x_i \oplus x_j} g \).
5. Disjoint translational operation. One obtains \( g = x_i \oplus f \) from
   \( f \) by XOR-ing it with input \( x_i \). We denote this operation as
   \( f \xrightarrow{x_i \oplus} \).

These operations partition all \( n \)-variable Boolean functions into
equivalence classes by means of the following equivalence relation.

Definition 2.2 (Affine equivalence [22]). We say that two \( n \)-variable
Boolean functions \( f \) and \( g \) are affine-equivalent, if there exist oper-
ations \( o_1, \ldots, o_k \) from Definition 2.1 such that

\[
f \xrightarrow{o_1 \ldots o_k} g.\]

One can readily verify that affine equivalence is an equivalence
relation. In the remainder, we write \( f \equiv g \), if \( f \) is affine-equivalent
to \( g \). Further, we refer to the equivalence class of \( f \) as \([ f ] = \{ g \mid f \equiv g \}\)

We can define one element of \([ f ]\) to be the representative func-
tion of that class. In an abuse of notation, we use \([ f ]\) both as the set of
all Boolean functions in the equivalence class, and also to denote
the representative itself. Note that \( f \equiv g \), if and only if \([ f ] = [ g ]\).

Example 2.3. We can show that \((x_1 x_2 x_3) = x_1 \land x_2\), where in this
case \( x_1 \land x_2 \) is considered a 3-variable Boolean function in which

\[
(x_1 x_2 x_3) \equiv (x_1 \land x_2) \in [ f ].
\]
Consider the full adder in Fig. 1(a), which has three gates corresponding to operations (i) an XOR gate, (ii) an inversion or (iii) a permutation of two inputs. All of them do not affect the number of AND gates in an XAG. Thus, to find the multiplicative complexity of a Boolean function, it is enough to know the multiplicative complexity of the representative of its equivalence class. In other words, each function can be written as an XAG using the same number of AND gates of its representative.

As the number of affine classes is orders of magnitudes smaller than the number of functions, a minimum circuit implementation over \{AND, XOR, NOT\} is known [4, 26] for each representative up to 6-input functions. In this scenario, minimum means minimum in terms of AND gates. The minimum XAG implementation of each Boolean function (up to 6-input) can thus be obtained by the XAG of its representative. This is obtained by adding XOR gates, inverters, and input permutation in accordance with the operations from Definition 2.1. As stated above, this will not influence the number of AND gates.

These two considerations allow us to optimize the number of AND gates of a Boolean function by (i) using the minimum XAG of the representative and (ii) augmenting it by the gates required for each transformation. In the following, we use the full adder from Fig. 1 as an example.

Example 3.1. Consider the full adder in Fig. 1(a), which has three AND gates. The objective is the minimization of such gates. Let us focus on the \(c_{\text{out}}\) output, which has the subgraph highlighted in Fig. 1(b). The subgraph implements the majority of three inputs \((a \oplus c_{\text{in}})\) which has truth table (in hexadecimal form) equal to \(0\times88\). The representative of the class is the function \(0\times88\), which is the AND gate represented in Fig. 2(a). As in Example 2.3, \(a \oplus b\) is considered a 3-variable Boolean function in which \(c_{\text{in}}\) is a don’t care input. This means that the full adder can be built using one AND gate together with some of the operations from Definition 2.1. The operations \(a_1, \ldots, a_k\) to transform a majority gate into a AND gate are the ones from Example 2.3: \(b, b \oplus c_{\text{in}}, a \oplus b, c_{\text{out}} \oplus a\). These add three XOR gates to the circuit in Fig. 2(a), and one inversion. The gates introduced are highlighted in Fig. 2(b). The final XAG of the full adder is shown in Fig. 2(c). We can conclude that the full adder has a multiplicative complexity of at most 1.

To sum up, we minimized the number of AND gates of a full adder by (i) using the minimum XAG of the representative (Fig. 2(a)) and (ii) by adding to it the gates corresponding to each operation (Fig. 2(b)).

4 OPTIMIZATION ALGORITHM

This section presents the optimization algorithm to reduce the multiplicative complexity of large (beyond 6-input) XAGs. It is based on the considerations and example shown in Section 3. First, we present the algorithm, then we give details on our implementation.

The algorithm is based on cut rewriting and is a general version of the DAG-aware AIG rewriting presented in [1]. The work in [1] aims at minimizing the AIG size by iteratively selecting AIG subgraphs and replacing them with smaller pre-computed subgraphs. Our algorithm implements a similar approach, based on cut enumeration [20]. The idea is to replace XAG subgraphs with new graphs which have smaller multiplicative complexity.

For each cut, the minimum representation in terms of AND gates can be computed as described in Example 3.1. Alg. 1 presents the

Algorithm 1 Cut-rewriting to minimize the number of AND gates (multiplicative complexity) of an XAG

**Input:** XAG of the cut \(c\) of node \(n\), DB_representative_to_xag

**Output:** Optimized XAG for cut \(c\)

1. \(f \leftarrow\) Boolean function of \(n\) with respect to the leaves
2. \(\text{representative} \leftarrow\) representative of the equivalence class of \(f\)
3. \(\text{operations} \leftarrow\) operations to go from \(f\) to \(\text{representative}\)
4. if \(\text{representative} \in\) DB_representative_to_xag then
5. \(\text{repr}\_\text{circuit} \leftarrow\) DB_representative_to_xag[\text{representative}]
6. else
7. \(\text{return} c\)
8. end if
9. \(\text{new}\_\text{cut}\_\text{circuit} \leftarrow\) repr\_circuit+ gates corresponding to operations on inputs and outputs
10. return new\_cut\_circuit
pseudo code. The minimum representations over the basis [AND, XOR, NOT] for all affine class representatives up to 6 inputs are used to create a database mapping all representatives up to 6 inputs to their minimum XAG representation. Further, as optimum results are known for functions with up to 6 inputs, the cut enumeration has been restricted to 6-cut. First, the Boolean function of the cut with respect to its leaves is computed. The work presented in [25] is then used to compute the affine class representative and the operations. The XAG of the representatives is retrieved from the database previously stored, and XOR gates, inverters and permutations according to the different operations are added in order to obtain the XAG implementing the correct function. Once the circuit for the cut is obtained, the algorithm continues as in [1]. In our case, the gain is evaluated considering the reduction in the number of AND gates.

4.1 Implementation details
The maximum number of leaves for each cut is equal to 6 (Alg. 1). Thus, as we are dealing with 6-input functions, we make use of truth tables to represent the Boolean functions. Truth tables for 6-input functions can be efficiently stored in computers as a single 64-bit unsigned integer, and are fast to compute. Further, our algorithm allows us to limit the maximum number of cuts computed for each node. In our experimental evaluation, we found that a cut limit of 12 leads to a good trade-off between runtime and quality.

The database stores the XAGs for each representative. In practice, this can be stored as one “large” XAG, called hereafter XAG_DB, XAG_DB has 6 inputs, and 147 998 outputs (this number is explained below). Each output is the XAG of one representative. The total size of this XAG is 2 339 563, XAG_DB is created once and can be reused for several rewriting calls. The database_to_xag function in Alg. 1 maps the truth table of each representative to its corresponding output in XAG_DB.

The work presented in [25] is used to calculate the affine representative and the required operations. The classification is performed by rearranging the coefficients of the function’s Rademacher-Walsh spectrum [21] based on their magnitudes. Depending on the distribution of coefficients, the number of iterations to reach the representative can vary significantly among different functions. In most cases, a representative is found very quickly, but for some functions this computation can be inefficient. We address this problem using two techniques. First, we maintain a cache of computed representatives and affine operations for all considered Boolean functions during rewriting. Therefore, no Boolean function needs to be classified twice. Also, we put an iteration limit on the classification routine, which causes us to omit some Boolean functions from rewriting. In our experiments, we consider 147 998 of all 150 357 affine equivalence classes.

5 EXPERIMENTS
In this section we evaluate the efficacy of the proposed algorithm. First, we compare our method to generic size optimization algorithms. Finally, we present results for benchmarks in the context of MPC and FHE.

We implemented the proposed algorithm into the open source logic synthesis framework mockturtle. All the experiments have been carried out on Intel Xeon E5-2680 CPU with 2.5 GHz and with 256 GB of main memory. The database containing the MC-optimum circuits for each representative of all 6-input functions fits into a compressed file of 12 MB. The limit on the number of cuts for each node has been set to 12, and we put an iteration limit on the classification routine to 100 000.

5.1 EPFL benchmarks
In this experiment, we demonstrate that our method decreases the number of AND gates when applied to benchmarks optimized using state-of-the-art generic size optimization.

We present our results on the EPFL benchmark suite [16], and we use the synthesis package ABC [15] as baseline for our comparison. In case of the EPFL benchmarks, our starting point are the best-known size-optimized 6-LUT benchmarks. As state-of-the-art size optimization, we apply a synthesis script that interleaves priority-cut-based 2-LUT mapping (if) [28], structural choices (dch and &synch2) [19, 29], and Boolean network optimization and resynthesis (&mfs) [30]. We apply the synthesis script

```
&st; &synch2; &if -m -a -K 2; &mfs -W 10;
&st; &dch; &if _m _a _K 2; &mfs -W 10
ten times, and we pick the final result as our baseline. The result is a 2-LUT network, i.e., a logic network in which each gate corresponds to an arbitrary 2-input function. Note that a 2-LUT network can be directly translated into an XAG without increasing the number of gates by choosing inverters appropriately. Therefore, it provides us with a good starting point, despite the fact that it uses a unit cost model that accounts the same cost for both AND and XOR gates.

The results are shown in Table 1. The initial benchmarks are generated as previously discussed. The “One round” results are obtained by applying one iteration of our proposed method, while the “Repeat until convergence” results show the number of AND and XOR gates after more iterations of our algorithm. In this last case, the algorithm is run until no further improvement is obtained. A (−) entry indicates that no improvement was possible even with applying a single iteration of our proposed method. On average, 15 iterations are needed before convergence. The maximum number of iterations encountered by our tool is equal to 58 (multiplier benchmark). The experiments show that the number of AND gates reaches a local minimum for all benchmarks, and the normalized geometric mean decreases both for arithmetic and random-control benchmarks. The total improvement is shown in the last column of both the “One round” and “Repeat until convergence” results. On average, we decrease the number of AND gates of 34%. The arithmetic benchmarks benefit more from our method and are optimized up to 77% in the number of AND gates. On the contrary, the random-control benchmarks are optimized 23% on average.

Note that we do not consider any XOR optimization in this work. An algorithm to minimize the number of XOR for cryptography applications can be found in [14].

1Available at: https://github.com/usnistgov/Circuits/tree/master/slp
2Available at: https://github.com/lsils/benchmarks
3See version v2018.1 on https://github.com/eletesta/dac19-experiments
Table 1: Experimental results for EPFL benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Initial</th>
<th>One round</th>
<th>Repeat until convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AND</td>
<td>XOR</td>
<td>AND</td>
<td>time [s]</td>
<td>impr.</td>
</tr>
<tr>
<td>Adder</td>
<td>256</td>
<td>129</td>
<td>550</td>
<td>255</td>
<td>318 529 3.74 42 %</td>
</tr>
<tr>
<td>Barrel shifter</td>
<td>135</td>
<td>128</td>
<td>2688</td>
<td>0</td>
<td>896 1728 15.41 67 %</td>
</tr>
<tr>
<td>Divisor</td>
<td>128</td>
<td>128</td>
<td>12001</td>
<td>3897</td>
<td>6378 8779 100.83 47 %</td>
</tr>
<tr>
<td>Log2</td>
<td>32</td>
<td>32</td>
<td>24941</td>
<td>3592</td>
<td>19942 8583 327.34 20 %</td>
</tr>
<tr>
<td>Max</td>
<td>512</td>
<td>130</td>
<td>2687</td>
<td>0</td>
<td>1471 1387 17.36 45 %</td>
</tr>
<tr>
<td>Multiplier</td>
<td>128</td>
<td>128</td>
<td>16119</td>
<td>4301</td>
<td>12209 8122 169.97 24 %</td>
</tr>
<tr>
<td>Sine</td>
<td>24</td>
<td>25</td>
<td>4937</td>
<td>519</td>
<td>4194 1572 56.76 15 %</td>
</tr>
<tr>
<td>Square</td>
<td>64</td>
<td>128</td>
<td>9225</td>
<td>3850</td>
<td>5323 7984 34.34 42 %</td>
</tr>
<tr>
<td></td>
<td><strong>AND</strong></td>
<td><strong>XOR</strong></td>
<td><strong>AND</strong></td>
<td><strong>time [s]</strong></td>
<td><strong>impr.</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17.85 0 %</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.83 70 %</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.6 0 %</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.62 0 %</td>
</tr>
<tr>
<td></td>
<td><strong>AND</strong></td>
<td><strong>XOR</strong></td>
<td><strong>AND</strong></td>
<td><strong>time [s]</strong></td>
<td><strong>impr.</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3.65 19 %</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3.9 14 %</td>
</tr>
<tr>
<td></td>
<td><strong>AND</strong></td>
<td><strong>XOR</strong></td>
<td><strong>AND</strong></td>
<td><strong>time [s]</strong></td>
<td><strong>impr.</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.83 70 %</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.62 0 %</td>
</tr>
<tr>
<td></td>
<td><strong>AND</strong></td>
<td><strong>XOR</strong></td>
<td><strong>AND</strong></td>
<td><strong>time [s]</strong></td>
<td><strong>impr.</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.62 0 %</td>
</tr>
</tbody>
</table>

Normalized geometric mean (arithmetic) 1 0.60 0.49

Round-robin arbiter

<table>
<thead>
<tr>
<th>Name</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Initial</th>
<th>One round</th>
<th>Repeat until convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AND</td>
<td>XOR</td>
<td>AND</td>
<td>time [s]</td>
<td>impr.</td>
</tr>
<tr>
<td>AES (No Key Expansion)</td>
<td>256</td>
<td>128</td>
<td>6800</td>
<td>25124</td>
<td>6800 25124 37.48 0 %</td>
</tr>
<tr>
<td>AES (Key Expansion)</td>
<td>1536</td>
<td>128</td>
<td>5440</td>
<td>20325</td>
<td>5440 20325 27.32 0 %</td>
</tr>
<tr>
<td>DES (No Key Expansion)</td>
<td>128</td>
<td>64</td>
<td>18124</td>
<td>1337</td>
<td>17404 4096 251.57 4 %</td>
</tr>
<tr>
<td>DES (Key Expansion)</td>
<td>832</td>
<td>64</td>
<td>18175</td>
<td>1348</td>
<td>17403 4168 256.69 4 %</td>
</tr>
<tr>
<td>MD5</td>
<td>512</td>
<td>128</td>
<td>29084</td>
<td>14133</td>
<td>12300 29270 101.53 58 %</td>
</tr>
<tr>
<td>SHA-1</td>
<td>512</td>
<td>160</td>
<td>37172</td>
<td>24166</td>
<td>17141 42415 114.55 54 %</td>
</tr>
<tr>
<td>SHA-256</td>
<td>512</td>
<td>256</td>
<td>89478</td>
<td>42024</td>
<td>52921 86304 311.68 41 %</td>
</tr>
<tr>
<td>32-bit Adder</td>
<td>64</td>
<td>33</td>
<td>127</td>
<td>61</td>
<td>38 146 0.83 70 %</td>
</tr>
<tr>
<td>64-bit Adder</td>
<td>128</td>
<td>65</td>
<td>265</td>
<td>115</td>
<td>100 260 2.06 62 %</td>
</tr>
<tr>
<td>32x32-bit Multiplier</td>
<td>64</td>
<td>64</td>
<td>5926</td>
<td>1069</td>
<td>4290 2351 57.19 28 %</td>
</tr>
<tr>
<td>Comp. 32-bit Signed LTEQ</td>
<td>64</td>
<td>1</td>
<td>150</td>
<td>0</td>
<td>121 69 3.65 19 %</td>
</tr>
<tr>
<td>Comp. 32-bit Signed LT</td>
<td>64</td>
<td>1</td>
<td>150</td>
<td>0</td>
<td>121 69 3.65 19 %</td>
</tr>
<tr>
<td>Comp. 32-bit Unsigned LTEQ</td>
<td>64</td>
<td>1</td>
<td>150</td>
<td>0</td>
<td>121 69 3.65 19 %</td>
</tr>
<tr>
<td>Comp. 32-bit Unsigned LT</td>
<td>64</td>
<td>1</td>
<td>150</td>
<td>0</td>
<td>121 69 3.65 19 %</td>
</tr>
</tbody>
</table>

Normalized geometric mean (random-control) 1 0.90 0.87

Table 2: Experimental results for MPC and FHE benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Initial</th>
<th>One round</th>
<th>Repeat until convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AND</td>
<td>XOR</td>
<td>AND</td>
<td>time [s]</td>
<td>impr.</td>
</tr>
<tr>
<td>AES (No Key Expansion)</td>
<td>256</td>
<td>128</td>
<td>6800</td>
<td>25124</td>
<td>6800 25124 37.48 0 %</td>
</tr>
<tr>
<td>AES (Key Expansion)</td>
<td>1536</td>
<td>128</td>
<td>5440</td>
<td>20325</td>
<td>5440 20325 27.32 0 %</td>
</tr>
<tr>
<td>DES (No Key Expansion)</td>
<td>128</td>
<td>64</td>
<td>18124</td>
<td>1337</td>
<td>17404 4096 251.57 4 %</td>
</tr>
<tr>
<td>DES (Key Expansion)</td>
<td>832</td>
<td>64</td>
<td>18175</td>
<td>1348</td>
<td>17403 4168 256.69 4 %</td>
</tr>
<tr>
<td>MD5</td>
<td>512</td>
<td>128</td>
<td>29084</td>
<td>14133</td>
<td>12300 29270 101.53 58 %</td>
</tr>
<tr>
<td>SHA-1</td>
<td>512</td>
<td>160</td>
<td>37172</td>
<td>24166</td>
<td>17141 42415 114.55 54 %</td>
</tr>
<tr>
<td>SHA-256</td>
<td>512</td>
<td>256</td>
<td>89478</td>
<td>42024</td>
<td>52921 86304 311.68 41 %</td>
</tr>
<tr>
<td>32-bit Adder</td>
<td>64</td>
<td>33</td>
<td>127</td>
<td>61</td>
<td>38 146 0.83 70 %</td>
</tr>
<tr>
<td>64-bit Adder</td>
<td>128</td>
<td>65</td>
<td>265</td>
<td>115</td>
<td>100 260 2.06 62 %</td>
</tr>
<tr>
<td>32x32-bit Multiplier</td>
<td>64</td>
<td>64</td>
<td>5926</td>
<td>1069</td>
<td>4290 2351 57.19 28 %</td>
</tr>
<tr>
<td>Comp. 32-bit Signed LTEQ</td>
<td>64</td>
<td>1</td>
<td>150</td>
<td>0</td>
<td>121 69 3.65 19 %</td>
</tr>
<tr>
<td>Comp. 32-bit Signed LT</td>
<td>64</td>
<td>1</td>
<td>150</td>
<td>0</td>
<td>121 69 3.65 19 %</td>
</tr>
<tr>
<td>Comp. 32-bit Unsigned LTEQ</td>
<td>64</td>
<td>1</td>
<td>150</td>
<td>0</td>
<td>121 69 3.65 19 %</td>
</tr>
<tr>
<td>Comp. 32-bit Unsigned LT</td>
<td>64</td>
<td>1</td>
<td>150</td>
<td>0</td>
<td>121 69 3.65 19 %</td>
</tr>
</tbody>
</table>

Normalized geometric mean 1 0.68 0.56
5.2 MPC and FHE benchmarks for cryptographic applications

In this section, we demonstrate our approach in the context of MPC and FHE, by optimizing the number of AND gates for best-known reported benchmarks. Both these cryptographic applications benefit from AND gates minimization. XOR gates and inverters are for free, while AND gates are considered more expensive in both cases [8].

The results are shown in Table 2. As in the previous case, we distinguish between “One round” results and “Repeat until convergence.” The first four benchmarks are block ciphers, followed by three hash functions, and seven arithmetic functions. Of most interest is the improvements in the block ciphers and hash functions. No improvement is possible with our technique in both variants of the AES block cipher, which indicates that the reported number of AND gates may be close to the multiplicative complexity of the function. An improvement of 17% was possible in the case of the DES cipher, while a much larger improvement was possible for all three hash functions, with more than 66% improvement after repeating the proposed approach until convergence.

It is worth noticing that our method optimizes the number of AND gates needed to implement the 32-bit adder down to 32, which is known to be optimum [31]. The same applies for the 64-bit case.

6 CONCLUSION

We have proposed an algorithm to reduce the number of AND gates in an XAG, which is a logic network composed of AND, XOR, and inverter gates. Such XAG optimization plays a central role in cryptography applications such as fully homomorphic encryption and multi-party computation. For both these applications, XOR gates and inverters are for free, while AND gates are considered slower and more expensive. We presented a fully automatic algorithm based on cut rewriting. It optimizes the number of AND gates by exploiting affine classification together with cut enumeration to explore optimization potential in local subcircuits. Our experiments show that we can reduce the number of AND gates by 34% on average when compared to generic size optimization. We also demonstrate improvement in best-known benchmarks for MPC and FHE applications.

ACKNOWLEDGMENTS

We wish to thank Alan Mishchenko, Tim Güneysu, and René Peralta and his team at NIST for fruitful discussions. This research was supported by the EPFL Open Science Fund, by the Swiss National Science Foundation (200021-169084 MAJesty) and by the ERC project H2020-ERC-2014-ADG 669354 CyberCare.

REFERENCES


