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

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

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

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

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:

E. Testa, M. Soeken, L. Amaru, and G. De Micheli. 2019. Reducing the Multiplicative Complexity in Logic Networks for Cryptography and Security Applications. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317893

1 INTRODUCTION

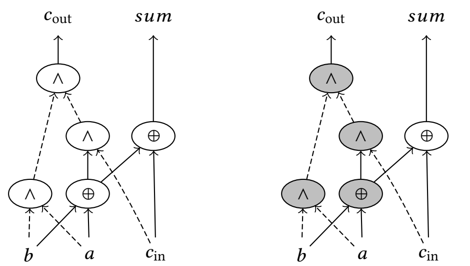
Logic synthesis is considered one of the fundamental steps in the realization of competitive and leading-edge integrated circuits. In the last decades, both heuristics and exact methods have been proposed, together with new data structures, for the abstraction and manipulation of Boolean circuits. Classical 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.

One logic basis widely used to represent circuits in cryptography is given by the gates {AND, XOR, NOT}. This is because the XOR and AND operations in this basis are equivalent to addition and multiplication in GF(2), respectively [3]. Logic synthesis for cryptographic applications addresses the minimization of the number of AND gates in a logic network composed of AND, XOR, and inverter gates. Indeed, the number of AND gates is an indicator of the degree of vulnerability of the circuit [4]. The minimum number of AND gates sufficient to implement a Boolean function over the basis {AND, XOR, NOT} is called *multiplicative complexity of the function* [4, 5]. Lower multiplicative complexity of a function corresponds to higher vulnerability to algebraic attacks [4, 6]. Moreover, the number of AND gates in a function representation, called here *multiplicative complexity of the circuit* [7], may not correspond to the multiplicative complexity of the function itself, but only provides an upper bound thereof. Therefore, the minimization of the number of AND gates in a circuit is important in order to assess the real multiplicative complexity of the function, and consequently its resistance against cryptanalysis attacks.

The minimization of 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 *multi-party 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 porpose 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

978-1-7281-2425-4/19/$31.00 ©2019 IEEE



(a) XAG implementation of the full adder

(b) One cut of the full adder highlighted in gray

Figure 1: XAG representation of the full adder. The signal c_{out} in the output carry. Dashed lines represent complemented edges.

method achieves lower numbers of AND gates for the EPFL combinational benchmark suite [16]. On average, our proposed method reduces the number of AND gates by 34%. Moreover, we demonstrate 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 applications 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 $\oplus$, and three AND gates, denoted by $\wedge$. Inversions are represented as dashed lines. Previous works in logic synthesis have considered XOR-AND logic networks, called XOR-AND Inverter Graphs (XAIGs, [18]). Even if our work and the one in [18] use the same data structure, XAIGs 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 subgraph 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_{out} with leaves a, b and c_{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 enumeration 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 operations.

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 $f(x_1, \dots, x_i, \dots, x_j, \dots, x_n)$, one obtains $g = f(x_1, \dots, x_j, \dots, x_i, \dots, x_n)$ by swapping variables x_i and x_j. We denote this operation as $f \xrightarrow{x_i \leftrightarrow x_j} g$.
2. *Complementing a variable.* From $f(x_1, \dots, x_i, \dots, x_n)$, one obtains $g = f(x_1, \dots, \bar{x}_i, \dots, x_n)$ by complementing variable x_i. We denote this operation as $f \xrightarrow{\bar{x}_i} g$.
3. *Complementing the function.* One obtains $g = \bar{f}$ from f by complementing the whole function. We denote this operation as $f \xrightarrow{\neg} g$.
4. *Translational operation.* One obtains $g = f(x_1, \dots, x_i \oplus x_j, \dots, x_n)$ from $f(x_1, \dots, x_i, \dots, x_n)$ 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 $f \xrightarrow{\oplus x_i} g$.

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

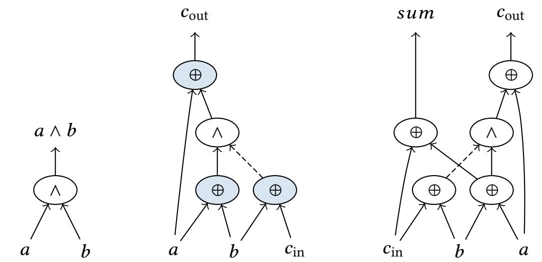
Definition 2.2 (Affine equivalence [22]). We say that two n-variable Boolean functions f and g are *affine-equivalent*, if there exist operations $o_1, \dots, o_k$ from Definition 2.1 such that

$$f \xrightarrow{o_1} \cdots \xrightarrow{o_k} g.$$

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

We can define one element of $[f]$ to be the *representative function* 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 $f \doteq g$, if and only if $[f] = [g]$.

Example 2.3. We can show that $\langle x_1 x_2 x_3 \rangle \doteq x_1 \wedge x_2$, where in this case $x_1 \wedge x_2$ is considered a 3-variable Boolean function in which



(a) Circuit of the representative **(b) Circuit of the representative and operations** **(c) Final full adder XAG**

Figure 2: Method overview example

x_3 is a don't care input.

$$x_1 \wedge x_2 \xrightarrow{\bar{x}_2} x_1 \wedge \bar{x}_2 \xrightarrow{x_2 \oplus x_3} x_1 \wedge (\bar{x}_2 \oplus x_3) \xrightarrow{x_1 \oplus x_2}$$
$$(x_1 \oplus x_2) \wedge (\bar{x}_2 \oplus x_3) = x_1\bar{x}_2 \oplus x_1x_3 \oplus x_2x_3 \xrightarrow{\oplus x_1}$$
$$x_1 \oplus x_1\bar{x}_2 \oplus x_1x_3 \oplus x_2x_3 = x_1x_2 \oplus x_1x_3 \oplus x_2x_3 = \langle x_1x_2x_3 \rangle$$

Using this equivalence relation the set of all n-variable Boolean functions for $n = 1, 2, 3, 4, 5, 6$ collapses into just 1, 2, 3, 8, 48, 150 357 equivalence classes [4, 23], respectively. An algorithm to compute the representative of each class and the set of operations $o_1 \ldots o_k$ has been recently proposed in [24, 25].

3 PROPOSED METHOD OVERVIEW

The goal of this work is to minimize the number of AND gates in an XAG, as AND gates minimization is a crucial aspect in many cryptography applications. This section presents the general idea and illustrates the proposed method using an example. The optimization algorithm and details on the implementation are given in Section 4.

Our optimization method is based on two major considerations:

(1) The multiplicative complexity of a function is invariant under affine operations. Each affine operation can be realized by (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.

(2) 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 transformations. In the following, we use the full adder from Fig. 1 as an example.

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

- f ←Boolean function of n with respect to the leaves
- *representative* ← representative of the equivalence class of f
- *operations* ← operations to go from f to *representative*
- **if** *representative* ∈ *DB_representative_to_xag* **then**
  - *repr_circuit* ← *DB_representative_to_xag*[*representative*]
- **else**
  - **return** c
- **end if**
- *new_cut_circuit* ← *repr_circuit*+ gates corresponding to operations on inputs and outputs
- **return** *new_cut_circuit*

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_{out} output, which has the subgraph highlighted in Fig. 1(b). The subgraph implements the majority of three inputs $\langle abc_{\text{in}} \rangle$ which has truth table (in hexadecimal form) equal to $0xe8$. The representative of the class is the function $0x88$, which is the AND gate represented in Fig. 2(a). As in Example 2.3, $a \wedge b$ is considered a 3-variable Boolean function in which c_{in} is a don't care input. This means that the full adder can be build using one AND gate together with some of the operations from Definition 2.1. The operations $o_1 \ldots o_k$ to transform a majority gate into a AND gate are the ones from Example 2.3: $\bar{b}$, $b \oplus c_{\text{in}}$, $a \oplus b$, $c_{\text{out}} \oplus a$. 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 in 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 term of AND gates can be computed as described in Example 3.1. Alg. 1 presents the pseudo code. The minimum representations over the basis {AND, XOR, NOT} for all affine class representatives up to 6 inputs[1] 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*.[2] 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.[3] 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 crypthography applications can be found in [14].

[1] Available at: https://github.com/usnistgov/Circuits/tree/master/slp

[2] Available at: https://github.com/lsils/mockturtle [27]. A python script for the experimental evaluation is also available at: https://github.com/eletesta/dac19-experiments

[3] See version v2018.1 on https://github.com/lsils/benchmarks

Table 1: Experimental results for EPFL benchmarks

Name	Inputs	Outputs	Initial		One round				Repeat until convergence			
			AND	XOR	AND	XOR	time [s]	impr.	AND	XOR	time [s]	impr.
Adder	256	129	550	255	318	529	3.74	42 %	128	549	5.36	77 %
Barrel shifter	135	128	2688	0	896	1728	15.41	67 %	832	1728	16.65	69 %
Divisor	128	128	12001	3897	6378	8779	100.83	47 %	6060	8994	1132.23	50 %
Log2	32	32	24941	3592	19942	8583	327.34	20 %	19436	9371	11988.6	22 %
Max	512	130	2687	0	1471	1387	17.36	45 %	931	1479	81.82	65 %
Multiplier	128	128	16119	4301	12209	8122	169.97	24 %	11940	8614	9202.11	26 %
Sine	24	25	4937	519	4194	1572	56.76	15 %	4075	1770	405.47	17 %
Square-root	128	64	12336	3746	7101	9122	103.35	42 %	6244	9640	418.98	49 %
Square	64	128	9225	3850	5323	7984	34.34	42 %	5181	8084	158.92	44 %
Normalized geometric mean (arithmetic)			1		**0.60**				**0.49**			
Round-robin arbiter	256	129	1181	0	1181	0	17.85	0 %		—//—		0 %
Alu control unit	7	26	86	2	85	8	0.7	1 %	85	8	1.22	1 %
Coding-cavlc	10	11	536	16	507	152	10.05	5 %	494	197	23.86	8 %
Decoder	8	256	341	0	341	0	0	0 %		—//—		0 %
i2c controller	147	142	823	15	659	342	17.22	20 %	623	502	109.12	24 %
int to float converter	11	7	133	13	112	76	1.82	16 %	100	101	4.66	25 %
Memory controller	1204	1231	7418	361	5393	3165	89.62	27 %	5113	4168	2592.97	31 %
Priority encoder	128	8	368	0	327	158	4.12	11 %	327	158	8.1	11 %
Lookahead XY router	60	30	96	0	96	0	1.6	0 %		—//—		0 %
Voter	1001	1	7308	1833	6046	4917	55.74	17 %	5651	6066	262.21	23 %
Normalized geometric mean (random-control)			1		**0.90**				**0.87**			

Table 2: Experimental results for MPC and FHE benchmarks

Name	Inputs	Outputs	Initial		One round				Repeat until convergence			
			AND	XOR	AND	XOR	time [s]	impr.	AND	XOR	time [s]	impr.
AES (No Key Expansion)	256	128	6800	25124	6800	25124	37.48	0 %		—//—		0 %
AES (Key Expansion)	1536	128	5440	20325	5440	20325	27.32	0 %		—//—		0 %
DES (No Key Expansion)	128	64	18124	1337	17404	4096	251.57	4 %	15093	11105	8876.11	17 %
DES (Key Expansion)	832	64	18175	1348	17403	4168	256.69	4 %	15126	11263	9262.73	17 %
MD5	512	128	29084	14133	12300	29270	101.53	58 %	9381	30325	145.44	68 %
SHA-1	512	160	37172	24166	17141	42415	114.55	54 %	11820	44311	293.8	68 %
SHA-256	512	256	89478	42024	52921	86304	311.68	41 %	30201	91278	12562.8	66 %
32-bit Adder	64	33	127	61	38	146	0.83	70 %	32	150	0.98	75 %
64-bit Adder	128	65	265	115	100	260	2.06	62 %	64	284	2.61	76 %
32x32-bit Multiplier	64	64	5926	1069	4290	2351	57.19	28 %	4107	2473	135.02	31 %
Comp. 32-bit Signed LTEQ	64	1	150	0	121	69	3.65	19 %	114	89	6.3	24 %
Comp. 32-bit Signed LT	64	1	150	0	129	74	3.9	14 %	108	116	10.17	28 %
Comp. 32-bit Unsigned LTEQ	64	1	150	0	121	69	3.23	19 %	114	89	6.38	24 %
Comp. 32-bit Unsigned LT	64	1	150	0	129	74	4.04	14 %	108	116	10.59	28 %
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.[4] 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 the 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

[1] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.

[2] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

[3] J. Boyar and R. Peralta, "A small depth-16 circuit for the AES S-Box," in *International Information Security Conference*, 2012, pp. 287–298.

[4] M. Turan Sönmez and R. Peralta, "The multiplicative complexity of Boolean functions on four and five variables," in *Lightweight Cryptography for Security and Privacy*, Cham, 2015, pp. 21–33.

[5] J. Boyar, R. Peralta, and D. Pochuev, "On the multiplicative complexity of Boolean functions over the basis $(\wedge, \oplus, 1)$," *Theoretical Computer Science*, vol. 235, no. 1, pp. 43–57, 2000.

[6] N. Courtois, D. Hulme, and T. Mourouzis, "Solving circuit optimisation problems in cryptography and cryptanalysis," *IACR Cryptology ePrint Archive*, vol. 2011, p. 475, 2011.

[7] D. Goudarzi and M. Rivain, "On the multiplicative complexity of Boolean functions and bitsliced higher-order masking," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2016, pp. 457–478.

[8] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for MPC and FHE," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2015, pp. 430–454.

[9] 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 *Conference on Computer and Communications Security*, 2017, pp. 1825–1842.

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

[11] A. C.-C. Yao, "How to generate and exchange secrets," in *Annual Symposium on Foundations of Computer Science*, 1986, pp. 162–167.

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

[13] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *International Colloquium on Automata, Languages, and Programming*, 2008, pp. 486–498.

[14] 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.

[15] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[16] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Int'l Workshop on Logic and Synthesis*, 2015.

[17] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference*, 2014, pp. 194:1–194:6.

[18] I. Háleček, P. Fišer, and J. Schmidt, "Are XORs in logic synthesis really necessary?" in *Design and Diagnostics of Electronic Circuits & Systems, International Symposium on*, 2017, pp. 134–139.

[19] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 240–253, 2007.

[20] P. Pan and C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *Int'l Symp. on Field Programmable Gate Arrays*, 1998, pp. 35–42.

[21] C. R. Edwards, "The application of the Rademacher-Walsh transform to Boolean function classification and threshold logic synthesis," *IEEE Trans. on Computers*, vol. 24, no. 1, pp. 48–62, 1975.

[22] R. J. Lechner, "Harmonic analysis of switching functions," in *Recent Developments in Switching Theory*, A. Mukhopadhyay, Ed. Academic Press, 1971, pp. 121–228.

[23] E. R. Berlekamp and L. R. Welch, "Weight distributions of the cosets of the (32, 6) Reed-Muller code," *IEEE Trans. on Information Theory*, vol. 18, no. 1, pp. 203–207, 1972.

[24] D. M. Miller and M. Soeken, "A spectral algorithm for ternary function classification," in *Int'l Symp. on Multiple-Valued Logic*, 2018, pp. 198–203.

[25] M. Miller and M. Soeken, "An algorithm for linear, affine and spectral classification of Boolean functions," *International Workshop on Boolean Problems*, pp. 237–254, 2018.

[26] C. Calik, M. S. Turan, and R. Peralta, "The multiplicative complexity of 6-variable Boolean functions," Cryptology ePrint Archive, Report 2018/002, 2018.

[27] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL logic synthesis libraries," May 2018, arXiv:1805.05121.

[28] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, "Combinational and sequential mapping with priority cuts," in *Int'l Conf. on Computer-Aided Design*, 2007, pp. 354–361.

[29] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2894–2903, 2006.

[30] A. Mishchenko, R. K. Brayton, J. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 34:1–34:23, 2011.

[31] J. Boyar and R. Peralta, "Tight bounds for the multiplicative complexity of symmetric functions," *Theoretical Computer Science*, vol. 396, no. 1–3, pp. 223–246, 2008.

[4] Available at: https://homes.esat.kuleuven.be/~nsmart/MPC/