RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers

Alireza Mahzoon[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Germany [2]DFKI GmbH, Bremen, Germany
{mahzoon,grosse,drechsle}@informatik.uni-bremen.de

ABSTRACT

In recent years, formal methods based on Symbolic Computer Algebra (SCA) have shown very good results in verification of integer multipliers. The success is based on removing redundant terms (vanishing monomials) early which allows to avoid the explosion in the number of monomials during backward rewriting. However, the SCA approaches still suffer from two major problems: (1) high dependence on the detection of Half Adders (HAs) realized as AND-XOR gates in the multiplier netlist, and (2) extremely large search space for finding the source of the vanishing monomials. As a consequence, if the multiplier consists of dirty logic, i.e. for instance using non-standard libraries or logic optimization, the existing SCA methods are completely blind on the resulting polynomials, and their techniques for effective division fail.

In this paper, we present RevSCA. RevSCA brings back light into backward rewriting by identifying the atomic blocks of the arithmetic circuits using dedicated reverse engineering techniques. Our approach takes advantage of these atomic blocks to detect all sources of vanishing monomials independent of the design architecture. Furthermore, it cuts the local vanishing removal time drastically due to limiting the search space to a small part of the design only. Experimental results confirm the efficiency of our approach in verification of a wide variety of integer multipliers with up to 1024 output bits.

1 INTRODUCTION

Multiplication is one of the most frequent and vital operations in many digital applications. Particularly, in computational-intensive applications such as signal processing and cryptography a large part of the chip is dedicated to multiplier circuitry in order to perform multiplication fast and efficient. Since the invention of the first integer multiplier, the demands for fast and area-efficient designs have encouraged designers to implement a wide variety of multiplier architectures. Most of these architectures take advantage of complex algorithms to shorten the critical path, to reduce the number of the wires, or to minimize the number of building blocks. As a consequence, rigorous verification is inevitable to ensure the correctness of the multiplier.

While the utilization of large and non-trivial multipliers is becoming more popular in industry, formal verification techniques for multipliers still suffer from many limitations: *Decision Diagrams* (such as BDDs and *BMDs), *Boolean Satisfiability* (SAT), and *Satisfiability Modulo Theories* (SMT) are facing scalability issues and cannot verify large designs; reverse engineering approaches (e.g. [16]) only support architecturally simple multipliers; and term rewriting techniques (e.g. [17]) are not fully automated.

Recently, *Symbolic Computer Algebra* (SCA) approaches have overcome many limitations of the just mentioned methods, see for instance [5, 18, 19, 11, 12, 7]. The general idea of SCA-based verification is to (1) represent the function of the multiplier based on its inputs and outputs as a *Specification Polynomial SP*, (2) capture the logical gates of the circuit also as a set of polynomials P_G, and (3) take advantage of Gröbner basis theory in order to prove the membership of SP in the ideal generated by P_G. The just mentioned 3rd step consists of the step-wise division of SP by P_G known as *backward rewriting*, and eventually the evaluation of the resulting remainder. If this remainder is zero, the multiplier is correct; otherwise, it is buggy.

The SCA-based verification techniques are scalable in proving the correctness of trivial multipliers. In contrast, for non-trivial multipliers the number of monomials always explodes during backward rewriting until recently – it has been shown that this explosion is caused by redundant monomials, known as vanishing monomials in literature [13, 14, 11]. Recently, in [8] a new theory for the source of vanishing monomials has been introduced. The vanishing monomials are formed when substituting a converging gate during backward rewriting, i.e. a gate where both outputs of a *Half Adder* (HA) converge. Based on these converging gates, [8] also proposed a local backward rewriting step to make the global backward rewriting vanishing-free. However, this solution still suffers from two major problems: (1) High dependency on the detection of HAs realized as AND-XOR pairs in the netlist, and (2) extremely large search space for finding the source of vanshing monomials, i.e. the converging gates. As a result, if the multiplier consists of dirty logic, i.e. for instance using non-standard libraries or logic optimization, the SCA methods are completely blind on the resulting polynomials, and their techniques for effective division fail.

Contribution: In this paper, we introduce *RevSCA*. RevSCA brings back light into backward rewriting by identifying *atomic blocks* of the multiplier, i.e. HAs, *Full Adders* (FAs) and *Compressors* (CMs), using dedicated reverse engineering techniques. This allows to resolve both problems, since based on the atomic blocks (a) the local vanishing removal phase becomes robust against design alterations, (b) more compact polynomials for the atomic blocks can be created, and finally (c) the search space to find the converging gates (and by this the sources of vanishing monomials) can be reduced to a significantly smaller fraction of gates.

The paper is structured as follows: Section 2 reviews related work and Section 3 provides the preliminaries. Then, in Section 4 we showcase the advantages of knowing the atomic blocks for SCA verification. In Section 5 we introduce RevSCA. We describe in detail how we perform reverse engineering on the *AND-Inverter Graph* (AIG) representation of a multiplier which allows to detect all sources of vanishing monomials independent of the design architecture and hence make backward rewriting feasible. The experimental results show that our proposed method can verify a large variety of (dirty) integer multipliers with 1024 output bits while the other state-of-the-art methods fail.[1]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317898

[1]Our tool RevSCA and all benchmarks are available on GitHub; links can be found at http://www.sca-verification.org/revsca

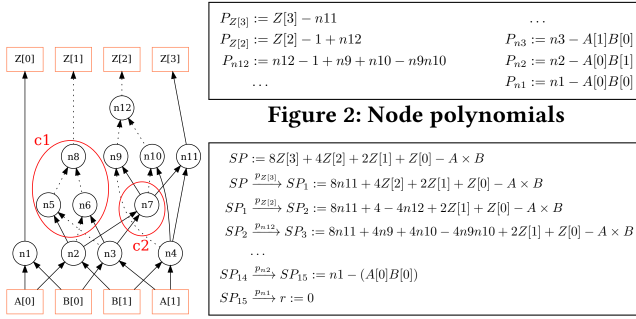


Figure 1: 2×2 mult

$$P_{Z[3]} := Z[3] - n11 \qquad \dots$$
$$P_{Z[2]} := Z[2] - 1 + n12 \qquad P_{n3} := n3 - A[1]B[0]$$
$$P_{n12} := n12 - 1 + n9 + n10 - n9n10 \qquad P_{n2} := n2 - A[0]B[1]$$
$$\dots \qquad P_{n1} := n1 - A[0]B[0]$$

Figure 2: Node polynomials

$$SP := 8Z[3] + 4Z[2] + 2Z[1] + Z[0] - A \times B$$
$$SP \xrightarrow{p_{Z[3]}} SP_1 := 8n11 + 4Z[2] + 2Z[1] + Z[0] - A \times B$$
$$SP_1 \xrightarrow{p_{Z[2]}} SP_2 := 8n11 + 4 - 4n12 + 2Z[1] + Z[0] - A \times B$$
$$SP_2 \xrightarrow{p_{n12}} SP_3 := 8n11 + 4n9 + 4n10 - 4n9n10 + 2Z[1] + Z[0] - A \times B$$
$$\dots$$
$$SP_{14} \xrightarrow{p_{n2}} SP_{15} := n1 - (A[0]B[0])$$
$$SP_{15} \xrightarrow{p_{n1}} r := 0$$

Figure 3: Backward rewriting steps

2 RELATED WORK

Several SCA-based verification methods have been proposed to verify integer multipliers. However, most of these methods only work for trivial multipliers. The authors of [18] introduced step-wise backward rewriting based on the reverse topological order of the circuit. The technique proposed in [5] identifies fanout-free cones before backward rewriting to reduce the total number of generated monomials. The method of [11] uses a column-wise rewriting which cuts the circuit into slices and verifies them incrementally. Techniques to identify HAs and FAs in the design, which can be viewed as a simple form of reverse engineering, have been introduced in [19, 12] in order to speed up the verification. Overall, all these techniques are only applicable to multipliers where 2nd and 3rd stages are fully made of HAs and FAs.

The goal of some approaches is to alleviate the vanishing monomials problem and make the verification of non-trivial multipliers possible. The authors of [13] proposed the XOR rewriting technique to group gates into cones based on the XOR gates. Then, the polynomials for the cones are extracted and vanishing monomials are removed. But, this method is not robust and cannot remove all vanishing monomials for many non-trivial multipliers.

[8] presented a theory for the origin of vanishing monomials in non-trivial multipliers. It introduced a local vanishing removal phase before (global) backward rewriting to remove the vanishing monomials in converging cones starting from HAs. However, this method is highly dependent on the detection of HAs realized as AND-XOR pairs. Moreover, the search space for finding the source of vanishing monomials on the netlist is extremely large.

3 PRELIMINARIES

In this section, first an overview of AIGs is given. Then, SCA verification using AIGs is reviewed.

3.1 And-Inverter Graph

An *AND-Inverter Graph* (AIG) is a directed, acyclic graph that represents the functionality of a circuit using two-input AND nodes and positive and negative edges. Since the operators $\{\wedge, \neg\}$ are functionally complete, any Boolean function can be represented in an AIG. Furthermore, the combinational logic of an arbitrary Boolean network can be easily transformed into an AIG using DeMorgan’s rule [9]. Fig. 1 shows the AIG of a 2×2 multiplier. Note that the dashed lines show the negated edges (i.e. NOT operators).

AIGs are widely used in logic synthesis. In this context, *cuts* are heavily used.

Definition 1. *A cut of a node n is a set of nodes c, called* leaves, *such that (i) every path from n to a primary input must visit at least one node in c, and (ii) every node in c must be included in at least one of these paths.*

As an example, $c1 = \{n5, n6, n8\}$ and $c2 = \{n7\}$ are cuts for $n8$ and $n7$ nodes, respectively. The nodes $n2$ and $n3$ are the inputs to both cuts. Cuts in an AIG can be computed using *cut enumeration* techniques [10].

3.2 SCA-based Verification

Before explaining the verification process using SCA, we summarize the important definitions and facts:

- A *monomial* $M = x_1^{a_1} x_2^{a_2} \dots x_n^{a_n}$ is the power product of the variables where $a_i \geq 0$
- A *polynomial* $P = c_1 M_1 + c_2 M_2 + \dots + c_j M_j$ is a finite sum of monomials with coefficients in field k
- A polynomial has a *monomial order* which is specified based on the ordering of variables and their powers
- Assuming p is a polynomial and F is a set of polynomials, the division of p by F is denoted by $p \xrightarrow{F} r$ where r is called remainder

In SCA-based verification of multipliers, the goal is to formally prove that the AIG representation (or gate-level netlist) and the *Specification Polynomial* (*SP*) are equivalent. The *SP* is a polynomial determining the function of a multiplier based on its inputs and outputs. By knowing the inputs and outputs names as well as the bit-width of the multiplier, it is easy to obtain the *SP*. For example, the specification polynomial for the 2-bit multiplier of Fig. 1 is $SP = 8Z[3]+4Z[2]+2Z[1]+Z[0]-(2A[1]+A[0])(2B[1]+B[0])$ where $8Z[3]+4Z[2]+2Z[1]+Z[0]$ describes the word-level representation of the 4-bit output, and $(2A[1] + A[0])(2B[1] + B[0])$ indicates the multiplication of the 2-bit inputs.

The nodes of an AIG graph can be captured as polynomials describing the relation between inputs and outputs. Based on the edges, five basic operations might occur for an AIG node with output z and inputs n_i and n_j.

$$z = n_i \Rightarrow p_N := z - n_i, \qquad z = n_i \wedge n_j \Rightarrow p_N := z - n_i n_j,$$
$$z = \neg n_i \Rightarrow p_N := z - 1 + n_i, \qquad z = \neg n_i \wedge n_j \Rightarrow p_N := z - n_j + n_i n_j,$$
$$z = \neg n_i \wedge \neg n_j \Rightarrow p_N := z - 1 + n_i + n_j - n_i n_j \tag{1}$$

Fig. 2 shows parts of the node polynomials of the 2×2 multiplier. Please note that all extracted polynomials are in the form $P_N = x - tail(P_N)$ where x is the node’s output, and $tail(P_N)$ is a function based on the node’s inputs.

Assume that the AIG nodes are ordered based on the reverse-topological order. The specification polynomial *SP* and the AIG are equivalent (i.e. the AIG is bug-free), iff the remainder of dividing *SP* by the ordered node polynomials is equal to zero. This theorem is concluded from the theory of Gröbner basis (see [4, 11] for more details).

The correctness of the 2×2 multiplier is proven in Fig. 3 by step-wise dividing *SP* by the node polynomials, which finally results in the remainder zero. In integer multipliers, dividing SP_i by a node polynomial $P_{N_i} = x_i - tail(P_{N_i})$ is equivalent to substituting x_i with $tail(P_{N_i})$ in SP_i. For example, dividing SP_2 by P_{n12} in Fig. 3 is equivalent to substituting $n12$ with $tail(P_{n12}) = 1 - n9 - n10 + n9n10$ in SP_2. The process of step-wise division (substitution) is known as *backward rewriting*.

3.3 Vanishing Monomials

Vanishing monomials are redundant monomials which are generated in intermediate steps of backward rewriting and are reduced to zero after potentially many steps. For non-trivial multipliers, the large number of generated vanishing monomials before reduction results in the explosion in the number of monomials, and consequently failure of verification. In this context, several papers exploited the fact that the product of HA’s outputs is zero for rewriting of the polynomials and by this to alleviate the vanishing monomial problem. In [8] a theory for the source of vanishing monomials has been introduced which is able to avoid the blow-up caused by the

vanishing monomials. The approach performs the following local backward rewriting:

(1) Detecting HAs in the multiplier architecture
(2) Finding all the gates where outputs of HAs converge. These gates are denoted as *Converging Gates* (CGs).
(3) Finding all cones starting from converging gates and ending in the related HA outputs. These cones are called *Converging Gate Cones* (CGCs).
(4) Extracting polynomials locally for CGCs, and removing vanishing monomials containing the product of HA outputs.

This local removal of vanishing monomials leads to a vanishing-free global backward rewriting.

4 ATOMIC BLOCKS IN SCA

In this section, we showcase the advantages of knowing the atomic blocks for SCA verification. Before this, we review the general structure of an integer multiplier and define atomic blocks.

4.1 Multiplier Architectures and Atomic Blocks

Fig. 4 shows the three stages of an integer multiplier as well as typical realizations for the stages on the right of the figure. The three stages are: (1) *Partial Product Generator* (PPG) which generates partial products from Multiplier and Multiplicand, (2) *Partial Product Accumulator* (PPA) which reduces the partial products by multi-operand adders and computes their sum, and (3) *Final Stage Adder* (FSA) which converts this sum to the corresponding binary output. In the rest of the paper, we use the notation $[\alpha\circ\beta\circ\gamma]$ to refer to a multiplier consisting of the stages: PPG α, PPA β, and FSA γ.

There are always some critical parameters in the design of multipliers such as area, delay, and power as well as technology constraints. These parameters play a major role in determining (i) which architecture is suitable for a specific stage and (ii) which realization for an atomic block can be chosen, see e.g. [20, 6, 3].

In the following we define how multiplier stages are realized using so called *atomic blocks*.

Definition 2. *An* atomic block *is a basic building block for a multiplier which gets n binary inputs with same bit positions, and computes their sum as m binary outputs. The typical atomic blocks with* 2, 3, *and* 5 *inputs are HA, FA, and CM. The corresponding word-level relations are:*

$$HA(in: X, Y \quad out: C, S) \Rightarrow 2C + S = X + Y \tag{2}$$
$$FA(in: X, Y, Z \quad out: C, S) \Rightarrow 2C + S = X + Y + Z$$
$$CM(in: X, Y, Z, W, Q \quad out: Co, C, S) \Rightarrow 2Co + 2C + S = X + Y + Z + W + Q$$

Please note that this definition does *not* require a specific realization of an atomic block. In fact, only the respective mathematical relation is defined (HA, FA, CM).

Definition 3. *A* specific multiplier architecture *consisting of the stages* $[\alpha\circ\beta\circ\gamma]$ *(cf. Fig. 4) is implemented by using atomic blocks and/or extra logic per stage. For trivial multipliers the PPA stage* β *and the FSA stage* γ *are only made of HA and FA atomic blocks. For non-trivial multipliers all kinds of atomic blocks plus highly parallel extra logic combining these blocks are allowed for all stages* [6].

In the next section we show how knowing the atomic blocks of multipliers helps for SCA-based verification.

4.2 Advantages of Atomic Blocks for SCA

Knowing atomic blocks in SCA-based verification of multipliers points up three major benefits.

4.2.1 Reveal All Vanishing Monomials. The authors of [8] have shown that detecting and removing *all* vanishing monomials before backward rewriting is the key factor for successful verification of non-trivial multipliers. However, the main disadvantage of this

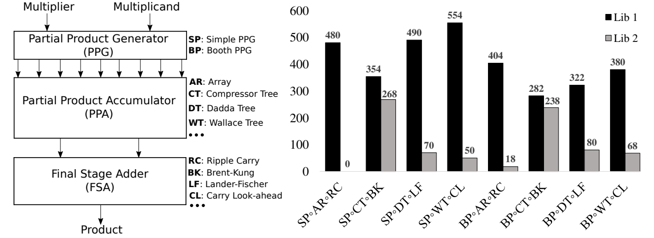
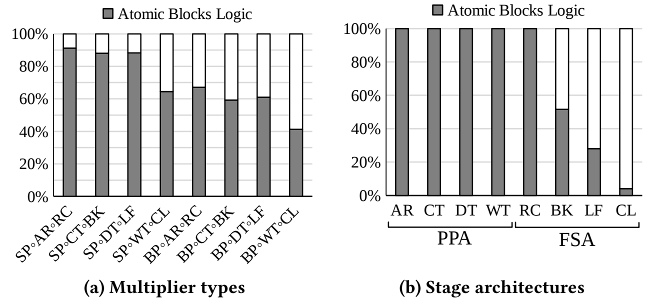


Figure 4: Mult arch. **Figure 5: AND-XOR numbers**



(a) Multiplier types **(b) Stage architectures**

Figure 6: Atomic blocks ratio in 64×64 multipliers

method is the trivial HA detection as it just looks for AND-XOR pairs (i.e. standard textbook HAs) in the design. Hence, this method only works for *"clean"* multipliers where all AND-XOR pairs are explicitly visible in the multiplier netlist. Please note that so far all other recent SCA approaches (see e.g. [18, 13, 11]) only considered HAs in the form of AND-XOR pairs. Obviously, an AND-XOR based HA agrees with our atomic block definition, but it is only a special case since there are many different realizations [6]. To demonstrate the consequence, we conducted an experiment where we varied the HA realization. Fig. 5 shows the number of detected AND-XOR pairs for several architectures of $16x16$ bit multipliers using two different HA/FA synthesis libraries. In case of the first synthesis library[2] (*Lib 1* in Fig. 5), all AND-XOR pairs are explicitly visible and the SCA-based method from [8] was able to verify all multipliers. However, when the second library[3] (*Lib 2* in Fig. 5) is used, most of the AND-XOR pairs disappear; an extreme case is the $SP\circ AR\circ RC$-multiplier where no AND-XOR pair remains. As a consequence, SCA-verif. fails due to vanishing monomial explosion.

Already, this experiment clearly shows that techniques are needed to make SCA verification more robust. We will show that our approach RevSCA is able to detect the atomic blocks independent of their realization using reverse engineering and hence backward rewriting becomes feasible.

4.2.2 Limit the Search Space for Vanishing Removal. In [8] the search space for finding converging gates is the entire netlist. The method first find all HAs, then traverses all paths from the HAs outputs to find possible converging gates. Nevertheless, this technique ignores the fact that (1) a large part of a multiplier is just made of atomic blocks, and (2) only a small part which can not be identified as atomic blocks, i.e. the extra logic, is responsible for generating vanishing monomials.

Fig. 6a shows the ratio of logic of atomic blocks to the entire logic in different multiplier architectures[4]. Despite the fact that this ratio slightly changes with respect to the design architecture, in average atomic blocks constitute 70% of a multiplier. In addition, Fig. 6b demonstrates the atomic blocks ratio for different PPA and

[2] HA realization as AND/XOR, and FA as two HAs and OR.
[3] Due to technology constraints in Lib 2 the HA realization is done without an XOR, and the FA realization with a multiplexer for faster carry computation.
[4] For this bar graph we have run the reverse engineering techniques of RevSCA.

Algorithm 1 RevSCA

Input: Multiplier AIG G, Specification polynomial SP
Output: TRUE if the circuit is correct, and FALSE otherwise
1: $AB \leftarrow$ ReverseEngineering (G) ▷ AB is the set of atomic blocks
2: $CN \leftarrow$ FindCones (G, AB) ▷ CN is the set of converging gate cones and fanout-free cones
3: $F \leftarrow$ ExtractVanishingFreePolys (CN) ▷ F is the set of cone polynomials
4: $r \leftarrow$ GlobalBackwardRewriting (SP, F, AB) ▷ r is the remainder
5: **if** $r == 0$ **then**
6: **return** $TRUE$
7: **else**
8: **return** $FALSE$

FSA architectures. Based on these results, we can conclude that the PPA stage of many non-trivial multipliers is completely made of atomic blocks. On the other hand, the FSA stage of the multipliers is a mixture of atomic blocks and extra logic and their ratio varies based on the architecture.

Overall, we will show that the reverse engineering techniques of RevSCA allow to limit the search space for finding the converging gates to the extra logic in the FSA. This drastically reduces the search time in the local vanishing removal phase.

4.2.3 Make Global Backward Rewriting Efficient. There is always a compact algebraic relation between inputs and outputs of an atomic block independent of the realization at the gate-level.

With respect to the fact that a large part of a design is constructed with atomic blocks (see Fig. 6a), detecting atomic blocks as done by RevSCA will speed up the global backward rewriting considerably.

5 REVSCA

In this section, we first give a top-level overview of our approach RevSCA. Then, we explain the three phases of RevSCA in detail.

5.1 Top-Level Algorithm

In order to overcome the obstacles of SCA-based verification techniques in proving the correctness of large and dirty multipliers, we introduce RevSCA. Algorithm 1 shows the pseudo-code of RevSCA consisting of three main phases: *Reverse Engineering*, *Local Vanishing Removal*, and *Global Backward Rewriting*. In the first phase, the atomic blocks are identified using a dedicated reverse engineering technique (see Line 1). Then, by knowing all atomic blocks in the multiplier, the converging gate cones, which are the source of vanishing monomials, are detected (Line 2). Consequently, the polynomial for each cone is obtained and vanishing monomials containing the product of HA outputs are removed (Line 3). Finally, the global backward rewriting is performed to substitute cone and atomic block polynomials in *SP* (Line 4). If the remainder is zero, the circuit is bug-free; otherwise, it is buggy (Line 5 – Line 8).

5.2 Reverse Engineering

In this section, we propose our dedicated reverse engineering method to identify atomic blocks in multipliers.

5.2.1 Atomic Blocks Specification Library. Before we can search on the design for atomic blocks, we have to specify the mathematical functions of the atomic blocks and collect them in a library. Since the atomic block functions depend only on a small number of inputs, we can make use of truth tables. Lets consider the example for the HA atomic block. The truth table of the HA outputs *sum* and *carry* can be seen in Fig. 7. We store the two output vectors $T_S = 0110$ and $T_C = 1000$ as the basic truth table for the HA. Since we represent the circuit netlist as an AIG we are interested in all variants of truth tables for an atomic block, i.e. we allow the negation for each input and output, respectively. For example, the truth table of the HA after negating the first input is $T_S^1 = 1001$, and $T_C^1 = 0010$ (see dashed area in Fig. 7). If n is the number of input bits and m is the number of output bits, in total 2^{m+n} sets of truth tables are

X	Y	S (T_S)	C (T_C)	S (T_S^1)	C (T_C^1)
1	1	0	1	1	0
1	0	1	0	0	0
0	1	1	0	0	1
0	0	0	0	1	0

Figure 7: HA truth table

X	Y	Z	W	Q	S	C	Co
1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1
1	1	1	0	1	0	1	1
1	1	1	0	0	1	1	0
1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	0
1	1	0	0	1	1	1	0
1	1	0	0	0	0	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	0	1	1	1	1	1	0
0	0	1	1	0	0	1	0
0	0	1	0	1	0	1	0
0	0	1	0	0	1	0	0
0	0	0	1	1	0	1	0
0	0	0	1	0	1	0	0
0	0	0	0	1	1	0	0
0	0	0	0	0	0	0	0

Figure 8: CM truth table

obtained after considering all possible combinations of negations for input and output bits. Following this principle, the complete set of truth tables of HAs and FAs can be obtained easily.

However, for the compressor CM, the story is different. The challenge originates from the fact that there are outputs with the same bit position (significance). For example this holds for the CM with outputs S, C, Co where C and Co have the same bit position. As a result, the value of these two outputs can be swapped for a certain input combination without changing the function of the CM. However, this would lead to the generation of a very big number of truth tables. We illustrate this by a concrete example: Fig. 8 shows the basic truth table (without any negations) of a CM and omitting some lines in the middle. As mentioned above, the i−th value of the vectors T_C and T_{Co} can be swapped. It means that if these values are not equal (red cells in Fig. 8), swapping them leads to generation of a completely new truth table. As in total there are 20 non-equal values of C and Co in the truth table, $2^{20} = 1,048,576$ new truth tables can be generated by swapping these values. To avoid dealing with millions of truth tables, we use arbitrary values X_i in T_C, and its complement $\overline{X_i}$ in T_{Co} where the i-th value of T_C and T_{Co} is different. For example in Fig. 8, T_C and T_{Co} can be encoded as:

$$\begin{aligned} T_C &= 111X_3 1X_5X_6X_7 \ldots X_{24}X_{25}X_{26}0X_{28}000 \\ T_{Co} &= 111\overline{X_3}1\overline{X_5}\overline{X_6}X_7 \ldots \overline{X_{24}X_{25}X_{26}}0\overline{X_{28}}000 \end{aligned} \quad (3)$$

The encoded values of T_C and T_{Co} in (3) cover all 2^{20} possible truth tables. Finally, all the obtained truth tables of atomic blocks are stored in the *Atomic Blocks Library* (ABLib).

5.2.2 Identifying Atomic Blocks in Multipliers. After creating ABLib, the next step is to identify atomic blocks in the multiplier. Algorithm 2 presents the general algorithm for identifying atomic blocks with n inputs and m outputs using ABLib. The input of the algorithm is the AIG G for a multiplier, the set of possible vectors for each output $ST_0, \ldots, ST_m$ from one concrete atomic block of ABLib, and the respective number of input bits n. The algorithm returns the list of found atomic blocks AB as output. First, all n-input cuts (cf. Definition 1) are computed on the AIG and stored in C (see Line 1). Then, the truth tables of the cuts are checked to see whether there is a cut c_i whose truth table is the member of one of the output vector sets ST_j. If yes, i.e. that the function of c_i is the same as the j-th output of the atomic block, and it is added to the list of possible candidates PC_j (Line 2 – Line 5). Subsequently, the possible candidates are scanned to find the set of cuts with the same inputs (Line 6). Finally, the cuts with the same inputs are merged since we have found an atomic block (Line 7).

We give an example: Consider the 2×2 multiplier of Fig. 1: $c1 = \{n5, n6, n8\}$ and $c2 = \{n7\}$ are among the extracted 2-input cuts. By computing the truth tables of these two cuts, the algorithm determines that T_{c1} and T_{c2} are members of ST_S and ST_C which are the set of possible vectors for *sum* and *carry* in ABLib, respectively. Moreover, $c1$ and $c2$ have exactly the same inputs $n2$ and $n3$. Therefore, merging these two cuts results in identifying the atomic block $B = \{n5, n6, n8, n7\}$ which is a HA.

Algorithm 2 Atomic blocks identification

Input: Mul AIG G, Set of output vectors $ST_1, \ldots, ST_m$ from ABLib, Atomic block inputs n
Output: List of identified atomic blocks AB
1: $C \longleftarrow$ FindCuts (G, n) ▷ Finding all n-input cuts
2: **for** $c_i \in C$ **do**
3: **for** $ST_j \in ST$ **do**
4: **if** TruthTable(c_i) $\in ST_j$ **then**
5: $PC_j = PC_j \cup c_i$
6: $SC \longleftarrow$ Find the cuts with the same inputs in $PC_0, PC_1, \ldots, PC_m$
7: $AB \longleftarrow$ Merge the cuts with the same inputs in SC
8: **return** AB

The run-time for computing cuts depends on the number of inputs for a cut, here n. In order to extract all atomic blocks efficiently, we first run Algorithm 2 for 2-input and 3-input cuts to detect all HAs and FAs. If the number of FAs is less than 20%[5] of the entire atomic blocks, then it can be concluded that the multiplier architecture has been implemented using larger atomic blocks, i.e. CM. Hence, we run the algorithm for 5-input cuts to detect the CMs.

5.3 Local Vanishing Removal

After the reverse engineering phase, all atomic blocks including HAs are identified in the multiplier. In order to ensure the cancellation of vanishing monomials, first all CGCs in the extra nodes are detected. As most of the circuit has now been classified as atomic blocks, the search space to find CGCs reduces to a small part of the multiplier. Then, the polynomial for each CGC is extracted by substitution of the node polynomials in the cone. The CGC polynomial determines the output of the cone based on its inputs. As vanishing monomials contain the product of HA’s outputs, and these outputs are the inputs of CGCs, removing vanishing monomials from the cone polynomials locally leads to a set of vanishing-free polynomials.

Fig. 9 shows a 4×4 non-trivial multiplier ($SP \circ WT \circ BK$ architecture). The first stage of the multiplier has been removed due to page limitation. H_n and F_m are the identified HAs and FAs, and $a, b, \ldots, h$ are the nodes of the extra logic in the AIG of the multiplier. Assuming A and B are two 4-bit inputs, the generated partial product from $A[i]$ and $B[j]$ is denoted by p_{ij}. The outputs of H_6 converge to d and e, so the corresponding CGCs are $y_1 = \{d, b\}$ and $y_2 = \{e, d, c, a\}$. The outputs of H_5 converge to e, consequently $y_3 = \{e, d, b, c, a\}$ is the only CGC for H_5. As e is a common converging gate for H_5 and H_6, we can merge y_2 and y_3 to obtain $y' = \{e, d, b, c, a\}$. On the other hand, y_1 is a subset of y'. Therefore, y' represents the only CGC in the multiplier. To extract the polynomial of y' and remove vanishing monomials, we start from the polynomial of the node located on the output of the cone i.e. $p_e = d - cd$, and continue substituting node polynomials until we reach the input of the cone. Finally, the monomials containing the product of H_5 and H_6 outputs ($C_{H_5}S_{H_5}$ and $C_{H_6}S_{H_6}$) are removed from the cone polynomial to obtain a vanishing-free polynomial. The rest of the nodes in the AIG graph, which are not part of any CGC, are grouped based on the fanout-free regions. For example $v = \{h, f, g\}$ is a fanout-free cone in Fig. 9.

5.4 Global Backward Rewriting

The final phase of verification is substituting the cone and atomic block polynomials in the specification polynomial. The relation of inputs and outputs for the atomic blocks (atomic block polynomials) can be obtained from Definition 2.

Assume that $SP_i = 2nC + nS + \sum_k M_k$ is the current polynomial during backward rewriting where C and S are the outputs of a FA, n is a coefficient, and $\sum_k M_k$ are the rest of the monomials. Based on (2), if X, Y, and Z are the inputs of the FA, the current polynomial after substitution will be $SP_{i+1} = nX + nY + nZ + \sum_k M_k$. The polynomial substitution for HAs and CMs is similar to FAs. Since more compact polynomials for atomic blocks can be inserted, the global backward rewriting after atomic block identification using reverse engineering is faster and more efficient.

[5]We justified this number by several experiments.

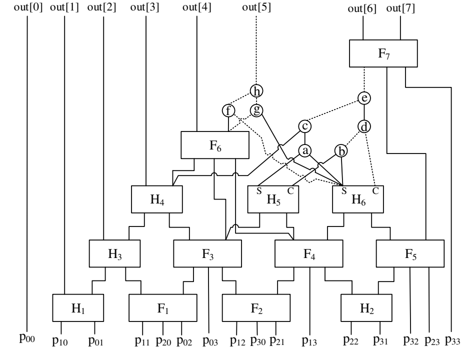


Figure 9: 4×4 non-trivial mult after reverse engineering

6 EXPERIMENTAL RESULTS

RevSCA has been implemented in C++. In order to extract cuts in the AIG for the reverse engineering phase, we used the mockturtle library [15]. The experiments have been carried out on an Intel(R) Xeon(R) CPU E3-1270 v3 3.50 GHz with 32 GByte of main memory. The efficiency of our proposed method is evaluated using a wide variety of trivial and non-trivial multiplier architectures. The multipliers with 16×16, 32×32, and 64×64 sizes have been generated with the AOKI generator [2]. This generator can build multiplier architectures only up to 64 bits per input. Therefore, we have generated multipliers up to 512 input bits using our own multiplier generator[6]. In addition, for the bigger multipliers, we used the same HA/FA synthesis library Lib 2 as in Section 4.2.1. All benchmarks have been converted to AIG using ABC [1].

In Table 1, we report the results of verifying different multiplier architectures. Please note that the *Time-Out* (TO) has been set to 150 hours. The first column of Table 1 presents the architecture of the multiplier based on its stages (see abbreviations below the table). The second column *Size* shows the size of the multiplier based on the input bits.

The verification data of our proposed method RevSCA is reported in the third column *Verification data*, which consists of five subcolumns: *Gates* shows the number of gates in the multiplier. *Atomic blocks* reports the number of identified atomic blocks after reverse engineering. *Cones* refers to the number of extracted vanishing-free and fanout-free cones. *Vanishing monomials* reports the total number of the removed vanishing monomials in the local vanishing removal phase. *Max poly size* shows the maximum size of the current polynomial SP_i during backward rewriting by counting the number of monomials.

Multipliers with 512 bit per input consist of more than 2 million gates. In these multipliers more than 38 million vanishing monomials have to be removed before backward rewriting to avoid the explosion in the number of monomials. The maximum polynomial sizes in *Max Poly Size* column indicates that RevSCA successfully avoid the blow up in the number of monomials by atomic block identification and removing all vanishing monomials.

The forth column of Table 1 reports the overall run-time of our proposed verification method which is the sum of consumed time for reverse engineering, local vanishing removal, and global backward writing. As can be seen, our approach can verify all multipliers with different architectures and sizes. Please note that the reverse engineering phase only constitutes 12% of the entire

[6]Our multiplier generator is available at http://www.sca-verification.org/genmul

Table 1: Results of verifying different multiplier architectures

Benchmark	Size	Verification data					Run-times (seconds)					
		Gates	Atomic Blocks	Cones	Vanishing Monomials	Max Poly Size	RevSCA	Commercial	[8]	[13]	[19]	[11]
$SP \circ BD \circ KS$	16×16	2,101	281	311	2,756	512	0.93	50.00	TO	TO	TO	TO
$BP \circ WT \circ CS$		1,821	195	388	56	823	0.49	47.00	TO	TO	TO	TO
$SP \circ DT \circ LF$	32×32	8,046	997	1,142	9,705	1,532	2.59	TO	6.37	64.62	TO	TO
$SP \circ WT \circ CL$		12,066	1,114	1,141	27,612	1,822	5.70	TO	13.58	1,045.89	TO	TO
$SP \circ BD \circ KS$		8,577	1,109	1,141	23,972	1,864	11.69	TO	TO	TO	TO	TO
$SP \circ AR \circ CK$		7,780	1,020	1,100	0	2,402	4.72	TO	TO	TO	TO	TO
$BP \circ AR \circ RC$		6,314	719	1,209	0	2,604	3.89	TO	4.21	51.19	0.02	TO
$BP \circ CT \circ BK$		5,766	652	1,261	946	3,621	9.06	TO	3.11	227.41	TO	TO
$BP \circ OS \circ CU$		7,357	673	1,454	0	3,449	6.88	TO	TO	TO	TO	TO
$BP \circ WT \circ CS$		6,640	706	1,277	56	3,383	5.14	TO	TO	TO	TO	TO
$SP \circ DT \circ LF$	64×64	32,680	4,038	4,341	76,515	6,058	31.10	TO	97.57	2,105.74	TO	TO
$SP \circ WT \circ CL$		52,083	4,365	4,340	266,684	6,930	96.27	TO	224.43	TO	TO	TO
$SP \circ BD \circ KS$		34,065	4,313	4,339	203,236	6,566	162.26	TO	TO	TO	TO	TO
$SP \circ AR \circ CK$		31,944	4,091	4,246	0	18,028	142.51	TO	TO	TO	TO	TO
$BP \circ AR \circ RC$		24,442	2,727	4,457	0	9,990	53.33	TO	56.80	882.52	0.09	TO
$BP \circ CT \circ BK$		21,872	2,413	4,589	3,586	15,890	119.15	TO	38.19	1,729.33	TO	TO
$BP \circ OS \circ CU$		26,821	2,509	5,077	80	13,434	94.91	TO	TO	TO	TO	TO
$BP \circ WT \circ CS$		24,830	2,548	4,584	0	13,176	74.88	TO	TO	TO	TO	TO
$SP \circ AR \circ RC$	128×128	129,535	16,256	16,639	0	16,640	348.60	TO	TO	TO	1.10	TO
$SP \circ WT \circ BK$		131,683	17,366	16,767	13,504	48,060	745.47	TO	TO	TO	TO	TO
$SP \circ DT \circ LF$		131,297	16,263	16,884	606,301	23,971	489.98	TO	TO	TO	TO	TO
$SP \circ AR \circ RC$	256×256	521,215	65,280	66,047	0	66,048	8,719.54	TO	TO	TO	Failed	TO
$SP \circ WT \circ BK$		526,520	67,974	66,304	52,217	430,112	21,454.30	TO	TO	TO	TO	TO
$SP \circ DT \circ LF$		525,531	65,288	66,547	4,823,639	96,654	12,873.65	TO	TO	TO	TO	TO
$SP \circ AR \circ RC$	512×512	2,091,007	261,632	263,167	0	263,168	192,640.30	TO	TO	TO	Failed	TO
$SP \circ WT \circ BK$		2,103,610	265,711	263,681	310,455	1,933,497	492,320.37	TO	TO	TO	TO	TO
$SP \circ DT \circ LF$		2,101,205	261,641	264,178	38,472,785	384,930	240,051.08	TO	TO	TO	TO	TO

Stage 1 ⇒ **SP**: Simple partial product generator **BP**: Booth partial product generator TO: Time-Out (150 hrs) Failed: Internal error
Stage 2 ⇒ **AR**: Array **BD**: Balanced delay tree **DT**: Dadda tree **WT**: Wallace tree **CT**: Compressor tree **OS**: Overturned-stairs tree
Stage 3 ⇒ **RC**: Ripple carry **BK**: Brent-Kung **LF**: Lander-Fischer **CL**: Carry look-ahead **KS**: Kogge-Stone **CK**: Carry-skip **CS**: Carry select **CU**: Conditional sum

run-time on average. On the other hand, global backward rewriting took up most of the run-time, i.e. 73% on average.

The run-times of the state-of-the-art verification methods are shown in the fifth column. This column consists of five subcolumns: While the first subcolumn *Commercial* reports the run-times of the commercial verification tool Onespin, the remaining subcolumns give the run-times of some of the most recent SCA verification techniques. The commercial tool only verifies 16×16 multipliers. The verification methods of [8] and [13] can verify some of the non-trivial multipliers due to the vanishing removal techniques. However, these methods are highly dependent on detection of AND-XOR pairs in the design. Therefore, for the benchmarks where AND-XOR pairs are not explicitly visible, their methods are blind and time out in verification. The proposed method of [19] can verify trivial multipliers (i.e. *BP∘AR∘RC* and *SP∘AR∘RC*) very fast. However, it fails to prove the correctness of non-trivial multipliers. Finally, the work of [11] fails in verification of all benchmarks.

7 CONCLUSION

In this paper, we have proposed a fast and robust SCA-based verification method integrating dedicated reverse engineering to verify big and dirty multipliers. The approach takes advantage of atomic block identification to overcome several obstacles when verifying non-trivial multipliers. The experimental results showed that our method allows for verification of a wide variety of multiplier architecture with up to 1024 output bits and more than 2 million gates while the other state-of-the-art approaches fail.

Acknowledgements: This work was supported by the University of Bremen's graduate school SyDe funded by the German Excellence Initiative, and by the German Academic Exchange Service (DAAD).

REFERENCES

[1] Abc: A system for sequential synthesis and verification. available at https://people.eecs.berkeley.edu/~alanmi/abc/, 2018.

[2] Arithmetic module generator based on acg. available at https://www.ecsis.riec.tohoku.ac.jp/topics/amg/i-amg, 2019.

[3] M. Bahadori, M. Kamal, A. Afzali-Kusha, and M. Pedram. High-speed and energy-efficient carry skip adder operating under a wide range of supply voltage levels. *TVLSI*, 24(2):421–433, Feb. 2016.

[4] D. A. Cox, J. Little, and D. O'Shea. *Ideals Varieties and Algorithms*. Springer, 1997.

[5] F. Farahmandi and B. Alizadeh. Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. *MICPRO*, 39(2):83–96, 2015.

[6] I. Koren. *Computer Arithmetic Algorithms*. A. K. Peters, Ltd., 2nd edition, 2001.

[7] A. Mahzoon, D. Große, and R. Drechsler. Combining symbolic computer algebra and boolean satisfiability for automatic debugging and fixing of complex multipliers. In *ISVLSI*, pages 351–356, 2018.

[8] A. Mahzoon, D. Große, and R. Drechsler. PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In *ICCAD*, pages 129:1–129:8, 2018.

[9] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *DAC*, pages 532–535, 2006.

[10] P. Pan and C.-C. Lin. A new retiming-based technology mapping algorithm for lut-based fpgas. In *FPGAs for Custom Computing Machines*, pages 35–42, 1998.

[11] D. Ritirc, A. Biere, and M. Kauers. Column-wise verification of multipliers using computer algebra. In *FMCAD*, pages 23–30, 2017.

[12] D. Ritirc, A. Biere, and M. Kauers. Improving and extending the algebraic approach for verifying gate-level multipliers. In *DATE*, pages 1556–1561, 2018.

[13] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler. Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In *DATE*, pages 1048–1053, 2016.

[14] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler. Equivalence checking using Gröbner bases. In *FMCAD*, pages 169–176, 2016.

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

[16] D. Stoffel and W. Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *TCAD*, 23(5):586–597, 2004.

[17] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham. Automatic verification of arithmetic circuits in RTL using stepwise refinement of term rewriting systems. *TC*, 56(10):1401–1414, 2007.

[18] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski. Formal verification of arithmetic circuits by function extraction. *TCAD*, 35(12):2131–2142, 2016.

[19] C. Yu, M. Ciesielski, and A. Mishchenko. Fast algebraic rewriting based on and-inverter graphs. *TCAD*, 37(9):1907–1911, 2017.

[20] R. Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*. PhD thesis, Swiss Federal Institute of Technology, 1997.