MEDIA SuUB Bremen
https://media.suub.uni-bremen.de

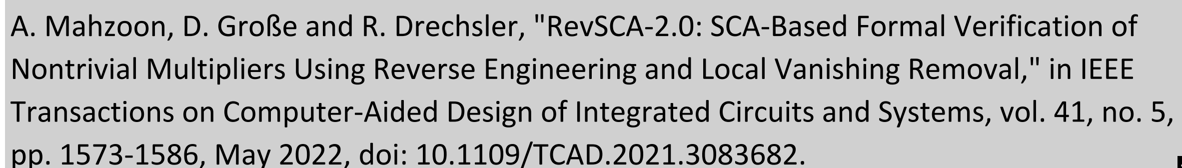
Titel/Title: RevSCA-2.0: SCA-Based Formal Verification of Nontrivial Multipliers Using Reverse Engineering and Local Vanishing Removal

Autor*innen/Author(s): Alireza Mahzoon, Daniel Große and Rolf Drechsler

Veröffentlichungsversion/Published version: Postprint

Publikationsform/Type of publication: Artikel/Aufsatz

Empfohlene Zitierung/Recommended citation:



A. Mahzoon, D. Große and R. Drechsler, "RevSCA-2.0: SCA-Based Formal Verification of Nontrivial Multipliers Using Reverse Engineering and Local Vanishing Removal," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 5, pp. 1573-1586, May 2022, doi: 10.1109/TCAD.2021.3083682.

Verfügbar unter/Available at:
(wenn vorhanden, bitte den DOI angeben/please provide the DOI if available)

10.1109/TCAD.2021.3083682

Zusätzliche Informationen/Additional information:

© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

REVSCA-2.0: SCA-Based Formal Verification of Nontrivial Multipliers Using Reverse Engineering and Local Vanishing Removal

Alireza Mahzoon, *Graduate Student Member, IEEE*, Daniel Große, *Senior Member, IEEE*, and Rolf Drechsler, *Fellow, IEEE*

***Abstract*—The formal verification of integer multipliers is one of the important but challenging problems in the verification community. Recently, the methods based on symbolic computer algebra (SCA) have shown very good results in comparison to all other existing proof techniques. However, when it comes to verification of huge and structurally complex multipliers, they completely fail as an explosion happens in the number of monomials. The reason for this explosion is the generation of redundant monomials known as vanishing monomials. This article introduces the SCA-based approach REVSCA-2.0 that combines reverse engineering and local vanishing removal to verify large and nontrivial multipliers. For our approach, we first come up with a theory for the origin of vanishing monomials, i.e., we prove that the gates/nodes where both outputs of half adders (HAs) converge are the origins of vanishing monomials. Then, we propose a dedicated reverse engineering technique to identify atomic blocks including HAs. The identified HAs are the basis for detecting converging cones and locally removing vanishing monomials, which finally results in a vanishing-free global backward rewriting. The efficiency of REVSCA-2.0 is demonstrated using an extensive set of multipliers with up to several million gates.**

***Index Terms*—Formal verification, multiplier, reverse engineering, symbolic computer algebra (SCA), vanishing monomial.**

I. INTRODUCTION

MULTIPLIERS nowadays play a crucial role in a wide range of different applications, e.g., signal processing and cryptography, as well as upcoming AI solutions employing machine learning and deep learning. Most of these applications require very large multipliers supporting a wide range of integer numbers. Furthermore, the multiplier architectures also vary based on the design goals in different applications. Several multiplication algorithms have been developed to satisfy the community demands for fast, area-efficient, and low-power designs. Employing these algorithms usually results in the generation of very complex architectures. Formal verification of huge and structurally complex multipliers is on the one hand necessary to ensure the correctness of the final design. On the other hand, it is a big challenge where most of the existing formal methods completely fail.

After the famous Pentium bug back in 1994, a lot of effort has been put into the development of formal verification methods. Although these methods accomplished big successes in many domains, they suffer from serious limitations when it comes to the verification of integer multipliers: 1) decision diagrams (DDs) (such as BDDs and *BMDs) are facing memory blow up due to the exponential growth in the size of the graph when the input width increases[1]; 2) *Boolean satisfiability* (SAT) and satisfiability modulo theories (SMTs) are not scalable and fail to verify large multipliers; 3) *theorem proving* is not automated, and needs considerable manual effort before checking the correctness; 4) reverse engineering approaches using *arithmetic bit level* [1] are exponential in the detection of carry structures and therefore, cannot support all multiplier architectures; and finally 5) *term rewriting* techniques [2] rely on a database of rewrite rules to support a wide range of architectures, however, for implementations that are not yet represented in the database a manual update of the database is required.

Recently, symbolic computer algebra (SCA) verification methods have shown very good results in proving the correctness of large but structurally simple integer multipliers [3]–[6]. They have been also employed in equivalence checking [7], debugging of faulty multipliers [8], [9], and verification of dividers [10], [11]. The general idea of the 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 [or nodes of an AND-inverter graph (AIG)] as a set of polynomials P_G; and 3) take advantage of the Gröbner basis theory to prove the membership of *SP* in the ideal generated by P_G. The just mentioned third step consists of the stepwise division of *SP* by P_G (or equivalently substitution of variables in *SP* with P_G) known as *backward rewriting*, and eventually, the evaluation of the remainder. If this remainder is zero, the multiplier is correct. Otherwise, it is buggy.

Despite the success of SCA-based methods in the verification of simple integer multipliers, verification of nontrivial multipliers (i.e., structurally complex multipliers) including

Manuscript received February 13, 2020; revised August 19, 2020 and January 18, 2021; accepted May 4, 2021. Date of publication May 25, 2021; date of current version April 21, 2022. This work was supported by the German Research Foundation (DFG) within the Project VerA under Grant GR 3104/6-1 and Grant DR 297/37-1. This article was recommended by Associate Editor P. Stanley-Marbell. *(Corresponding author: Alireza Mahzoon.)*

Alireza Mahzoon and Rolf Drechsler are with the Institute of Computer Science, University of Bremen, 28359 Bremen, Germany (e-mail: mahzoon@informatik.uni-bremen.de; drechsle@informatik.uni-bremen.de).

Daniel Große is with the Institute for Complex Systems, Johannes Kepler University Linz, 4040 Linz, Austria (e-mail: daniel.grosse@jku.at).

Digital Object Identifier 10.1109/TCAD.2021.3083682

[1]In contrast to BDDs, *BMDs are able to represent the multiplier function efficiently, but exponential memory peak sizes have been observed during the *BMD construction for architecturally complex multipliers.

highly parallel architectures is still a big challenge for these methods as an explosion happens in the number of monomials during backward rewriting. The dramatic increase in the number of monomials makes the calculations on the current polynomial very expensive and practically impossible in the case of large bug-free multipliers. A common understanding is that one of the main reasons for this explosion is the generation of redundant monomials known as *vanishing monomials*. These monomials are generated during verification of nontrivial multipliers, and reduced to zero after several steps of division/substitution. However, the huge number of vanishing monomials before cancelation causes a blow-up in computations.

In this article, we propose REVSCA-2.0*, an approach that combines reverse engineering and local vanishing removal to verify large and nontrivial multipliers*.[2] To understand the limitations of the SCA-based verification methods, we have conducted several experiments on different multiplier architectures. We clearly show how these methods fail due to the large number of vanishing monomials generated in the backward rewriting steps. After analyzing the intermediate results of the substitution during backward rewriting, we come up with an extended theory for the origin of vanishing monomials. The theory is based on basic building blocks heavily found in every multiplier architecture. We call these building blocks *atomic blocks*, and prominent examples include HA, full-adder (FA), and compressor (CM). Utilizing these atomic blocks allows us to state and prove a theorem on the origins of vanishing monomials. Essentially, the vanishing monomials originate from logical gates of the netlist (or nodes in the AIG representation) to which the output paths of an HA converge. A monomial is formed at these gates/nodes during backward rewriting, which creates many new (vanishing) monomials in each following substitution step. These monomials remain in calculations and, even worse, make the current polynomial larger and larger with each new substitution step until the HA is reached. After substituting the gate/node polynomials of the HA, all these vanishing monomials are reduced to zero.

Therefore, to avoid the explosion during backward rewriting, we divide backward rewriting into a global step and several local rewriting steps. Based on our theory, in the local rewriting steps, we can create a vanishing-free polynomial representation for different parts of the multiplier. For this, we have to: 1) identify all atomic blocks including HAs and 2) find converging cones starting from HAs and remove vanishing monomials locally. The first step requires reverse engineering that can be performed very fast on AIGs using cut enumeration. This lays the foundation for our algorithm for removing the vanishing monomials locally such that a fast global backward rewriting becomes possible. We have implemented all this in the SCA-verifier REVSCA-2.0 to verify nontrivial million-gate multipliers, which was not possible before.[3]

II. RELATED WORK

In the last five years, several SCA-based methods have been introduced to verify integer multipliers. Yu *et al.* [4] and Ciesielski *et al.* [14] proposed a method to capture the gate-level netlist as a set of polynomials, and then, substituting these polynomials in the specification polynomial step-by-step following the reverse topological order of the circuit. The work of [3] divides the netlist into the fanout-free cones and extracts the polynomial for each cone. Subsequently, it uses the cone polynomials instead of the gate polynomials in substitution steps to reduce the total number of generated monomials during backward rewriting. The columnwise method of [5] and [15] cuts the circuit into slices and verifies correctness incrementally. The just mentioned approaches have two main disadvantages: 1) they extract the polynomials for the smallest building blocks of a multiplier, i.e., gates, thus these methods are unaware of larger building blocks, e.g., HAs and FAs, having more compact polynomials and 2) they only work for structurally simple multipliers where no vanishing monomials appear during backward rewriting.

The proposed approaches in [6] and [16] take advantage of reverse engineering to identify HAs and FAs in the AIG representation of a multiplier. Then, they use the compact polynomials of HAs and FAs during backward rewriting, which speeds up the verification process significantly. However, these approaches do not provide any solution to avoid explosion during the backward rewriting of nontrivial multipliers, which confines their applicability to structurally simple designs. Furthermore, they do not support the detection of larger atomic blocks such as compressors.

The work of [17] aims to attack the vanishing monomials problem and make the verification of nontrivial multipliers possible. It presents an XOR rewriting technique, which groups the gates into cones based on the XOR gates. Then, it extracts the polynomials for each cone and removes vanishing monomials. The method works for some complex architectures. However, it is not robust since it misses many vanishing monomials. Moreover, the approach does not investigate the origin of vanishing monomials in nontrivial multipliers.

The proposed method in [18] uses a combination of SAT and SCA to verify nontrivial multipliers. The authors come up with an algorithm to detect the final stage adder (FSA) in a multiplier and verify it using SAT. Then, the adder is substituted with an architecturally simple adder. Finally, the SCA-based verification is performed on the new architecture. The method achieves very good runtimes if the FSA can be detected, which is not always possible.

III. PRELIMINARIES

In this section, we first introduce the general multiplier structure. Then, we review the AIG representation of a circuit. Finally, we explain the SCA-based verification of multipliers in detail.

A. Multiplier Structure

An integer multiplier consists of three stages: 1) partial product generator (PPG); 2) partial product accumulator (PPA); and 3) FSA. The PPG stage generates partial products from the multiplier and the multiplicand inputs. Then, the PPA stage reduces the partial products by multioperand adders and computes their sum. Eventually, the sum is converted to the corresponding binary output at the FSA [19], [20].

Several algorithms have been proposed to implement each stage of an integer multiplier. The architectures generated by them have some pros and cons in terms of design parameters,

[2] Our tool REVSCA-2.0 and all benchmarks are available on GitHub; links can be found at http://www.sca-verification.org/revsca.

[3] This journal paper includes and extends published material from the two previous conference papers [12], [13].

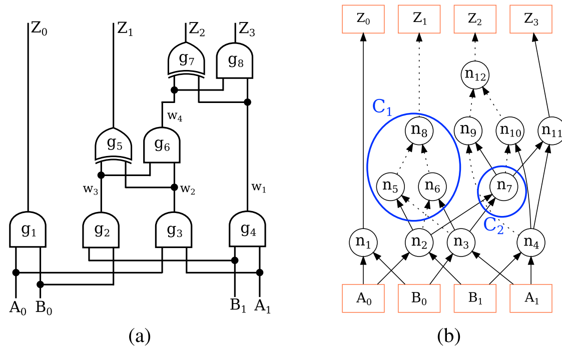


Fig. 1. 2×2 unsigned multiplier. (a) Gate-level representation. (b) AIG representation.

e.g., area, delay, power, and the number of wiring tracks. The designer can choose between different algorithms to achieve the design goal, e.g., minimizing the chip area. For example, Booth PPG generates fewer partial products compared to simple PPG; thus, it reduces the overall area of the multipliers with long operands. However, it has a higher design and logic complexity. As another example, Wallace tree and balanced delay tree are two well-known algorithms for implementing the PPA stage. Wallace tree guarantees the lowest overall delay but it has the largest number of wiring tracks. On the other hand, balanced delay tree requires the smallest number of wiring tracks but suffers from the highest overall delay compared to other algorithms. In the remainder of this article, we use the notation $[\boldsymbol{\alpha} \circ \boldsymbol{\beta} \circ \boldsymbol{\gamma}]$ to refer to a multiplier consisting of the stages: PPG α, PPA β, and FSA γ.

B. AND-Inverter Graph

An AIG is a directed acyclic graph with the following properties.

1) A node has either zero or two incoming edges.
2) A node with no incoming edge is a primary input (PI).
3) A node with two incoming edges is an AND gate.
4) A complemented edge indicates the negation of a signal.

Fig. 1(a) and Fig. 1(b) show the gate-level netlist of a 2×2 unsigned multiplier and its AIG representation,[4] respectively. The dashed lines in Fig. 1(b) indicate the complemented edges.

AIGs and particularly, the *cut* concept are widely used in logic synthesis since it helps for optimization.

Definition 1: A cut of a node n is a set of nodes C, called leaves, such that: 1) every path from n to a PI must visit at least one node in C and 2) every node in C must be included in at least one of these paths.

Example 1: In Fig. 1(b), $C_1 = \{n_5, n_6, n_8\}$ and $C_2 = \{n_7\}$ are cuts for the nodes n_8 and n_7, respectively. The nodes n_2 and n_3 have output edges to both cuts C_1 and C_2; thus, n_2 and n_3 are inputs of C_1 and C_2.

Cuts on an AIG can be computed using *cut enumeration* [21], [22], which we use for reverse engineering.

C. SCA-Based Verification

Before explaining the SCA-based verification of multipliers, we first give the basic definitions.

Definition 2: A *monomial* is the power product of variables

$$M = x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n} \quad \text{with} \quad \alpha_i \in \mathbb{N}_0. \tag{1}$$

A monomial with a coefficient is called a *Term*.

Definition 3: A *polynomial* is a finite sum of monomials with coefficients in k

$$P = c_1 M_1 + c_2 M_2 + \dots + c_j M_j \quad \text{with} \quad c_i \in k. \tag{2}$$

In the remainder of this article, the coefficients are always integers from $\mathbb{Z}$. Although $\mathbb{Z}$ is not a field, it has been proven in [18] that we can use it when it comes to the verification of circuits.

The order of monomials in a polynomial can be determined by Lexicographic order. Assume that the variables are ordered as $x_1 > x_2 > x_3 > \cdots$ The lexicographic order first compares exponents of x_1 in the monomials, and in the case of equality, it compares exponents of x_2, and so forth.

One of the crucial operations in SCA is the division of a given polynomial p by a set of polynomials F denoted by $p \xrightarrow{F} r$, where r is the remainder of the division.

Example 2: If $p = xy$, $f_1 = x - z$, and $f_2 = yz$ with the variable order $x > y > z$, then $xy \xrightarrow{f_1} yz \xrightarrow{f_2} 0$. To perform the division of xy by f_1, first f_1 is multiplied by y to create the same leading monomial xy as p, so $f_1 y = xy - zy = xy - yz$. Subsequently, the subtraction is performed, i.e., $p - (f_1 y) = xy - (xy - yz) = yz$, which is the result of the first division. Finally, yz is divided by f_2 to get remainder 0.

In SCA-based verification, the goal is to formally prove that all signal assignments consistent with the AIG or gate-level netlist evaluate the specification polynomial (*SP*) to 0. The *SP* is a polynomial determining the function of an arithmetic circuit based on its inputs and outputs. For an $N \times N$ unsigned integer multiplier with $A_{N-1}A_{N-2}\dots A_0$ and $B_{N-1}B_{N-2}\dots B_0$ inputs and $Z_{2N-1}Z_{2N-2}\dots Z_0$ output, the *SP* is

$$SP = \sum_{i=0}^{2N-1} 2^i Z_i - \left(\sum_{i=0}^{N-1} 2^i A_i\right) \times \left(\sum_{i=0}^{N-1} 2^i B_i\right). \tag{3}$$

For signed multipliers using two's complement, the *SP* is slightly different and equal to

$$\begin{aligned} SP = {} & -2^{2N-1} Z_{2N-1} + \sum_{i=0}^{2N-2} 2^i Z_i \\ & - \left(-2^{N-1} A_{N-1} + \sum_{i=0}^{N-2} 2^i A_i\right) \\ & \times \left(-2^{N-1} B_{N-1} + \sum_{i=0}^{N-2} 2^i B_i\right). \end{aligned} \tag{4}$$

Example 3: The SP for the 2×2 unsigned multiplier of Fig. 1 is $SP = 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (2A_1 + A_0)(2B_1 + B_0)$, where $8Z_3 + 4Z_2 + 2Z_1 + Z_0$ shows the world-level representation of the 4-bit output, and $(2A_1 + A_0)(2B_1 + B_0)$ indicates the product of the 2-bit inputs.

The nodes of an AIG (or the gates of a netlist) can be captured as polynomials describing the relation between inputs and output. Each AIG node with output z and inputs n_i and n_j performs one of the five basic operations

$$z = n_i \Rightarrow P_N := z - n_i$$
$$z = n_i \wedge n_j \Rightarrow P_N := z - n_i n_j$$

[4] Ignore the label annotations, i.e., C_1 and C_2, for now.

$$z = \neg n_i \Rightarrow P_N := z - 1 + n_i$$
$$z = \neg n_i \wedge n_j \Rightarrow P_N := z - n_j + n_i n_j$$
$$z = \neg n_i \wedge \neg n_j \Rightarrow P_N := z - 1 + n_i + n_j - n_i n_j. \tag{5}$$

The polynomials in (5) are in the form $P_N = x - \text{tail}(P_N)$ where x is the node output, and $\text{tail}(P_N)$ is a function based on the node inputs.

Example 4: The captured node polynomials for the AIG representation of the multiplier in Fig. 1(b) are

$$\begin{aligned} P_{Z_3} &:= Z_3 - n_{11}, \\ P_{Z_2} &:= Z_2 - 1 + n_{12}, \\ P_{n_{12}} &:= n_{12} - 1 + n_9 + n_{10} - n_9 n_{10}, \\ &\ldots \\ P_{n_3} &:= n_3 - A_1 B_0, \\ P_{n_2} &:= n_2 - A_0 B_1, \\ P_{n_1} &= n_1 - A_0 B_0. \end{aligned} \tag{6}$$

Note that the variables are ordered based on the reverse topological order of the circuit, i.e., if z is the output of a gate and $\{x, y\}$ are inputs, then $z > \{x, y\}$.

Theorem 1: Assume that the AIG nodes are ordered based on the reverse topological order of the circuit. All signal assignments consistent with the AIG evaluate the SP to 0 iff the remainder of dividing SP by the node polynomials is equal to 0.

Theorem 1 is concluded from the Gröbner basis theory. Refer to [5], [15], and [23] for the proof.

Example 5: The correctness of the 2×2 multiplier in Fig. 1(b) is proven by the stepwise division of SP by the node polynomials as follows:

$$\begin{aligned} &SP := 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (2A_1 + A_0)(2B_1 + B_0) \\ &SP \xrightarrow{P_{Z_3}} SP_1 := 8n_{11} + 4Z_2 + 2Z_1 + Z_0 \\ &\qquad - (2A_1 + A_0)(2B_1 + B_0) \\ &SP_1 \xrightarrow{P_{Z_2}} SP_2 := 8n_{11} + 4 - 4n_{12} + 2Z_1 + Z_0 \\ &\qquad - (2A_1 + A_0)(2B_1 + B_0) \\ &\quad \vdots \\ &SP_{13} \xrightarrow{P_{n_3}} SP_{14} := n_2 + n_1 - A_0 B_1 - A_0 B_0 \\ &SP_{14} \xrightarrow{P_{n_2}} SP_{15} := n_1 - (A_0 B_0) \\ &SP_{15} \xrightarrow{P_{n_1}} r := 0. \end{aligned} \tag{7}$$

The remainder r equals zero, thus the multiplier is bug free.

In the verification of integer multipliers, all variables in polynomials are Boolean. Thus, x^n can be replaced by x. Furthermore, dividing SP_i by a node polynomial $P_N = x - \text{tail}(P_N)$ is equivalent to *substituting* x with $\text{tail}(P_N)$ in SP_i. For example, to obtain the result of the first division step in Example 5, Z_3 can be substituted with n_{11} in SP to obtain SP_1. The process of dividing the SP by node polynomials (or equivalently substituting node polynomials in the SP) is called *backward rewriting*. We always prefer substitution over division as the substitution is less expensive in terms of run-time [24]. We refer to the intermediate polynomial during backward rewriting as SP_i in the rest of this article.

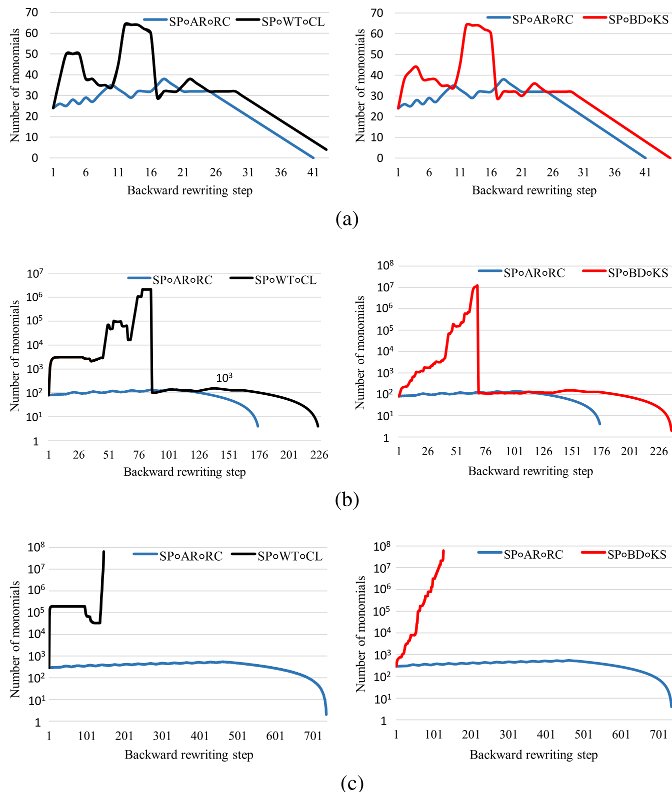


Fig. 2. Number of monomials at each step of backward rewriting. (a) 4×4 multipliers. (b) 8×8 multipliers. (c) 16×16 multipliers.

IV. Limitations of SCA-Based Verification

In this section, we investigate the efficiency of SCA-based verification in proving the correctness of different multiplier architectures and illustrate the limitations. To achieve this goal, we provide experimental evidence for the verification of three types of multipliers. These multiplier architectures are as follows.

1) *Simple PPG ∘ array ∘ ripple carry adder* (*SP∘AR∘RC*), which is a trivial multiplier.
2) *Simple PPG ∘ Wallace tree ∘ carry look-ahead* (*SP ∘ WT∘CL*), which is a highly parallel and thus, a nontrivial multiplier.
3) *Simple PPG ∘ balanced delay tree ∘ Kogge-Stone* (*SP ∘ BD∘KS*), which again has a highly parallel and therefore, a nontrivial architecture.

The results for different multiplier sizes are shown in Fig. 2(a), Fig. 2(b), and Fig. 2(c), respectively. In the figures, we plot the number of monomials in the consecutive substitution steps of backward rewriting. The results point us to two important observations.

1) For the trivial multipliers, i.e., *SP ∘ AR ∘ RC* (blue lines), the number of monomials remains almost constant during backward rewriting. Then, it starts to decrease at the final steps until it eventually becomes one.[5]
2) During the verification of nontrivial multipliers, i.e., *SP ∘ WT ∘ CL* (black lines) and *SP ∘ BD ∘ KS* (red lines), the number of monomials grows dramatically after a few substitution steps. For the *SP ∘ WT ∘ CL* (*SP ∘ BD ∘ KS*) multipliers with 4×4 and 8×8 input sizes, the

[5] All multipliers considered here are correct; hence, the final result is the zero polynomial containing only one monomial, which is 0.

number of monomials reaches 2.7× (2.7×) and 26 000× (150 000×) compared to the initial number of monomials, respectively. The situation is even worse for the 16×16 nontrivial multipliers. As can be seen in Fig. 2c, the number of monomials explodes after about 150 steps of substitution for both nontrivial multipliers.

In general, the exponential growth in the size of SP_i makes the verification of nontrivial multipliers with input bit width larger than 8 bit practically impossible.

In the last five years, some methods have been proposed to overcome the monomial explosion problem. As a common understanding, so-called *vanishing monomials* (redundant monomials that are finally reduced to zero after several steps of substitution) are the root cause of the explosion [5], [17]. As already discussed in the related work section, the previous approaches either consider large but trivial architectures where no vanishing monomial appears, or carry out rewriting of the polynomials before performing backward rewriting but do not put insight into the vanishing monomial problem and are, therefore, not robust. In this article, we present an extended theory for the origin of vanishing monomials. Then, we come up with an approach to locally remove vanishing monomials and thus, prevent the explosion during global backward rewriting.

V. Vanishing Monomials

In this section, we first present an illustrative example to show vanishing monomials in SCA-based backward rewriting of a nontrivial multiplier. Then, we make the general case of vanishing monomials, i.e., we come up with the basic theory for the origin of vanishing monomials. Finally, we clarify the relation between vanishing monomials and different multiplier architectures.

A. Vanishing Monomials Example

As a circuit example, we consider a 3×3 unsigned multiplier of type *[simple PPG ∘ Wallace tree ∘ carry look-ahead adder]* (*SP ∘ WT ∘ CL*). The AIG representation of the multiplier is shown in Fig. 3. We assume the atomic blocks, including HAs and FAs, are identified before the backward rewriting process using reverse engineering techniques (more details in Section VII-B). We use H_i and F_i to show the HA and FA blocks, respectively. The AIG nodes for H_4 (i.e., n_k, n_l, n_m, n_o), H_5 (i.e., n_x, n_y, n_z, n_t), and H_6 (i.e., n_p, n_u, n_q, n_r) are depicted in Fig. 3. For the rest of the HA and FA blocks, the internal nodes are not shown to keep the size of the circuit small and to avoid confusion. As can be seen in the figure, the inputs of the multiplier are $A = A_2A_1A_0$ and $B = B_2B_1B_0$ (to simplify the graph, we omit the input terminals, but mark the successor nodes accordingly), while the output is $Z = Z_5Z_4Z_3Z_2Z_1Z_0$.

An excerpt of the substitution steps when performing backward rewriting for the 3×3 nontrivial multiplier is depicted in Fig. 4.

1) *SP* is the specification polynomial for the 3×3 multiplier at hand. Performing backward rewriting in reverse topological order, i.e., substituting variables in *SP* with the node polynomials of Fig. 3, will finally result in the remainder zero,[6] since the considered AIG representation is correct.
2) In the first step of backward rewriting, Z_5, which is one of the primary outputs (POs) of the circuit, is substituted with $1 - n_A$ [see NOT polynomial in (5)]. The result after the substitution is shown as the new polynomial SP_1. Since the coefficient of Z_5 is 32, we have to perform the multiplication $32(1 - n_A) = 32 - 32n_A$.
3) Subsequently, n_A is substituted with $n_C - n_C C_{H_5}$ to obtain the new polynomial SP_2.
4) The next 14 steps of backward rewriting are omitted due to page limitation.
5) Four intermediate backward rewriting steps, which are done by substituting n_o, S_{H_5}, n_z, and n_t, are presented, respectively. The intermediate result after the aforementioned substitutions is SP_{24} (see bold line).

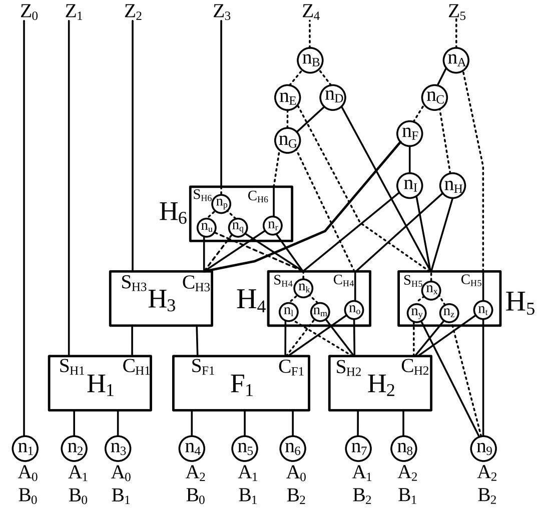


Fig. 3. AIG representation of a 3 × 3 nontrivial multiplier.

As can be seen in Fig. 4, we have marked several monomials in red. The reason is that they are finally reduced to zero, i.e., after substituting S_{H_5}, C_{H_5}, n_x, n_y, n_z, and n_t, they are canceled out completely in SP_{24}. Hence, we call them vanishing monomials. Before explaining the origin and properties of vanishing monomials, we provide some numbers. We have 3 red monomials (12 variables) in SP_{17}, 6 red monomials (21 variables) in SP_{18}, and 15 red monomials (72 variables) in SP_{21}. These numbers show an explosion in the backward rewriting of the 3×3 nontrivial multiplier of Fig. 3. Note that even more vanishing monomials appear in the complete backward rewriting steps.

Now, two major questions arise as follows.

1) Why are the red monomials finally reduced to zero in SP_{24}?
2) What is the origin of the red monomials?

For Answering 1), just take a look on all three red monomials in SP_{17}. They all contain the product $C_{H_5}S_{H_5}$. In the next six substitution steps, this product is reduced to zero[7]

$$
\begin{aligned}
C_{H_5}S_{H_5} &= n_t(1 - n_x) = n_t - n_t n_x \\
&= n_t - n_t\big(1 - n_y - n_z + n_y n_z\big) \\
&= n_t n_y + n_y n_z - n_t n_y n_z \\
&= C_{H_2}n_9\big(n_9 - C_{H_2}n_9\big) + C_{H_2}n_9\big(C_{H_2} - C_{H_2}n_9\big) \\
&\quad - C_{H_2}n_9\big(n_9 - C_{H_2}n_9\big)\big(C_{H_2} - C_{H_2}n_9\big) = \\
&= \cancel{C_{H_2}n_9} - \cancel{C_{H_2}n_9} + \cancel{C_{H_2}n_9} - \cancel{C_{H_2}n_9} \\
&\quad - \cancel{C_{H_2}n_9} + \cancel{C_{H_2}n_9} + \cancel{C_{H_2}n_9} - \cancel{C_{H_2}n_9} = 0. \qquad (8)
\end{aligned}
$$

[6]It is not shown due to space limitations.

[7]Since all variables are Boolean, x^n is replaced by x in calculations.

$$
\begin{aligned}
&SP := 32Z_5 + 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (4A_2 + 2A_1 + A_0) \times (4B_2 + 2B_1 + B_0)\\
&\xrightarrow{Z_5 = 1 - n_A} SP_1 := 32(1 - n_A) + 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (4A_2 + 2A_1 + A_0) \times (4B_2 + 2B_1 + B_0)\\
&\qquad\qquad = 32 - 32n_A + 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (4A_2 + 2A_1 + A_0) \times (4B_2 + 2B_1 + B_0)\\
&\xrightarrow{n_A = n_C - n_C C_{H_5}} SP_2 := 32 - 32n_C + 32n_C C_{H_5} + 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (4A_2 + 2A_1 + A_0) \times (4B_2 + 2B_1 + B_0)\\
&\ldots\\
&\xrightarrow{n_o = C_{F_1} S_{H_2}} SP_{17} := 32C_{H_5} + 16S_{H_5} + 8C_{H_6} + 8C_{F_1} + 8S_{H_2} + 4S_{H_6} + 2Z_1 + Z_0 \underbrace{-32C_{H_5}S_{H_5}S_{H_2}C_{F_1} - 32C_{H_5}S_{H_5}S_{H_2}C_{H_6} - 32C_{H_5}S_{H_5}C_{H_6}C_{F_1}}_{\text{3 monomials (12 variables)}}\\
&\qquad\qquad - (4A_2 + 2A_1 + A_0) \times (4B_2 + 2B_1 + B_0)\\
&\xrightarrow{S_{H_5} = 1 - n_x} SP_{18} := 32C_{H_5} + 16 - 16n_x + 8C_{H_6} + 8C_{F_1} + 8S_{H_2} + 4S_{H_6} + 2Z_1 + Z_0 \underbrace{-32C_{H_5}S_{H_2}C_{F_1} + 32C_{H_5}n_x S_{H_2}C_{F_1} - 32C_{H_5}S_{H_2}C_{H_6} + 32C_{H_5}S_{H_5}n_x C_{H_6} - 32C_{H_5}C_{H_6}C_{F_1} + 32C_{H_5}n_x C_{H_6}C_{F_1}}_{\text{6 monomials (21 variables)}}\\
&\qquad\qquad - (4A_2 + 2A_1 + A_0) \times (4B_2 + 2B_1 + B_0)\\
&\ldots\\
&\xrightarrow{n_z = C_{H_2} - C_{H_2} n_9} SP_{21} := 32n_t + 16n_y + 16C_{H_2} + 8C_{H_6} + 8C_{F_1} + 8S_{H_2} + 4S_{H_6} + 2Z_1 + Z_0 - 16C_{H_2}n_9 - 16n_y C_{H_2} + 16n_y C_{H_2} n_9\\
&\qquad \underbrace{-32n_t n_y S_{H_2}C_{F_1} - 32n_t C_{H_2} S_{H_2} C_{F_1} + 32n_t n_y C_{H_2} S_{H_2} C_{F_1} - 32n_t n_y C_{H_2} n_9 S_{H_2} C_{F_1} + 32n_t C_{H_2} n_9 S_{H_2} C_{F_1} + \cdots + 32n_t n_y C_{H_2} C_{H_6} C_{F_1} - 32n_t n_y C_{H_2} n_9 C_{H_6} C_{F_1}}_{\text{15 monomials (72 variables)}}\\
&\qquad - (4A_2 + 2A_1 + A_0) \times (4B_2 + 2B_1 + B_0)\\
&\ldots\\
&\xrightarrow{n_t = C_{H_2} n_9} \mathbf{SP_{24} := 16n_9 + 16C_{H_2} + 8C_{H_6} + 8C_{F_1} + 8S_{H_2} + 4S_{H_6} + 2Z_1 + Z_0 - (4A_2 + 2A_1 + A_0) \times (4B_2 + 2B_1 + B_0)}\\
&\ldots
\end{aligned}
$$

Fig. 4. Backward rewriting of 3-bit nontrivial multiplier.

This is in line with the following observation: C_{H_5} and S_{H_5} are the outputs of H_5. As it is impossible to have both outputs of a HA "1" at the same time, the product $C_{H_5}S_{H_5}$ is always equal to zero. In summary, this is the reason why the red monomials finally vanish in SP_{24}.

Now, We Give an Answer to 2): As just discussed, all red monomials in SP_{14} contain the product $C_{H_5}S_{H_5}$. Traversing back all substitution steps (i.e., moving in the direction of the outputs on the AIG), this product originates from the product $n_C C_{H_5}$ formed via the substitution of $n_A = 1 - n_C C_{H_5}$ as can be seen in SP_1. Interpreting this observation on the AIG means that there are two paths[8] starting from the two HA outputs (here, C_{H_5} and S_{H_5}) and these paths finally converge to a node (here n_A, node before output Z_5).

Overall, we conclude from this illustrating example that the origin of vanishing monomials is an AIG node where HA outputs converge, while the cancelation happens much later only after substituting the HA node polynomials.

In the next section, we provide the underlying theory of vanishing monomials. We also show that the vanishing monomials can be handled efficiently such that the size of the current polynomial SP_i does not grow dramatically during backward rewriting.

B. Basic Theory of Vanishing Monomials

We now generalize the observation from the illustrating example of the previous section. Therefore, we formulate the following theorem.

Theorem 2: Assume that x and y are two AIG nodes representing the outputs of an HA. The product of x and y appears during backward rewriting of a multiplier, if at least one path from x and one path from y converge to an AIG node n_C, and the product of n_C inputs is not canceled out in calculations.

Proof: Fig. 5(a) shows two paths starting from the HA outputs x and y and converge to the node n_C. The first path starting from x is a chain of AIG nodes $n_1, n_2, \ldots, n_i, n_C$. The second path starting from y consists of $n'_1, n'_2, \ldots, n'_j, n_C$. The edges connecting these nodes in the chains might be normal or complemented. Based on (5), we know that the polynomial of a 2-input AIG node contains the product of its inputs. Therefore,

[8] Path 1: C_{H_5} and n_A; Path 2: S_{H_5}, n_I, n_H, n_F, n_C, and n_A.

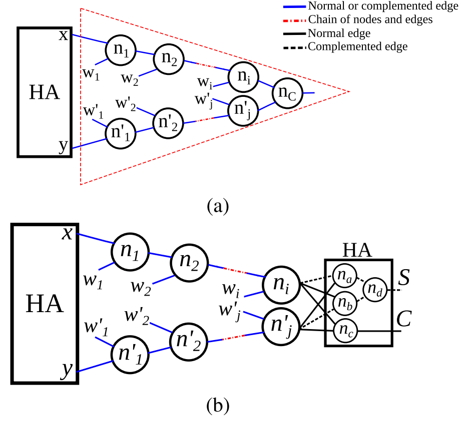


Fig. 5. Converging cone. (a) General case. (b) Converging to an HA.

the polynomial of n_C can be written as

$$
n_C = f\left(n_i, n'_j\right) + c\ n_i n'_j
\begin{cases}
n_C = n_i \wedge n'_j \rightarrow f\left(n_i, n'_j\right) = 0, & c = 1\\
n_C = \neg n_i \wedge n'_j \rightarrow f\left(n_i, n'_j\right) = n'_j, & c = -1\\
n_C = n_i \wedge \neg n'_j \rightarrow f\left(n_i, n'_j\right) = n_i, & c = -1\\
n_C = \neg n_i \wedge \neg n'_j \rightarrow f\left(n_i, n'_j\right) = 1 - n_i - n'_j, & c = 1.
\end{cases}
\tag{9}
$$

The functions that describe n_i and n'_j based on $x, y, w_1, w_2, \ldots, w_i, w'_1, w'_2, \ldots, w'_j$ are obtained after substituting the polynomials of the nodes located on the paths. We get after the substitutions

$$
\begin{aligned}
n_i &= f'(w_1, w_2, \ldots, w_i, x)\\
n'_j &= f''\left(w'_1, w'_2, \ldots, w'_j, y\right).
\end{aligned}
\tag{10}
$$

Based on (9) and (10), we conclude

$$
\begin{aligned}
n_C &= f\left(n_i, n'_j\right) + cf'(w_1, w_2, \ldots, w_i, x) f''\left(w'_1, w'_2, \ldots, w'_j, y\right)\\
&= f\left(n_i, n'_j\right)\\
&\quad + \underbrace{cxyT'_1 + cxyT'_2 + \cdots + cxyT'_r + cT_1 + cT_2 + \cdots + cT_s}_{\text{comes from the product } n_i n'_j}
\end{aligned}
\tag{11}
$$

where $cxyT'_h$ denotes the terms containing the product of x and y. Note that the xy product is generated as a result of multiplying two polynomials: one depending on x and the other one depending on y. After extracting the polynomial of n_C in (11), we now look on the backward rewriting process, i.e., we investigate the result of substituting n_C in current polynomials SP_i. Assume that the current polynomial SP_i before substituting n_C with the node polynomial is

$$SP_i = n_C X'_1 + n_C X'_2 + \cdots + n_C X'_l + X_1 + X_2 + \cdots + X_q \tag{12}$$

where $n_C X'_i$ denotes the terms containing n_C. Now, we distinguish between two cases, which might happen after substituting n_C in (12).

1) *Product of n_C Inputs, i.e., $n_i n'_j$, Is Canceled Out Early in Calculations:* Since the product $n_i n'_j$ is not contained in SP_i anymore, we can conclude that the product xy is not generated in the next steps of backward rewriting. For example, assume that n_C is a part of another HA (see Fig. 5(b)). Also, assume that S and $2C$ are two terms in SP_i while the rest of the terms are denoted by X_i. After substituting the polynomials of the HA nodes, the result is

$$\begin{aligned}
SP_i &= S + 2C + X_i + \cdots + X_q \\
SP_i &\xrightarrow{S} SP_{i+1} = 1 - n_d + 2C + X_i + \cdots + X_q \\
SP_{i+1} &\xrightarrow{C} SP_{i+2} = 1 - n_d + 2n_c + X_i + \cdots + X_q \\
SP_{i+2} &\xrightarrow{n_d} SP_{i+3} = n_a + n_b - n_a n_b + 2n_c \\
&\qquad + X_i + \cdots + X_q \\
SP_{i+3} &\xrightarrow{n_c} SP_{i+4} = n_a + n_b - n_a n_b + 2n_i n'_j \\
&\qquad + X_i + \cdots + X_q \\
SP_{i+4} &\xrightarrow{n_a} SP_{i+5} = n'_j - n_i n'_j + n_b - n'_j n_b + n_i n'_j n_b \\
&\qquad + 2n_i n'_j + X_i + \cdots + X_q \\
SP_{i+5} &\xrightarrow{n_b} SP_{i+6} = n'_j + n_i n'_j + n_i - n_i n'_j - n_i n'_j + n_i n'_j \\
&\qquad + n_i n'_j - n_i n'_j + X_i + \cdots + X_q \\
&= n_i + n'_j + X_i + \cdots + X_q.
\end{aligned} \tag{13}$$

As the product $n_i n'_j$ does not exist in SP_{i+5}, thus xy is not generated later during backward rewriting.

2) *Product of n_C Inputs, i.e., $n_i n'_j$, Remains in Calculations:* The product xy appears in the upcoming steps of backward rewriting as shown in (11). ■

Based on this theorem, we make the following definitions.

Definition 4: Let n_C be an AIG node fulfilling Theorem 2. Then, n_C is called a *converging node*.

Definition 5: Let n_C be a converging node. Then, the monomials containing the product of HA's outputs originating from n_C are *vanishing monomials* as they are reduced to zero after the HA's nodes substitution.

For managing the size of the current polynomial SP_i during backward rewriting, it is essential to prevent the inclusion of vanishing monomials since for nontrivial multipliers explosion occurs. Hence, the goal is to determine a vanishing-free polynomial representation for each converging node. In order to do this, we first look for the cones starting from a converging node and ending in the related HA outputs. Such a cone is called converging node cone (CNC) in the rest of this article

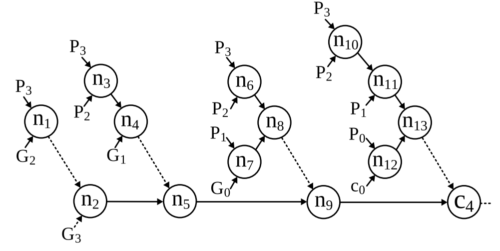


Fig. 6. Partial AIG representation of a 4-bit CLA.

[see also the red area in Fig. 5(a)]. Then, we locally extract the converging node polynomial based on the CNC inputs and remove all vanishing monomials. As a result, global backward rewriting becomes vanishing free and large nontrivial multipliers can be verified. Note that local vanishing removal is independent of the circuit's function; thus, it is applicable to both correct and buggy multipliers.

Before explaining the algorithm to remove vanishing monomials, we illustrate the relation between vanishing monomials and different multiplier architectures in the next section. In particular, we explain which stage of the multiplier is responsible for generating vanishing monomials. This helps us to narrow down the search space for finding CNCs and remove vanishing monomials efficiently.

C. Vanishing Monomials and Multiplier Architecture

Conducting several experiments on different multiplier architectures shows that vanishing monomials cause an explosion in multipliers using complex carry propagation hardware. This hardware is widely used in the third stage of the multiplier (i.e., FSA) to reduce the propagation delay of trivial ripple carry adders [20]. As a result, at the cost of some growth in the area, the multiplier becomes faster. Carry look-ahead adder (CLA) and parallel prefix adders (e.g., Kogge-Stone, Ladner-Fischer, and Han-Carlson) are among the architectures using complex carry propagation hardware. We now investigate a 4-bit CLA and show why many vanishing monomials are generated when it is used in the third stage of a multiplier.

Example 6: The Boolean formulation of a 4-bit CLA is

$$\begin{aligned}
G_i &= x_i \wedge y_i \\
P_i &= x_i \oplus y_i \\
c_1 &= G_0 \vee (c_0 \wedge P_0) \\
c_2 &= G_1 \vee (G_0 \wedge P_1) \vee (c_0 \wedge P_0 \wedge P_1) \\
c_3 &= G_2 \vee (G_1 \wedge P_2) \vee (G_0 \wedge P_1 \wedge P_2) \\
&\quad \vee (c_0 \wedge P_0 \wedge P_1 \wedge P_2) \\
c_4 &= G_3 \vee (G_2 \wedge P_3) \vee (G_1 \wedge P_2 \wedge P_3) \\
&\quad \vee (G_0 \wedge P_1 \wedge P_2 \wedge P_3) \\
&\quad \vee (c_0 \wedge P_0 \wedge P_1 \wedge P_2 \wedge P_3)
\end{aligned} \tag{14}$$

where x_i and y_i are ith bits of the first and second inputs, and c_i is the final carry. The AIG nodes dedicated to compute c_4 in the 4-bit CLA are shown in Fig. 6. If the Boolean formulation is transformed to the polynomial form, it consists of 31 monomials. However, 26 monomials contain the product of P_i and G_i. As P_i and G_i are outputs of an HA, based on (8), this product equals zero (i.e., $P_i G_i = 0$). Therefore,

all 26 monomials are reduced to zero and vanish from the calculation after substituting the HA nodes polynomials.

The generation of vanishing monomials during the substitution of node polynomials in the CLA can be justified by Theorem 2: P_i and G_i, which are outputs of an HA, converge to c_i, and the product of c_i inputs is not canceled out in the calculations. For example, in Fig. 6, the signal pairs (G_3, P_3), (G_2, P_2), and (G_1, P_1) converge to the node c_4. Thus, the product of P_i and G_i appears during backward rewriting.

The paths from HA outputs to converging nodes are usually long in multipliers using complex carry propagation hardware. Therefore, it takes several steps of substitution to reach the HA outputs during global backward rewriting. The generated vanishing monomials remain in the calculations in all these steps and cause an explosion in the number of monomials. As a result, finding CNCs and determining vanishing-free polynomials for each converging node is essential to avoid explosion during global backward rewriting. However, detecting CNCs is not possible without identifying HAs as all converging paths start from HA outputs. In the next section, we explain how reverse engineering helps us to efficiently identify HAs as well as other atomic blocks.

VI. Atomic Blocks in SCA

In this section, we introduce atomic blocks and showcase the advantages of identifying them for SCA-based verification.

A. Definitions

Definition 6: An *atomic block* is a basic building block for a multiplier, which gets n one-bit binary inputs with the same bit positions,[9] and computes their sum as m one-bit binary outputs. The typical atomic blocks with 2, 3, and 5 inputs are HA, FA, and *compressor* (CM). The corresponding word-level relations are

$$\begin{aligned} &\text{HA}(\text{in}: X, Y \quad \text{out}: C, S) \quad \Rightarrow \quad 2C + S = X + Y \\ &\text{FA}(\text{in}: X, Y, Z \quad \text{out}: C, S) \quad \Rightarrow \quad 2C + S = X + Y + Z \\ &\text{CM}(\text{in}: X, Y, Z, W, C_{\text{in}} \quad \text{out}: Co, C, S) \\ &\quad \Rightarrow \quad 2Co + 2C + S = X + Y + Z + W + C_{\text{in}}. \end{aligned} \tag{15}$$

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, and CM).

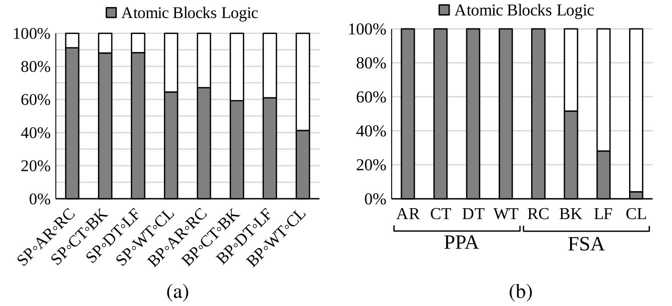
Definition 7: A specific multiplier architecture consisting of the stages $[\alpha \circ \beta \circ \gamma]$ is implemented by using atomic blocks and/or extra logic per stage. For trivial multipliers, the PPA stage β and the FSA stage γ are only made of HA and FA atomic blocks. For nontrivial multipliers, all kinds of atomic blocks plus highly parallel extra logic combining these blocks are allowed for all stages [20].

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

B. Advantages of Atomic Blocks for SCA

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

[9]Assuming $A_{N-1}A_{N-2}\ldots A_0$ and $B_{M-1}B_{M-2}\ldots B_0$ as two binary numbers, A_i and B_i have the same bit positions.



SP: Simple PPG, **BP**: Booth PPG

AR: Array, **CT**: Compressor Tree, **WT**: Wallace Tree, **DT**: Dadda Tree

RC: Ripple Carry, **BK**: Brent-Kung, **LF**: Ladner-Fischer, **CL**: Carry-Look ahead

Fig. 7. Atomic blocks ratio in 64×64 multipliers. (a) Multiplier types. (b) Stage architectures.

Detect CNCs: As discussed in Section V, detecting CNCs and removing vanishing monomials are crucial to avoid explosion during backward rewriting. An important step before the CNC detection is the identification of all HAs in the design as a CNC always starts from the HA outputs. The AIG representation of an HA is not unique. Using different atomic block libraries or applying optimization techniques results in HAs with different node numbers. However, the function of an HA (i.e., the relation between outputs and inputs) never changes. Thus, a reverse engineering technique to identify all atomic blocks (including HAs) independent of their implementation is necessary to guarantee the detection of CNCs.

Limit the Search Space for Vanishing Removal: In Section V-C, we explained that the explosion during backward rewriting occurs in multipliers using complex carry propagation hardware in the third stage. A part of the logic dedicated to this hardware always remains as the extra logic after reverse engineering. On the other hand, the second stage of the multiplier, which contains the largest number of AIG nodes, is fully made of atomic blocks. As a result, only a small part of the circuit, which cannot be identified as atomic blocks, i.e., the extra logic, is responsible for generating vanishing monomials.

Fig. 7(a) shows the ratio of atomic blocks logic to the entire logic in the different multiplier architectures after reverse engineering. Despite the fact that this ratio changes with respect to the design architecture, on average, atomic blocks constitute 70% of a multiplier. In addition, Fig. 7(b) depicts the atomic blocks ratio for the different PPA and FSA architectures. The results confirm that the PPA stage of many multipliers is completely made of atomic blocks. In contrast, the FSA stage of the multipliers using complex carry propagation hardware (e.g., BK, LF, and CL) is a mixture of atomic blocks and extra logic and their ratio varies based on the architecture.

Overall, reverse engineering allows limiting 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.

Speed Up Global Backward Rewriting: As (15) indicates, there is always a compact algebraic relation between inputs and outputs of an atomic block independent of their realization at the gate level. This algebraic relation can be shown as

$$f(\text{outputs}) = g(\text{inputs}) \tag{16}$$

Algorithm 1 REVSCA-2.0

Input: Multiplier AIG G
Output: TRUE if the circuit is correct, and FALSE otherwise

```
 1: SP ← CreateSP(G)
 2: AB, N ← ReverseEngineering(G)      ▷ AB is set of atomic blocks, N is set of extra nodes
 3: CN ← FindCNCs(N, filter_HAs(AB))                    ▷ CN is set of CNCs
 4: CF ← FindFFCs(G, N, CN, AB)               ▷ CF is set of fanout-free cones
 5: C ← CN ∪ CF
 6: F ← ExtractVanishingFreePolys(C)           ▷ F is set of cone polynomials
 7: r ← GlobalBackwardRewriting(SP, F, AB)                ▷ r is the remainder
 8: if r == 0 then
 9:     return TRUE
10: else
11:     return FALSE
```

where f(outputs) and g(inputs) are functions based on the output and input signals, respectively. Therefore, if f(outputs) appears in SP_i during backward rewriting, it can be substituted with g(inputs) instantly. With respect to the fact that a large part of a design is constructed with atomic blocks (see Fig. 7(a)), detecting atomic blocks will speed up the global backward rewriting considerably.

Example 7: Assume C and S are the outputs, and X, Y, and Z are the inputs of an FA. If $2C+S$ appears in SP_i during the backward rewriting, it can be substituted by $X+Y+Z$. As a result, we skip the substitution of the FA node polynomials and speed up the whole backward rewriting process.

VII. REVSCA-2.0

In this section, we first give a top-level overview of our SCA-verifier REVSCA-2.0. Then, we explain reverse engineering and CNC detection techniques used in REVSCA-2.0.

A. Top-Level Overview

To alleviate the vanishing monomials explosion problem during backward rewriting, we propose our new SCA-based verification method REVSCA-2.0.

In our proposed method, first SP is generated. Then, all atomic blocks are identified using reverse engineering. Subsequently, all CNCs starting from the HAs' outputs are detected and the polynomial for each CNC is extracted by the substitution of the node polynomials in the cone. The CNC polynomial determines the output of the cone (i.e., output of the converging node) based on its inputs. We know that a vanishing monomial contains the product of HA's outputs, and these outputs are the inputs of CNCs. Therefore, the vanishing monomials appear in the extracted CNC polynomials. Local removal of vanishing monomials from these polynomials leads to a set of vanishing-free polynomials. Now, global backward rewriting can be performed by substituting vanishing-free polynomials in SP_i without the appearance of any new vanishing monomial.

Algorithm 1 shows the pseudocode of REVSCA-2.0. In the first step, SP is created based on the input and output bit width of the multiplier (Line 1). Then, the atomic blocks are identified using a dedicated reverse engineering technique (Line 2). The CNCs are extracted based on the identified HAs [filter_HAs(AB)] and the set of extra nodes from the reverse engineering phase (Line 3). The rest of the nodes, which are not part of any atomic blocks or CNCs, is grouped based on the fanout-free regions as it increases the chance of monomial cancelation during global backward rewriting (Line 4). These cones are called *fanout-free cones* [3], [17]. In the next step, the polynomial for each cone is extracted by substitution, and the vanishing monomials are locally removed (Line 6). Finally, global backward rewriting is performed by substituting extracted polynomials in SP_i (Line 7). If the resulting remainder equals zero, the circuit is correct, otherwise, it is buggy (Line 8–Line 11).

In the next three sections, we explain reverse engineering, CNC detection, and local vanishing monomial removal in detail.

B. Reverse Engineering

In this section, we propose our dedicated reverse engineering method to identify atomic blocks in multipliers. First, we collect the truth tables of atomic blocks in a library,[10] which has to be done only once. Then, we extract cuts in the AIG representation of a multiplier and check whether the output vector of each cut matches one of the output vectors in a truth table in our library. If we find a set of cuts with common inputs, whose output vectors match the truth table of an atomic block, we have identified an atomic block. Truth tables can be computed efficiently during cut enumeration for cuts with up to 16 inputs.[11] Moreover, they require an acceptable amount of memory. Thus, they are preferred to other symbolic representations such as BDDs. In the following, we explain the two steps of atomic block identification in detail.

Atomic Blocks Specification Library: First, we have to specify the mathematical functions of the atomic blocks and collect them in a library.

Assume that $f_i(x_1, x_2, \ldots, x_n)$ is the Boolean function for the ith output of an atomic block, and $x_1, x_2, \ldots, x_n$ are the atomic block inputs. The library for the atomic block should contain all the functions in NPN class [25] of f_i, i.e., all the functions generated by swapping and complementing $x_1, x_2, \ldots, x_n$. Thus, the first step to create the library is the extraction of the NPN class for each atomic block output. The atomic block inputs are symmetric and have the same bit positions, so swapping does not create new functions. As a result, the only transformation that leads to the generation of new functions for the NPN class is complementing. After extracting the NPN class, the truth table for each function in the class is stored in the library. By following this principle, the complete set of truth tables for HAs and FAs can be obtained. We use notation T_x to refer to the vector in column x of a truth table.

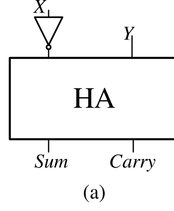
Example 8: The Boolean functions and the NPN classes for the two outputs of an HA are as follows:

$$\begin{aligned}\text{Sum} = X \oplus Y &\xrightarrow{\text{NPN class}} X \oplus Y\\ &\quad \neg X \oplus Y,\ X \oplus \neg Y,\ \neg X \oplus \neg Y\\ \text{Carry} = X \wedge Y &\xrightarrow{\text{NPN class}} X \wedge Y\\ &\quad \neg X \wedge Y,\ X \wedge \neg Y,\ \neg X \wedge \neg Y. \end{aligned} \tag{17}$$

If the first input of the HA is complemented [see Fig. 8(a)], the Boolean functions for the Sum and Carry are $\neg X \oplus Y$ and $\neg X \wedge Y$, respectively. Thus, the truth table contains two output vectors $T_{\text{Sum}} = 1001$ and $T_{\text{Carry}} = 0010$ based on Fig. 8(b).

[10] A truth table has several output vectors showing the value of the outputs for different input combinations, e.g., the truth table of an HA consists of two output vectors: one for *Sum* and another one for *Carry*.

[11] Cuts have a maximum of five inputs in our atomic block identification phase.



(a)

$T_{Sum} = 1001$ $T_{Carry} = 0010$

$\neg X$	Y	*Sum*	*Carry*
0	1	1	0
0	0	0	0
1	1	0	1
1	0	1	0

(b)

Fig. 8. HA with one input complement. (a) Structure. (b) 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

Fig. 9. CM truth table.

The story for CM is different. The challenge originates from the fact that there are outputs with the same bit position. For example, this holds for the CM with outputs S, C, and Co, where C and Co have the same bit position (see CM word-level description in (15) where C and Co has the same coefficient). As a result, the value of these two outputs can be swapped for a certain input combination without changing the function of CM. This would lead to the generation of a large number of truth tables.

Example 9: Fig. 9 shows the basic truth table (without complementing inputs) of a CM. Some rows have been omitted in the middle of the table. As C and Co have the same bit position, they can always be swapped. If the Boolean values of C and Co are not equal (red cells in Fig. 9), swapping them results in a completely new truth table. As in total there are 20 nonequal values of C and Co in the truth table of Fig. 9, $2^{20} = 1048576$ new truth tables are generated by swapping these values. To avoid dealing with millions of truth tables, we use the arbitrary values X_i in T_C, and its complement $\overline{X_i}$ in T_{Co}, where the ith value of T_C and T_{Co} is different. For example, in Fig. 9, T_C and T_{Co} can be encoded as

$$
\begin{aligned}
T_C &= 111X_3 1X_5X_6X_7 \dots X_{24}X_{25}X_{26}0X_{28}000 \\
T_{Co} &= 111\overline{X_3}1\overline{X_5X_6}X_7 \dots \overline{X_{24}X_{25}X_{26}}0\overline{X_{28}}000.
\end{aligned} \tag{18}
$$

The encoded values of T_C and T_{Co} in (18) cover all 2^{20} possible truth tables.

Finally, all the obtained truth tables of atomic blocks are stored in the atomic blocks library (ABLib).

Algorithm 2 Atomic Blocks Identification

Input: Multiplier AIG G, Set of output vectors $ST_1, \dots, ST_m$ from ABLib, Number of atomic block inputs n
Output: List of identified atomic blocks AB
1: $C \leftarrow$ 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: $SC \leftarrow$ Find the cuts with common inputs in $PC_0, PC_1, \dots, PC_m$
7: $AB \leftarrow$ Merge the cuts with common inputs in SC
8: **return** AB

Identifying Atomic Blocks: After creating ABLib, the next step is the identification of atomic blocks in the multiplier. Algorithm 2 presents the general algorithm for identifying atomic blocks with n inputs and m outputs using ABLib. The algorithm finds all cuts whose output vectors match one of the output vectors in a truth table in ABLib. If a set of cuts with common inputs has exactly the same output vectors as an atomic block truth table in ABLib, this set can be considered as an atomic block. The input of the algorithm is the AIG G of a multiplier, the set of possible vectors for each output $ST_0, \dots, ST_m$ 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 (see Definition 1) are computed on the AIG and stored in C (see Line 1). Then, the output vectors of the cuts are checked to see whether there is a cut c_i whose output vector is the member of one of the output vector sets ST_j. If yes, i.e., the function of c_i is the same as the jth output of the atomic block, 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 common inputs (Line 6). Finally, the cuts with common inputs are merged since we have found an atomic block (Line 7).

Example 10: Consider the 2×2 multiplier of Fig. 1(b): $C_1 = \{n_5, n_6, n_8\}$ and $C_2 = \{n_7\}$ are among the extracted 2-input cuts. By computing the output vectors of these two cuts, it is realized that T_{C_1} and T_{C_2} are members of ST_S and ST_C, which are the set of possible vectors for *Sum* and *Carry* in ABLib, receptively. Moreover, C_1 and C_2 have the common inputs n_2 and n_3. Thus, merging them results in identifying the atomic block $B = \{n_5, n_6, n_8, n_7\}$ which is an HA.

The runtime for computing cuts depends on the number of cut inputs, 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% of the entire atomic blocks,[12] 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 CMs.

C. Detecting Converging Node Cones

Algorithm 3 shows the proposed algorithm for detecting CNCs after reverse engineering. The algorithm receives the HAs H and the extra logic N as inputs, and returns the set of CNCs as output. As discussed in Sections V-C and VI-B, the CNCs are the subsets of the extra nodes. Therefore, we limit the search space for finding CNCs to these extra nodes. First, for each HA in H, all paths in N starting from the

[12] We justified this number by several experiments.

Algorithm 3 Detecting CNCs

Input: Set of HAs H, Set of extra nodes N
Output: Set of converging node cones CN

```
1:  CN ← ∅
2:  for each h ∈ H do
3:      P_S ← Find all paths starting from h_Sum in N
4:      P_C ← Find all paths starting from h_Carry in N
5:      for each p_S ∈ P_S do
6:          for each p_C ∈ P_C do
7:              if p_S ∩ p_C ≠ ∅ then
8:                  g = First common member of p_S and p_C
9:                  CNC ← [(p_S ∪ p_C) − (p_S ∩ p_C)] ∪ {g}
10:                 CN ← CN ∪ {CNC}
11: CN ← Merge all CNCs in CN with the same or included converging node
12: return CN
```

Sum and *Carry* outputs are extracted (see Line 3–Line 4 in Algorithm 3). The end of a path is where the POs or inputs of an atomic block are reached. In fact, P_S and P_C contain all the possible node chains connecting the *Sum* and *Carry* outputs of HAs to POs or inputs of atomic blocks. Then, the paths in P_S and P_C are checked to find out if there are paths which intersect (Line 5–Line 7). If that is the case, the first common member (i.e., g) is a converging node as it is the first place where two paths from HA's outputs meet (Line 8). In order to determine the CNC for the corresponding converging node g, the union of two paths ($p_S \cup p_C$) is subtracted by their intersection ($p_S \cap p_C$) and g is added to the result to obtain all the nodes from HA's outputs to the converging node (Line 9–Line 10). This process is repeated for all HAs to obtain the complete set of CNCs. Finally, all the cones with the same converging nodes (and thus, the same outputs) and the cones whose converging nodes are included in other CNCs are merged as there should be only one cone with a specific output signal (Line 11). In other words, cones C_1 and C_2 should be merged if: 1) they have the same converging node or 2) converging node of C_1 (C_2) is a member of C_2 (C_1).

Example 11: Consider again the 3×3 nontrivial multiplier of Fig. 3. The two HAs H_4 and H_5 are responsible for generating CNCs as there are paths from the outputs of these HAs converging to a node. Based on Algorithm 3 , first the paths from the H_4 outputs are extracted: $p_1 = \{n_I, n_F, n_C, n_A\}$ and $p_2 = \{n_H, n_C, n_A\}$ are the paths starting from S_{H_4} and C_{H_4}, respectively. After calculating the intersection of these paths, we observe $p_1 \cap p_2 \neq \emptyset$. Thus, the first common member, i.e., n_C, is a converging node. Using the equation in Line 9 of Algorithm 3 results in the detection of the CNC $C_1 = \{n_C, n_H, n_F, n_I\}$. The members of a CNC are sorted based on the reverse topological order of the circuit. Hence, the first member of a CNC (n_C in C_1) is always the converging node. Additionally, $p_3 = \{n_r, n_G, n_E, n_B\}$ and $p_4 = \{n_G, n_E, n_B\}$ are two other paths starting from S_{H_4} and C_{H_4}. These paths converge to n_G; thus, after using Algorithm 3 , we get $C_2 = \{n_G, n_r\}$. The complete list of detected CNCs after applying Algorithm 3 are $C_1 = \{n_C, n_H, n_F, n_I\}$, $C_2 = \{n_G, n_r\}$, $C_3 = \{n_A, n_C, n_F, n_I\}$, and $C_4 = \{n_A, n_C, n_H\}$ where C_1 and C_2 are related to H_4, and C_3 and C_4 are related to H_5. The cones C_3 and C_4 have the same converging node n_A, moreover, the converging node of C_1, i.e., n_C, is a member of C_3 and C_4. Consequently, we can merge these three cones to obtain $C = C_1 \cup C_3 \cup C_4 = \{n_A, n_C, n_H, n_F, n_I\}$. The cones C and C_2 are the final outputs of Algorithm 3.

The rest of the extra nodes, which are not part of any atomic blocks or CNCs, is grouped as fanout-free cones. In Fig. 3, $CF_1 = \{n_B, n_E, n_D\}$ is a fanout-free cone.

D. Local Removal of Vanishing Monomials

After detecting CNCs, the polynomial for each cone is extracted by a local backward rewriting. If a monomial containing the product of HAs' outputs (i.e., vanishing monomial) appears during local backward rewriting, we remove the monomial instantly to avoid the generation of more vanishing monomials in the next steps. Finally, we have a set of polynomials, which are completely vanishing free.

Example 12: Consider $C = \{n_A, n_C, n_H, n_F, n_I\}$, which is the CNC of Fig. 3. The steps of local backward rewriting and the vanishing monomials removal are as follows:

$$\begin{aligned} n_A &= n_C - n_C C_{H_5} \\ &= 1 - n_F - n_H + n_F n_H - C_{H_5} + n_F C_{H_5} \\ &\quad + n_H C_{H_5} - n_F n_H C_{H_5} \\ &= 1 - n_F - C_{H_4} S_{H_5} + n_F C_{H_4} S_{H_5} - C_{H_5} \\ &\quad + n_F C_{H_5} + \cancel{C_{H_4} S_{H_5} C_{H_5}} - \cancel{n_F C_{H_4} S_{H_5} C_{H_5}} \\ &= 1 - n_I C_{H_3} - C_{H_4} S_{H_5} + n_I C_{H_3} C_{H_4} S_{H_5} \\ &\quad - C_{H_5} + n_I C_{H_3} C_{H_5} \\ &= 1 - S_{H_4} S_{H_5} C_{H_3} - C_{H_4} S_{H_5} + \cancel{S_{H_4} C_{H_3} C_{H_4} S_{H_5}} - C_{H_5} \\ &\quad + \cancel{S_{H_4} S_{H_5} C_{H_3} C_{H_5}} \\ &= \boxed{1 - S_{H_4} S_{H_5} C_{H_3} - C_{H_4} S_{H_5} - C_{H_5}}. \end{aligned} \tag{19}$$

The red monomials contain $S_{H_4} C_{H_4}$ or $S_{H_5} C_{H_5}$, which are the product of H_4 and H_5 outputs in Fig. 3, respectively. Therefore, they are canceled out immediately when they appear during local backward rewriting.

VIII. Experimental Results

We have implemented RevSCA-2.0 in C++. In order to extract cuts in the reverse engineering phase, we used the mockturtle library [26]. All experiments are performed on an Intel Xeon E3-1270 v3 with 3.50 GHz and 32 GByte of main memory. In order to evaluate the efficiency of RevSCA-2.0 in verification of different signed and unsigned multipliers, we consider a variety of architectures. Table I shows the used architectures and also their abbreviations in the three stages of a multiplier. To generate the multipliers up to 64×64 input sizes, we use the *arithmetic module generator* [27] known as AOKI, which supports a wide range of architectures. However, AOKI cannot generate multipliers bigger than 64×64. Therefore, we also employ our multiplier generator GenMul[13] [28] to create large multipliers with 128×128, 256×256, and 512×512 input sizes. We have also balanced or refactored the AIG of some multipliers (*balance* and *refactor* commands in abc [29]) to evaluate the effects of design alteration/optimization on the verification methods.

Table II reports the verification times for the different multipliers. The *time out* (T.O.) has been set to 48 h. *Not supported* (N.S.) indicates that the method does not support the verification of the benchmark. The first column of Table II *Benchmark* presents the type of the multiplier based on the stage architectures. The second column *size* denotes the size

[13] Available at http://www.sca-verification.org/genmul.

TABLE I
MULTIPLIER ARCHITECTURES AND ABBREVIATIONS

Stage 1 (PPG)	Simple PPG (**SP**)	Booth PPG (**BP**)						
Stage 2 (PPA)	Array (**AR**)	Overturned-stairs tree (**OS**)	Dadda tree (**DT**)	Wallace tree (**WT**)	Balanced delay tree (**BD**)	Compressor tree (**CT**)		
Stage 3 (FSA)	Ripple carry adder (**RC**)	Conditional sum adder(**CU**)	Ladner-Fischer adder (**LF**)	Carry look-ahead adder (**CL**)	Kogge-Stone adder (**KS**)	Brent-Kung adder (**BK**)	Block CL adder (**BL**)	Ripple-block CL (**RL**)

TABLE II
RUNTIMES OF VERIFYING MULTIPLIERS (SECONDS)

Benchmark	Size	Type	**PolyCleaner**					**State-of-the-art methods**						
			Reverse Engineering	Cone Detection	Local Van. Removal	Global Backw. Rewriting	Overall	Comm.	[3]	[5]	[6]	[16]	[17]	[18]
$SP\circ BD\circ KS$	16×16	unsigned	0.04	0.00	0.02	0.03	**0.09**	48.00	T.O.	T.O.	T.O.	T.O.	T.O.	0.10
$BP\circ DT\circ LF$	16×16	unsigned	0.06	0.00	0.01	0.06	0.13	53.00	T.O.	T.O.	T.O.	T.O.	2.89	**0.07**
$SP\circ AR\circ RC$	64×64	unsigned	0.69	0.01	0.00	2.61	3.31	T.O.	49.91	T.O.	T.O.	14.17	781.12	**0.74**
$SP\circ OS\circ CU$			1.19	0.02	0.02	7.10	8.33	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**0.95**
$SP\circ DT\circ LF$			0.78	0.01	0.02	4.27	5.09	T.O.	T.O.	T.O.	T.O.	T.O.	2,105.74	**1.14**
$SP\circ WT\circ CL$			2.00	0.35	2.84	11.35	16.53	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**4.20**
$SP\circ BD\circ KS$			1.06	0.04	4.18	6.78	12.05	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1.56**
$SP\circ CT\circ BK$			12.52	0.05	0.06	10.01	22.64	T.O.	T.O.	T.O.	T.O.	T.O.	1,726.67	**0.74**
$SP\circ AR\circ BL$			0.67	0.01	0.01	3.16	**3.86**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ OS\circ RL$			1.14	0.01	0.04	6.57	7.77	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**0.98**
$SP\circ AR\circ RC$	64×64 (Refactored)	unsigned	0.52	0.01	0.01	2.50	3.03	T.O.	T.O.	T.O.	T.O.	14.39	T.O.	**0.79**
$SP\circ OS\circ CU$			1.13	0.02	0.02	6.67	7.83	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**0.93**
$SP\circ DT\circ LF$			0.67	0.01	0.03	17.16	**17.86**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ WT\circ CL$			1.93	0.31	2.33	10.01	**14.57**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ BD\circ KS$			0.95	0.04	3.79	6.80	**11.58**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ CT\circ BK$			12.51	0.05	0.06	10.20	**22.82**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ AR\circ BL$			0.63	0.01	0.02	4.27	**4.93**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ OS\circ RL$			1.01	0.01	0.02	75.09	**76.13**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$BP\circ AR\circ RC$	64×64	unsigned	1.38	0.02	0.03	12.93	14.36	T.O.	37.18	T.O.	T.O.	T.O.	882.52	**0.91**
$BP\circ OS\circ CU$			1.56	0.03	0.04	22.09	23.73	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1.11**
$BP\circ DT\circ LF$			0.83	0.02	0.05	14.13	15.04	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1.15**
$BP\circ WT\circ CL$			2.39	0.38	3.14	16.79	22.71	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**5.10**
$BP\circ BD\circ KS$			1.29	0.04	4.39	13.32	19.05	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1.56**
$BP\circ CT\circ BK$			6.57	0.04	0.07	21.86	28.53	T.O.	T.O.	T.O.	T.O.	T.O.	1,729.33	**0.98**
$BP\circ AR\circ BL$			1.63	0.02	0.05	16.01	**17.72**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$BP\circ OS\circ RL$			1.34	0.02	0.07	19.09	20.51	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1.01**
$BP\circ AR\circ RC$	64×64 (Refactored)	unsigned	1.31	0.02	0.03	12.97	14.33	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**0.96**
$BP\circ OS\circ CU$			1.47	0.03	0.04	22.88	24.43	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1.10**
$BP\circ DT\circ LF$			0.78	0.02	0.06	46.90	**47.76**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$BP\circ WT\circ CL$			2.20	0.33	5.27	18.07	**25.88**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$BP\circ BD\circ KS$			1.24	0.05	3.92	13.38	**18.59**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$BP\circ CT\circ BK$			6.72	0.04	0.06	20.88	**27.70**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$BP\circ AR\circ BL$			1.59	0.02	0.05	16.35	**18.01**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$BP\circ OS\circ RL$			1.37	0.02	0.12	180.89	**182.40**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ WT\circ BK$	128×128	unsigned	7.54	0.06	0.04	134.58	142.22	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**7.13**
$SP\circ DT\circ LF$			3.71	0.07	0.08	149.74	153.60	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**7.22**
$SP\circ WT\circ BK$	128×128	signed	8.39	0.08	0.04	180.83	189.34	T.O.	N.S.	N.S.	N.S.	N.S.	N.S.	**10.32**
$SP\circ DT\circ LF$			4.40	0.08	0.08	217.65	222.22	T.O.	N.S.	N.S.	N.S.	N.S.	N.S.	**10.03**
$SP\circ WT\circ BK$	128×128 (Balanced)	unsigned	7.47	0.06	0.03	123.54	**131.11**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ DT\circ LF$			3.58	0.07	0.06	146.66	**150.37**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ WT\circ BK$	256×256	unsigned	72.15	0.32	0.16	3,700.59	3,773.21	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**67.84**
$SP\circ DT\circ LF$			27.84	0.35	0.32	5,593.94	5,622.45	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**66.96**
$SP\circ WT\circ BK$	256×256	signed	76.84	0.32	0.15	3,676.48	3,753.80	T.O.	N.S.	N.S.	N.S.	N.S.	N.S.	**87.40**
$SP\circ DT\circ LF$			34.83	0.36	0.32	6,059.79	6,095.31	T.O.	N.S.	N.S.	N.S.	N.S.	N.S.	**100.44**
$SP\circ WT\circ BK$	256×256 (Balanced)	unsigned	81.22	0.31	0.12	3,262.04	**3,343.69**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ DT\circ LF$			30.06	0.34	0.20	5,262.22	**5,292.81**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ WT\circ BK$	512×512	unsigned	960.27	1.60	0.56	66,530.90	67,493.30	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**640.90**
$SP\circ DT\circ LF$			480.85	1.73	1.29	113,774.00	114,257.87	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**790.12**
$SP\circ WT\circ BK$	512×512	signed	1,184.83	1.65	0.57	67,076.30	68,263.40	T.O.	N.S.	N.S.	N.S.	N.S.	N.S.	**879.82**
$SP\circ DT\circ LF$			740.61	1.76	1.30	114,750.00	115,493.00	T.O.	N.S.	N.S.	N.S.	N.S.	N.S.	**841.99**
$SP\circ WT\circ BK$	512×512 (Balanced)	unsigned	844.13	1.51	0.46	58,621.40	**59,467.50**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
$SP\circ DT\circ LF$			321.59	1.66	0.74	105,450.46	**105,774.45**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.

of the multiplier based on the two inputs' bit width. The third column *type* shows whether the multiplier is signed or unsigned. The runtime (in seconds) of our proposed method is reported in detail in the forth column REVSCA-2.0 consisting of five subcolumns: 1) *Reverse Engineering* reports the required time for extracting cuts in the AIG representation of a multiplier and then identifying atomic blocks; 2) *Cone Detection* refers to the time needed for detecting CNCs and fanout-free cones; 3) *Local Van. Removal* presents the consumed time for extracting the polynomial for each CNC and fanout-free cone and removing vanishing monomials; 4) *Global Backw. Rewriting* reports the time for the global backward rewriting phase. Finally, the overall runtime of our proposed method is presented in *Overall*; and 5) the fifth column *State-of-the-art methods* of Table II reports the runtimes of the state-of-the-art verification methods. This column consists of six subcolumns: *Comm.* refers to the runtime of the commercial formal verification tool OneSpin 360 DV. The remaining columns show the runtimes of the most recent SCA-based verification approaches.

It is evident in Table II that our proposed approach can verify all benchmarks including both unsigned and signed multipliers. The reverse engineering time in most of the cases is small compared to the overall verification time as the algorithm to extract cuts and our proposed algorithm to identify atomic blocks are very fast. The only

TABLE III
VERIFICATION DATA OF MULTIPLIERS

Benchmark	Size	Type	#Node	#Atomic	#Van.	#MaxPoly
$SP\circ BD\circ KS$	16×16	unsigned	3144	281	7716	444
$BP\circ DT\circ LF$	16×16	unsigned	2571	166	803	1186
$SP\circ AR\circ RC$	64×64	unsigned	48,128	4,032	0	4,224
$SP\circ OS\circ CU$			51,766	4,502	0	9,825
$SP\circ DT\circ LF$			48,808	4,038	2,249	7,256
$SP\circ WT\circ CL$			68,875	4,365	266,684	4,461
$SP\circ BD\circ KS$			50,756	4,313	613,454	5,607
$SP\circ CT\circ BK$			41,466	4,253	808	6,080
$SP\circ AR\circ BL$			48,212	4,079	96	4,589
$SP\circ OS\circ RL$			49,989	4,412	929	7,929
$BP\circ AR\circ RC$	64×64	unsigned	38,439	2,732	0	20,099
$BP\circ OS\circ CU$			39,798	2,667	0	25,803
$BP\circ DT\circ LF$			36,867	2,205	2,249	20,098
$BP\circ WT\circ CL$			57,684	2,513	280,604	12,530
$BP\circ BD\circ KS$			39,053	2,514	616,166	12,532
$BP\circ CT\circ BK$			33,172	2,433	815	20,101
$BP\circ AR\circ BL$			38,555	2,799	170	20,099
$BP\circ OS\circ RL$			38,050	2,577	909	20,100
$SP\circ WT\circ BK$	128×128	unsigned	166,938	17,366	1,623	22,406
$SP\circ DT\circ LF$			164,572	16,263	3,642	29,811
$SP\circ WT\circ BK$	128×128	signed	198,697	17,366	1,623	22,449
$SP\circ DT\circ LF$			196,322	16,263	3,642	29,811
$SP\circ WT\circ BK$	256×256	unsigned	663,505	67,974	3,376	81,460
$SP\circ DT\circ LF$			657,622	65,288	5,800	120,738
$SP\circ WT\circ BK$	256×256	signed	792,538	67,974	3,376	81,552
$SP\circ DT\circ LF$			786,653	65,288	5,800	120,738
$SP\circ WT\circ BK$	512×512	unsigned	2,641,643	267,985	6,851	296,667
$SP\circ DT\circ LF$			2,627,536	261,641	9,718	485,741
$SP\circ WT\circ BK$	512×512	signed	3,161,854	267,986	6,851	296,728
$SP\circ DT\circ LF$			3,147,734	261,641	9,718	485,741

exceptions are *SP∘CT∘BK* and *BP∘CT∘BK*, which require the extraction of cuts with five inputs as they contain compressors. Therefore, the reverse engineering time increases for these benchmarks. The most time-consuming verification phase is the global backward rewriting as it requires many calculations (e.g., polynomial substitution) on large polynomials.

On the other hand, the commercial tool only verifies multipliers up to 16×16, and it times out for the bigger benchmarks. The proposed SCA-based verification methods of [3], [5], [6], and [16] either cannot verify any benchmarks or only work on trivial multiplier architectures, i.e., *SP∘AR∘RC* and *BP∘AR∘RC*. The main reason is that these methods do not provide any solution to remove vanishing monomials early in the calculations to avoid explosion during global backward rewriting. The proposed method in [17] can verify some of the nontrivial multipliers as the authors presented a heuristic to detect and remove vanishing monomials. However, it is not robust as can be seen in column [17] and fails for most of the benchmarks. Moreover, it is drastically slower than our method.

The proposed method in [18] reports very good results in verification of nontrivial multipliers if the FSA can be detected. However, it fails to verify two AOKI benchmarks (i.e., *SP∘AR∘BL* and *BP∘AR∘BL*), as well as 18 (balanced or refactored) multipliers. Hence, the method is not robust against design alterations/optimizations as they usually destroy the clean boundaries between the multiplier stages. Nevertheless, it is promising to integrate the FSA detection principle in REVSCA-2.0 to speed up our approach.

Table III presents the verification data reported by REVSCA-2.0 for multipliers. The first, second, and third columns of the table show the architecture, size, and the type of the multiplier, respectively. The forth column *#node* reports the number of nodes in the AIG representation of the multiplier. The number of identified atomic blocks is presented in the fifth column *#Atomic*. The sixth column *#Van.* gives the total number of canceled vanishing monomials in the local vanishing removal phase. Finally, the seventh column *#MaxPoly* reports the maximum size of SP_i during global backward rewriting by counting the number of monomials.

The results in Table III confirm that REVSCA-2.0 can verify nontrivial multipliers with more than 3M AIG nodes, e.g., the signed 512×512 *SP∘WT∘BK* multiplier contains 3 161 854 nodes. The number of detected atomic blocks varies base on the size of the multiplier and its architecture. For example, the multipliers with radix-4 Booth encoding in the PPG stage have less atomic blocks compared to those which use the simple PPG. The reason is that the number of generated partial products is smaller in case of Booth encoding; thus, less atomic blocks are required to reduce these partial products. The total number of canceled vanishing monomials also varies based on the architecture. No vanishing monomial is generated during verification of trivial multipliers, e.g., *SP∘AR∘RC* and *BP∘AR∘RC*; therefore, the number of canceled vanishing monomials is zero. On the other hand, during the verification of 64×64 *BP∘WT∘CL* and *BP∘BD∘KS*, which are nontrivial multipliers, approximately $280K$ and $616K$ vanishing monomials are canceled, respectively.

IX. CONCLUSION AND FUTURE WORK

In this article, we presented the SCA-verifier REVSCA-2.0, which combines reverse engineering and local vanishing removal to prove the correctness of nontrivial million-gate multipliers. REVSCA-2.0 first identifies all atomic blocks including HAs. Then, based on an extended theory for the origin of vanishing monomials, it detects converging cone nodes starting from HAs and locally removes vanishing monomials. As a consequence, global backward rewriting becomes vanishing free and no explosion happens in the number of monomials during verification. The experiments using an extensive set of nontrivial million-gate multipliers demonstrated the efficiency of REVSCA-2.0 in comparison to the state-of-the-art techniques.

In our future research, we will focus on the verification of optimized multipliers. To achieve this goal, we plan to extend the dynamic backward rewriting approach of [30] to support highly optimized architectures.

REFERENCES

[1] D. Stoffel and W. Kunz, “Equivalence checking of arithmetic circuits on the arithmetic bit level,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 5, pp. 586–597, May 2004.

[2] 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,” *IEEE Trans. Comput.*, vol. 56, no. 10, pp. 1401–1414, Oct. 2007.

[3] F. Farahmandi and B. Alizadeh, “Gröbner basis based formal verification of large arithmetic circuits using Gaussian elimination and cone-based polynomial extraction,” *Microprocess. Microsyst.*, vol. 39, no. 2, pp. 83–96, 2015.

[4] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, “Formal verification of arithmetic circuits by function extraction,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 12, pp. 2131–2142, Dec. 2016.

[5] D. Ritirc, A. Biere, and M. Kauers, “Column-wise verification of multipliers using computer algebra,” in *Proc. Int. Conf. Formal Methods Comput.-Aided Design (FMCAD)*, Vienna, Austria, 2017, pp. 23–30.

[6] C. Yu, M. Ciesielski, and A. Mishchenko, “Fast algebraic rewriting based on and-inverter graphs,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 9, pp. 1907–1911, Sep. 2018.

[7] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler, "Equivalence checking using Gröbner bases," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design (FMCAD)*, Mountain View, CA, USA, 2016, pp. 169–176.

[8] A. Mahzoon, D. Große, and R. Drechsler, "Combining symbolic computer algebra and boolean satisfiability for automatic debugging and fixing of complex multipliers," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Hong Kong, 2018, pp. 351–356.

[9] F. Farahmandi and P. Mishra, "Automated test generation for debugging multiple bugs in arithmetic circuits," *IEEE Trans. Comput.*, vol. 68, no. 2, pp. 182–197, Feb. 2019.

[10] C. Scholl and A. Konrad, "Symbolic computer algebra and SAT based information forwarding for fully automatic divider verification," in *Proc. 57th ACM/IEEE Design Autom. Conf.*, San Francisco, CA, USA, 2020, pp. 1–6.

[11] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, "Verifying dividers using symbolic computer algebra and don't care optimization," in *Proc. Design Autom. Test Eur.*, 2021.

[12] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: Clean your polynomials before backward rewriting to verify million-gate multipliers," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, San Diego, CA, USA, 2018, pp. 1–8.

[13] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *Proc. 56th ACM/IEEE Design Autom. Conf.*, Las Vegas, NV, USA, 2019, pp. 1–6.

[14] M. Ciesielski, C. Yu, D. Liu, W. Brown, and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *Proc. Design Autom. Conf.*, San Francisco, CA, USA, 2015, pp. 1–6.

[15] D. Kaufmann, A. Biere, and M. Kauers, "Incremental column-wise verification of arithmetic circuits using computer algebra," *Formal Methods Syst. Design*, vol. 56, no. 1, pp. 22–54, 2020.

[16] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *Proc. Design Autom. Test Eur.*, Dresden, Germany, 2018, pp. 1556–1561.

[17] 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 *Proc. Design Autom. Test Eur.*, Dresden, Germany, 2016, pp. 1048–1053.

[18] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design (FMCAD)*, San Jose, CA, USA, 2019, pp. 28–36.

[19] R. Zimmermann, "Binary adder architectures for cell-based VLSI and their synthesis," Ph.D. dissertation, Dept. Doctor Techn. Sci., Swiss Federal Inst. Technol., Zürich, Switzerland, 1997.

[20] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. Natick, MA, USA: A. K. Peters, Ltd., 2001.

[21] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *Proc. ACM/SIGDA 6th Int. Symp. Field Program. Gate Arrays*, 1998, pp. 35–42.

[22] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 240–253, Feb. 2007.

[23] D. A. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms*. Cham, Switzerland: Springer, 1997.

[24] M. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding algebraic rewriting for arithmetic circuit verification: A bit-flow model," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 6, pp. 1346–1357, Jun. 2020.

[25] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *Proc. Int. Conf. Field-Programm. Technol.*, Kyoto, Japan, 2013, pp. 310–313.

[26] M. Soeken *et al.*, "The EPFL logic synthesis libraries," May 2018. [Online]. Available: arXiv:1805.05121.

[27] (2019). *Arithmetic Module Generator Based on ACG*. [Online]. Available: https://www.ecsis.riec.tohoku.ac.jp/topics/amg/i-amg

[28] A. Mahzoon, D. Große, and R. Drechsler, "GENMul: Generating architecturally complex multipliers to challenge formal verification tools," in *Recent Findings in Boolean Techniques*, R. Drechsler and D. Große, Eds. Cham, Switzerland: Springer, 2021, pp. 177–191.

[29] (2018). *ABC: A System for Sequential Synthesis and Verification*. [Online]. Available: https://people.eecs.berkeley.edu/∼alanmi/abc/

[30] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *Proc. Design Autom. Test Eur.*, Grenoble, France, 2020, pp. 544–549.

Alireza Mahzoon (Graduate Student Member, IEEE) received the master's degree in electrical engineering from the University of Tehran, Tehran, Iran, in 2016. He is currently pursuing the Ph.D. degree with the Research Group of Computer Architecture, University of Bremen, Bremen, Germany.

He published several papers on international conferences, such as DATE, ICCAD, and DAC. His current research interests include formal verification and debugging of arithmetic circuits with a focus on highly complex and industrial multipliers.

Mr. Mahzoon received the best paper award at ICCAD 2018.

Daniel Große (Senior Member, IEEE) received the Dr.-Ing. degree in computer science from the University of Bremen, Bremen, Germany, in 2008.

He remained as a Postdoctoral Researcher with the Group of Computer Architecture, University of Bremen, Bremen, Germany. In 2010, he was a substitute Professor of Computer Architecture with Albert-Ludwigs University, Freiburg im Breisgau, Germany. From 2013 to 2014, he was the CEO of the EDA start-up solvertec focusing on automated debugging techniques. Since 2015, he has been a Senior Researcher with the University of Bremen and the German Research Center for Artificial Intelligence, and the Scientific Coordinator of the Graduate School System Design, funded within the German Excellence Initiative. Since July 2020, he has been a Full Professor with the Johannes Kepler University Linz, Linz, Austria, where he is the Head of the Institute for Complex Systems (ICS). His current research interests include verification, virtual prototyping, debugging, synthesis, and RISC-V. He published over 150 papers in peer-reviewed journals and conferences in the above areas.

Prof. Große received best paper awards (FDL 2007, DVCon Europe 2018, ICCAD 2018, and FDL 2020) as well as business-related awards (IKT Innovativ Award 2013, Weconomy Award 2013, and Embedded Award 2014). He served in program committees of numerous conferences, including ASP-DAC, DAC, DATE, ICCAD, CODES+ISSS, FDL, and MEMOCODE. He is an Allied Member of the Accellera Systems Initiative in the SystemC Verification Working Group.

Rolf Drechsler (Fellow, IEEE) received the Diploma and Dr.Phil.nat. degrees in computer science from J. W. Goethe University Frankfurt am Main, Frankfurt, Germany, in 1992 and 1995, respectively.

He was with the Institute of Computer Science, Albert-Ludwigs University, Freiburg im Breisgau, Germany, from 1995 to 2000, and the Corporate Technology Department, Siemens AG, Munich, Germany, from 2000 to 2001. Since 2001, he has been with the University of Bremen, Bremen, Germany, where he is currently a Full Professor and the Head of the Group for Computer Architecture, Institute of Computer Science. In 2011, he became the Director of the Cyber-Physical Systems Group, German Research Center for Artificial Intelligence, Bremen, Germany. His current research interests include the development and design of data structures and algorithms with a focus on circuit and system design.

Prof. Drechsler was a recipient of best paper awards at HVC in 2006, FDL in 2007, 2010, and 2020, DDECS in 2010, DSD in 2020, and ICCAD in 2013 and 2018. He was a member of Program Committees of numerous conferences, including DAC, ICCAD, DATE, ASP-DAC, FDL, MEMOCODE, and FMCAD, the Symposium Chair of ISMVL 1999 and 2014 and ETS 2018, and the Topic Chair for "Formal Verification" at DATE 2004, DATE 2005, DAC 2010, and DAC 2011 and 2018. He was the General Chair of the ETS 2018 and the Program Chair of ICCAD 2020. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION, and further journals.