Automatic Correction of Arithmetic Circuits in the Presence of Multiple Bugs by Groebner Basis Modification

NEGAR AGHAPOUR SABBAGH, School of Electrical and Computer Engineering, University of Tehran, College of Engineering, Tehran, Iran (the Islamic Republic of)
BIJAN ALIZADEH, School of Electrical and Computer Engineering, University of Tehran, College of Engineering, Tehran, Iran (the Islamic Republic of) and School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

One promising approach to verify large arithmetic circuits is making use of Symbolic Computer Algebra (SCA), where the circuit and the specification are translated to a set of polynomials, and the verification is performed by the ideal membership testing. Here, the main problem is the monomial explosion for buggy arithmetic circuits, which makes obtaining the word-level remainder become unfeasible. So, automatic correction of such circuits remains a significant challenge. Our proposed correction method partitions the circuit based on primary output bits and modifies the related Groebner basis based on the given suspicious gates, which makes it independent of the word-level remainder. We have applied our method to various signed and unsigned multipliers, with various sizes and numbers of suspicious and buggy gates. The results show that the proposed method corrects the bugs without area overhead. Moreover, it is able to correct the buggy circuit on average 51.9× and 45.72× faster in comparison with the state-of-the-art correction techniques, having single and multiple bugs, respectively.

CCS Concepts: • **Hardware → Semi-formal verification**;

Additional Key Words and Phrases: Arithmetic circuit, Ideal basis replacement, Generator, Groebner basis, Polynomial, Remainder

ACM Reference Format:
Negar Aghapour Sabbagh and Bijan Alizadeh. 2024. Automatic Correction of Arithmetic Circuits in the Presence of Multiple Bugs by Groebner Basis Modification. *ACM Trans. Des. Autom. Electron. Syst.* 29, 5, Article 76 (August 2024), 19 pages. https://doi.org/10.1145/3672559

1 Introduction

The findings from the study in [1, 2] indicate that, in 2020, design engineers spent a significant amount of their time in verification (51% of FPGA-based design project time and 56% of IC/ASIC design project time). There are many flaws, causing bugs escape in IC/ASIC and FPGA designs, like crosstalk, clocking, power consumption, and so on. The main reason is logic or functional flaws,

Authors' Contact Information: Negar Aghapour Sabbagh, School of Electrical and Computer Engineering, University of Tehran, College of Engineering, Tehran, Tehran, Iran (the Islamic Republic of); e-mail: n.aghapour@ut.ac.ir; Bijan Alizadeh, School of Electrical and Computer Engineering, University of Tehran, College of Engineering, Tehran, Tehran, Iran (the Islamic Republic of) and School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Tehran, Iran; e-mail: b.alizadeh@ut.ac.ir.

ACM 1084-4309/2024/08-ART76
https://doi.org/10.1145/3672559

which have an almost share of 52% and 49% among various categories of design flaws of FPGA and ASIC designs, respectively, in 2020. The main root of the functional flaws is a design error, which happened in almost 75% and 81% of IC/ASIC and FPGA designs, in 2020, respectively. As mentioned in this research, the design error issue is getting worsening, over time. The other common roots are changing in the specification and incomplete or incorrect specification. Moreover, almost 46% and 47% of the verification time in FPGA and ASIC designs, respectively, are spent on the debugging process to localize and therefore rectify potential bugs.

On the other hand, the extensive usage of arithmetic circuits in computation-intensive tasks such as cryptography, signal processing, and machine learning makes designers propose a large variety of different arithmetic circuit architectures to meet the power, speed, and area constraints. These architectures are very complex and therefore prone to design errors. So, designers are seeking for efficient and automated formal verification and rectification approaches.

Automatic formal verification and debugging approaches are categorized into four groups: (1) **Decision Diagrams** (**DDs**) such as **Binary Decision Diagrams** (**BDDs**) and **Binary Moment Diagrams** (**BMDs**) [3], (2) Boolean **Satisfiability** (**SAT**) and **Satisfiability Modulo Theories** (**SMT**), (3) reverse engineering and (4) **Symbolic Computer Algebra** (**SCA**). Verifying complex arithmetic circuits based on DDs needs exponential space complexity. Some improved methods, like [4], verify multipliers up to 64 bits. Although word-level decision diagrams, like **Taylor Expansion Diagrams** (**TEDs**) [5] and **Modular Horner Expansion Diagrams** (**M-HED**) [6] which also support modular computations, are proposed, they are not efficient for applying to bit-level arithmetic circuits. SAT-based methods model the arithmetic circuit as propositional logic [7–9] and verify it by making use of SAT or SMT solvers [10]. Note that SAT-based verification methods have not scaled well for arithmetic circuit verification, due to the time complexity. In addition, reverse engineering-based methods verify combinational arithmetic circuits by extracting predefined subcircuits like half/full adders [11]. Such methods suffer from memory explosion issues.

Among all formal verification methods, SCA-based methods have shown good results for verifying integer arithmetic circuits [12–15], especially multipliers [16–19], and also Galois field arithmetic circuits [20–24]. They benefit from the polynomial model to verify a given circuit. First of all, the circuit and its specification are represented by a set of polynomials F_{CIR} and P_S. All linear combination of polynomials in F_{CIR} is called the circuit ideal I_{CIR}. The circuit is verified if P_S is a member of I_{CIR} and tested by the remainder of the serial division of P_S over F_{CIR}. Note that the uniqueness of the remainder is a challenge in SCA-based approaches, which is guaranteed by using the Groebner basis algorithm as will be discussed in Section II. Note that using special data structures like **Zero-Suppressed Decision Diagrams** (**ZDD**) [25] and BMD [26] improves the memory requirement of SCA-based methods. Although SCA-based methods performed very well in the verification, automatic correction of large arithmetic circuits still faces major challenges.

When the functionality of the design is not correct, the designer needs to find the exact location of the bugs and then correct them. Although some research works have proposed SCA-based debugging methods to localize buggy gates [27, 28] and some other methods try to minimize the suspicious gates [29, 30], automatic correction of arithmetic circuits has not been satisfactorily addressed. Note that some works have addressed the rectification of random-logic circuits by replacing each suspicious gate with a corrected one which is chosen by a SAT-based method [31, 32].

The correction method proposed in [33] uses the non-zero remainder and proves whether the circuit can be corrected by modifying the given single buggy gate or not. Correction of circuits with multiple bugs is a more challenging situation. The method in [34] uses the non-zero remainder and automatically tries to correct the circuits by adding a corrector sub-circuit. This method may have excessive area overhead and increases the critical path delay. Other methods [35, 36] resynthesize buggy gates based on computed rectification function. In these methods, it is assumed that all of buggy gates are fan-in-independent.

All the above-mentioned methods are based on the word-level remainder while obtaining this remainder for large arithmetic circuits is not feasible due to the monomial explosion. So, proposing a word-level remainder-independent correction method would be promising. Hence, the main contributions of this work are as follows:

— Formulating the problem of arithmetic circuit correction by modifying the Groebner basis of I_{CIR}. To the best of our knowledge, such a formulation is presented for the first time.
— Partitioning the arithmetic circuit based on primary output bits, making the correction become remainder-independent.
— Translating the specification polynomial to a set of binary polynomials, according to each output bit, helps utilize binary polynomials for all logic gates and reduces the required memory and time during the verification process.
— Demonstrating its application by performing correction of multi-bug large arithmetic circuits without area overhead.

Note that in this article, the remainder means the word-level remainder, which is obtained by SCA-based verification methods. The rest of the article is organized as follows. Section 2 represents the concept of the ideal and the Groebner basis reduction. In Section 3, the SCA-based verification method is explained. Section 4 presents the proposed correction method. In Section 5, we report the experimental results, and Section 6 concludes the article.

2 Preliminaries

A polynomial is a linear combination of monomials over ring $k[x_1,\ x_2, \ldots,\ x_n]$, where $x_1,\ x_2, \ldots, x_n$ are variables. It means that the linear combination coefficients are members of k, where k can be the collection of real numbers $\mathbb{R}$, the collection of integer numbers $\mathbb{Z}$, and so on. [37]. Also, a monomial is in the form of $x_1^{\alpha_1}.\ x_2^{\alpha_2}.\ldots.x_n^{\alpha_n}$, where the n-tuple of exponents $\alpha = (\alpha_1,\ \alpha_2, \ldots, \alpha_n) \in\ \mathbb{Z}^n_{\geq 0}$ (all powers are non-negative integer numbers.). Also, a monomial can be written as x^α. To have an ordered polynomial with pre-defined variable ordering in monomials, we need to define monomial ordering in polynomials. Any ordering $>$ on the $\mathbb{Z}^n_{\geq 0}$, used for exponents of the monomial, gives a special monomial ordering. In other words, it is written that $x^\alpha > x^\beta$ when $\alpha > \beta$ according to this ordering. For example, a well-known lexicographic order says $\alpha >_{lex} \beta$ if the leftmost non-zero entry in the difference vector $\alpha - \beta \in \mathbb{Z}^n$ is positive. For example, $xy^2z^4 >_{lex} xz^5$ with variable ordering $x > y > z$, where the leftmost non-zero entry of the difference vector $(1, 2, 4) - (1, 0, 5) = (0, 2, -1)$ is positive. The first monomial in the ordered polynomial is the leading monomial (LM), and its coefficient is the leading coefficient (LC). The leading term is defined as LT = LC×LM.

An ideal, I, is a subset of $k[x_1,\ x_2, \ldots, x_n]$, satisfying three conditions: (i) $0 \in I$, (ii) $f, g \in I \Rightarrow f + g \in I$, (iii) $f \in I,\ h \in k[x_1,\ x_2, \ldots, x_n] \Rightarrow hf \in I$. Suppose polynomial set $F = \{f_1,\ f_2, \ldots, f_s\}$ is given. All linear combinations of F, denoted by $\langle f_1, \ldots, f_s\rangle$ in Equation (1), satisfies three ideal conditions. So, $\langle f_1, \ldots, f_s\rangle$ is an ideal and the polynomials $f_1,\ f_2, \ldots, f_s$ are called the ideal generators.

$$\langle f_1, \ldots, f_s\rangle \ = \ \left\{ \sum_{i=1}^{s} h_i f_i \ :\ h_1, \ldots, h_s \in\ k\left[x_1,\ x_2, \ldots, x_n\right] \right\}. \tag{1}$$

One of the basic questions in algebraic geometry is the ideal membership testing, asking if the given polynomial p is a member of the given ideal $I = \langle f_1, \ldots, f_s\rangle$. If so, p can be rewritten as a linear combination of ideal generators; otherwise, it is in the form of $p = \sum_{i=1}^{s} h_i f_i + REM$,

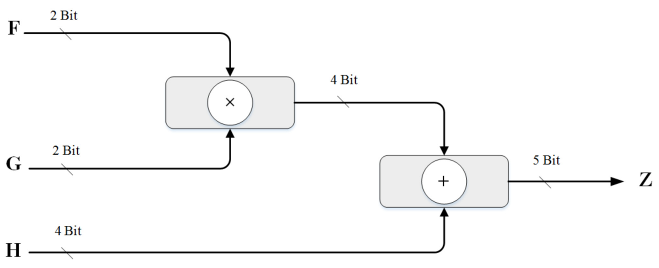

Fig. 1. A 2-bit MAC block diagram for $Z = (F \times G) + H$.

Table 1. Polynomials Related to Logical Gates [12, 21]

Gate Types	Boolean function	Related Polynomial	Related Binary Polynomial
AND	$y = in_1 \wedge in_2$	$f_{and} : y - in_1 in_2$	$f_{and\_b} : y + in_1 in_2$
NAND	$y = in_1 \bar{\wedge} in_2$	$f_{nand} : y + in_1 in_2 - 1$	$f_{nand\_b} : y + in_1 in_2 + 1$
OR	$y = in_1 \vee in_2$	$f_{or} : y - in_1 - in_2 + in_1 in_2$	$f_{or\_b} : y + in_1 + in_2 + in_1 in_2$
NOR	$y = in_1 \bar{\vee} in_2$	$f_{nor} : y + in_1 + in_2 - in_1 in_2 - 1$	$f_{nor\_b} : y + in_1 + in_2 + in_1 in_2 + 1$
XOR	$y = in_1 \oplus in_2$	$f_{xor} : y - in_1 - in_2 + 2 in_1 in_2$	$f_{xor\_b} : y + in_1 + in_2$
XNOR	$y = in_1 \bar{\oplus} in_2$	$f_{xnor} : y + in_1 + in_2 - 2 in_1 in_2 - 1$	$f_{xnor\_b} : y + in_1 + in_2 + 1$

where there is no term in non-zero remainder REM, divisible by any $LT(f_i)$. A simple way to obtain the remainder REM is to perform serial division of p over generators denoted as $p \xrightarrow{F} REM$ ($p \xrightarrow{f_1} g_1 \xrightarrow{f_2} g_2 \xrightarrow{f_3} \cdots \xrightarrow{f_s} REM$). Note that changing the order of generators with a fixed monomial ordering may result in different remainders (for example $p \xrightarrow{f_2} g_1' \xrightarrow{f_1} g_2' \xrightarrow{f_3} \cdots \xrightarrow{f_s} REM'$). A special set of ideal generators, called Groebner basis, should be constructed to obtain a unique remainder. The Groebner basis is a finite subset $\mathrm{GB} = \{\mathrm{gb}_1, \ldots, \mathrm{gb}_t\}$ of I if $< \mathrm{LT}(\mathrm{gb}_1), \ldots, \mathrm{LT}(\mathrm{gb}_t) > = < \mathrm{LT(I)} >$, where $< \mathrm{LT(I)} >$ is another ideal generated by all leading terms of I [37]. When the leading monomials of each ideal generator pair are relatively prime, the given generator set is a Groebner basis of the ideal [37]. The meaning of leading monomials of f and g being prime to each other is the **least common multiple (LCM)** of leading monomials is $LCM(LM(f), \ LM(g)) = LM(f) \times LM(g)$.

3 SCA-based Formal Verification of Arithmetic Circuits

The SCA-based verification methods **convert the given circuit** (***CIR***) and the **specification** (***Spec***) to polynomial sets F_{CIR} and P_S, respectively, and then check whether P_S is a member of the ideal $I_{CIR} = \langle F_{CIR} \rangle$ or not. A zero-remainder means P_S is a member of I_{CIR} and the given arithmetic circuit is correct. Otherwise, the given circuit is buggy. So, utilizing the Groebner basis to obtain a unique remainder is required. Choosing the **Reverse Topological Term Order (RTTO)** as variable ordering and the lexicographical monomial ordering makes the obtained F_{CIR} become the Groebner basis of I_{CIR}.

As a simple example, let us consider a 2-bit **multiply and accumulate (MAC)** block diagram implementing $Z = F \times G + H$, shown in Figure 1. The specification polynomial of the unsigned MAC circuit would be $P_S : 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 = (2F_1 + F_0) \times (2G_1 + G_0) + 8H_3 + 4H_2 + 2H_1 + H_0)$ which is rewritten as $\mathrm{P_S}$: $16Z_4 + 8\mathrm{Z}_3 + 4\mathrm{Z}_2 + 2\mathrm{Z}_1 + \mathrm{Z}_0 - ((2\mathrm{F}_1 + \mathrm{F}_0) \times (2\mathrm{G}_1 + \mathrm{G}_0) + (8H_3 + 4H_2 + 2H_1 + H_0)) = 0$. Figure 2 shows the gate-level circuit of unsigned MAC, with a buggy gate (g4), and related polynomials of each gate, based on Table 1, constructing $F_{CIR} = \{p_1, p_2, \ldots, p_{25}\}$. The RTTO variable ordering is represented in Figure 2, where the variable

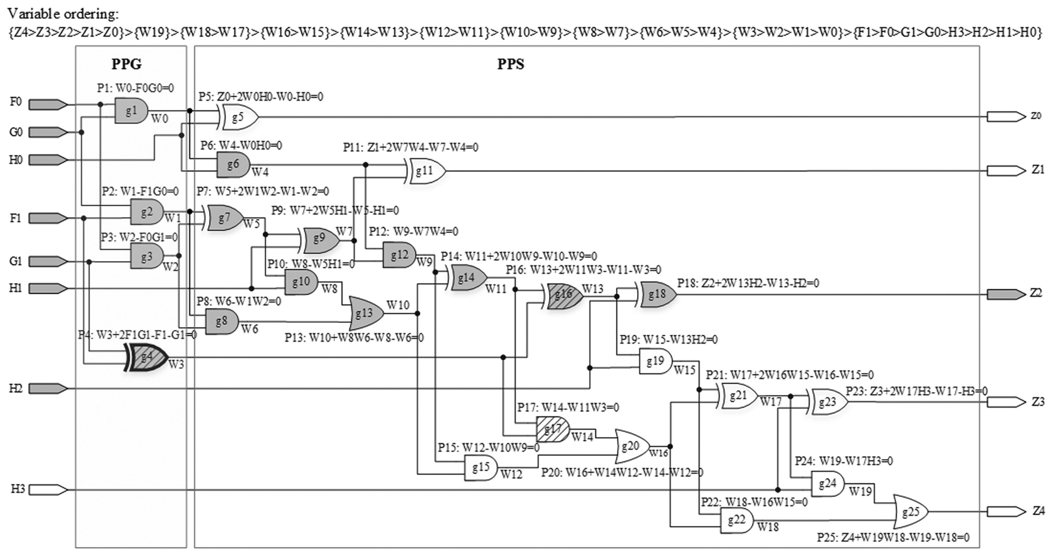

Fig. 2. A buggy gate-level circuit of 2-bit MAC, with one buggy gate (g_4), and three suspicious gates (g_4, g_{16}, and g_{17}).

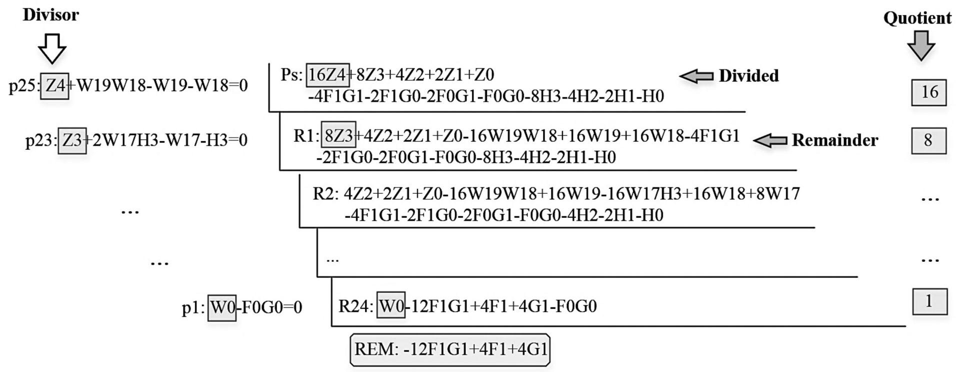
Divisor
Quotient
p25: Z4+W19W18-W19-W18=0
p23: Z3+2W17H3-W17-H3=0
...
...
p1: W0-F0G0=0
Ps: 16Z4+8Z3+4Z2+2Z1+Z0 -4F1G1-2F1G0-2F0G1-F0G0-8H3-4H2-2H1-H0
Divided
R1: 8Z3+4Z2+2Z1+Z0-16W19W18+16W19+16W18-4F1G1 -2F1G0-2F0G1-F0G0-8H3-4H2-2H1-H0
Remainder
R2: 4Z2+2Z1+Z0-16W19W18+16W19-16W17H3+16W18+8W17 -4F1G1-2F1G0-2F0G1-F0G0-4H2-2H1-H0
...
R24: W0-12F1G1+4F1+4G1-F0G0
REM: -12F1G1+4F1+4G1
16
8
...
...
1

Fig. 3. The Groebner basis reduction of P_s related to Figure 2.

ordering in each bracket is arbitrary. The ideal membership testing is done by serial division of P_S over I_{CIR} as shown in Figure 3.

The leading monomial of polynomials related to each gate is in the form of $1 \times mono$, where *mono* is the variable according to the output of the desired gate, as shown in Figure 2. So, the division process is translated to replacing each variable in the specification polynomial with the related polynomial of the corresponding gate, based on RTTO ordering, which is represented in Figure 3. The remainder of each division is divided into the next one, and the quotient is equal to the leading term of the divided, where the leading variable is eliminated. So, the last obtained remainder contains numbers and variables related to inputs. The final non-zero remainder of $P_S \xrightarrow{F_{CIR}} REM : -12F_1G_1 + 4F_1 + 4G_1$ shows that P_S is not a member of I_{CIR}, and therefore the circuit is buggy.

4 Proposed Correction Method

The non-zero remainder shows that $P_S \notin I_{CIR}$, meaning that I_{CIR} is not the correct ideal. So, the basic idea to automatically correct the circuit is to modify I_{CIR} in such a way that it covers P_S. For doing so, we need to modify the generator set of the ideal, and therefore, two sorts of ideal modification are defined: (1) Ideal basis addition [34] and (2) Ideal basis replacement. In the first modification proposed in [34], a new generator is added to the generator set of the ideal, and a new ideal is obtained, which covers new polynomials rather than the original one.

Definition 1 (Ideal Basis Replacement). Replacing one (or more) existing generator(s) f_i of $I = \langle f_1, \ldots, f_i, \ldots, f_s \rangle$ with f_i' obtains a new ideal $I_{rep} = \langle f_1, \ldots, f_i', \ldots, f_s \rangle$. I_{rep} can cover some polynomials that were not covered by I, and also, some polynomials covered by I might not be covered by I_{rep}.

This sort of modification is the basic idea of the proposed correction method, which tries to modify I_{CIR} based on Definition 1, where the modified $I_{CIR_{rep}}$ covers the specification polynomial ($P_S \xrightarrow{GB(I_{CIR_{rep}})} 0$). Here, it is assumed that the list of generators, which are candidates to be modified is known, and the proposed method tries all possible replacements for them. In other words, since each generator is a translation of one gate of the circuit, it is assumed that the list of suspicious gates is obtained by corresponding methods, like method [29, 30], and given to the proposed method. Algorithm 1 shows the proposed correction method, where the netlist of the buggy circuit (*CIR*), the specification polynomial (P_S), and a list of suspicious gates (*SUSP*) are inputs, and a list of legal replacements (*LegalRep*) for suspicious gates is the output. The proposed method partitions the arithmetic circuit into cones of the primary output bits and corrects each cone. The meaning of the cone, is a list of gates, starting from the specified gate and inserting its input gates into the list. Also, the input gates of each newly inserted gate are added to the list, iteratively, until achieving the primary inputs. As a simple example of an output cone, all gates placed in the cone of Z_2 are marked in Figure 2.

In Algorithm 1, at first, the number of outputs is calculated (line 1), and the corresponding polynomials of suspicious gates (*SUSP*) are constructed (line 2). The polynomials obtained here, are binary, written based on the last column of Table 1, which are denoted by $\_b$ at the end of the polynomial name. Line 3 computes all possible replacements for all suspicious gates. Although there are various types of flaws for buggy circuits, like gate misplacements, wrong connection, adding or removing invertors, and so on, most buggy circuits can be corrected by changing the type of buggy gate(s). In other words, the topologies of the buggy and the correct circuit are the same or very close to each other [38]. So, the proposed algorithm tries to correct this type of bug. So, all possible replacements, which is shown by $\mathbb{G}$ in line 3, are six types of each 2-input gate {XOR, XNOR, OR, NOR, AND, NAND}, in addition to all possible gate removement conditions, which are direct connection between the output of the suspicious gate to one of its inputs and also, the inversion of one of its inputs (NOT gate), as shown in Figure 4. So, replacements will be performed for all suspicious polynomial, not buggy ones, where $\mathbb{G}^{|SUSP|}$ would be an n-tuple of possible polynomials. The given arithmetic circuit is partitioned based on primary output bits, and the correction process is applied to each partition (lines 5 to 10). The proposed method tries to correct the cone of each output bit i, from the **least significant bit** (**LSB**) to the **most significant bit** (**MSB**), through three main phases: (1) constructing the polynomial ring of I_{CIR} and generating the cone ideal $I_{CONE}(i)$ (line 6), (2) calculating the corresponding bit-level specification $P_{S\_b}(i)$ and carry-out polynomials, shown by $c(i+1)$, (line 7), (3) solving the ideal basis replacement problem to cover $P_{S\_b}(i)$ by $GB(I_{CONE})$ and extracting the new legal replacements (line 8). In the following subsections, we discuss each phase in more detail.

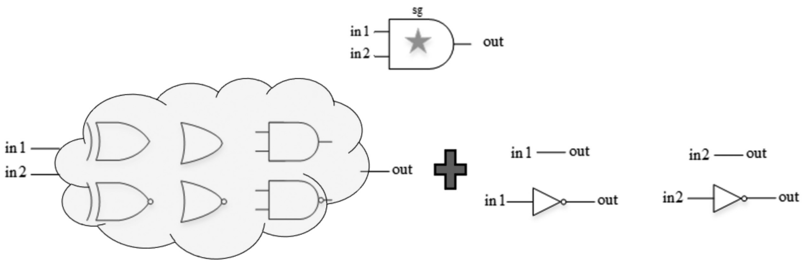

Fig. 4. Types of considered replacements.

ALGORITHM 1: Correction Algorithm based on Ideal Basis Replacement

```
Inputs: Buggy circuit (CIR), Specification ($P_S$), Suspicious gates(SUSP)
Output: Legal Replacements (LegalRep)
1:   $N \leftarrow$ NumberOfOutputBits (CIR);
2:   $F_{SUSP} \leftarrow$ ExtractBinaryPolynomial (SUSP);
3:   $LegalRep \leftarrow \mathbb{G}^{|SUSP|}$      //all possible replacement solutions
4:   $carry(0) \leftarrow 0$
5:   for $i$ in 0 to $N$-1 do
6:      $I_{CONE}(i) \leftarrow$ GenerateConeIdeal (CIR, $i$);
7:      $\{P_S(i), c(i+1)\} \leftarrow$ BitLevelSPEC ($P_S$, $c(i)$, $i$);
8:      $LegalRep \leftarrow$ IdealBasisRep ($P_S(i)$, $I_{CONE}(i)$, $F_{SUSP}$, $LegalRep$);
9:   end for
return LegalRep;
```

4.1 Cone Ideal Generation

The cone ideal generation phase is responsible for generating an ideal for the cone of the given output bit *i*, which gets the netlist of the circuit *CIR* and returns the related cone ideal. Algorithm 2 represents the GenerateConeIdeal function, which contains three steps to generate the cone ideal of the *i*th output: (1) specifying all gates located in the cone of *i*th output (line 1), (2) generating the corresponding polynomial ring, based on the variables related to the inputs and outputs of gates (line 2). Since our correction method is implemented in the PolyBori framework, the polynomial ring is constructed by calling the Generate Boolean Polynomial Ring function, (3) extracting the binary polynomial of each gate and inserting it into the defined ideal object as a new generator (lines 4 to 7), which is done by calling add generator function of the same framework. Note that partitioning the circuit into output cones helps utilize binary polynomials, which are represented by ZDDs in the PolyBori framework.

For example, applying Algorithm 2 to the MAC circuit of Figure 2 for the output Z_2, generates the ideal of the cone, by detecting all gates located in the intended cone as highlighted in Figure 2. Then it generates the Boolean polynomial ring by the variables related to the highlighted gates, using the RTTO variable ordering. Here, the related variables are $\{Z_2, W_{13}, W_{11}, \ldots, F_1, F_0, G_1, G_0, H_2, H_1, H_0\}$. Finally, the binary polynomial of each gate is generated and added to the ideal, as a new generator. In other words, the cone ideal of Z_2 will be $I_{CONE}(2) = <P_{18\_b}, P_{16\_b}, P_{14\_b}, P_{13\_b}, P_{12\_b}, P_{10\_b}, P_{9\_b}, P_{8\_b}, P_{7\_b}, P_{6\_b}, P_{4\_b}, P_{3\_b}, P_{2\_b}, P_{1\_b}>$, where P_{x_b} represents the generators of $I_{CONE}(2)$, which is the binary form of the polynomial P_x. For example, $P_{18\_b}: \ Z_2 + W_{13} + H_2 = 0$ and $P_{13\_b}: \ W_{10} + W_8W_6 + W_8 + W_6 = 0$, which are the binary form of P_{18} and P_{13}, respectively.

ALGORITHM 2: Cone Ideal Generation

```
Inputs: Buggy circuit (CIR), Output bit (i)
Output: Cone Ideal of the given output ($I_{CONE}$)
1:    GTs ← DetectGatesInCone (CIR, i);
2:    BR ← GenerateBooleanPolynomialRing (GTs);      // polynomial ring
3:    $I_{CONE}$ ← GenerateIdeal (BR);                    //instance an ideal object
4:    for g ∈ GTs do:
5:        p_b ← ExtractBinaryPolynomial (g);
6:        $I_{CONE}$.add_generator (p_b);
7:    end for
return $I_{CONE}$;
```

4.2 Constructing *i*th Bit Level Specification

All arithmetic circuits consist of the **partial product generation** (**PPG**) and **partial product summation** (**PPS**) stages. So, the specification polynomial $P_{S\_b}(i)$ of arithmetic circuits is the summation of partial products, where the partial products can be the input terms or the multiplication of them. Assuming this fact, the proposed method extracts the bit-level specification polynomials P_{S_b} from P_S in three steps. (1) The partial products are separated into some sets based on the *i*th bit position of the coefficient. (2) Elements of each set are grouped, and the sum and carry-out polynomials are calculated, where the carry polynomials are placed in the proper locations based on the number system. In other words, the carry polynomials are transferred to the higher order. This step is similar to the PPS stage of arithmetic circuits, which is called PPS calculation here. (3) the final sum polynomial is converted to bit-level specification.

To clarify these steps, let us consider a 2-bit MAC circuit with P_S: $16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - 4F_1G_1 - 2F_1G_0 - 2F_0G_1 - F_0G_0 - 8H_3 - 4H_2 - 2H_1 - H_0 = 0$. At first, each partial product term is placed in the correct set based on its coefficient of 2^i, represented as the first step in Figure 5. Next, starting from set 0 to 4, terms in each set are categorized into three-member groups, and the related sum and carry-out polynomials are calculated for each group. Note that it is possible to have a one- or two-member group next to three-member ones. The sum and carry-out polynomials of each three-member group $\{m_1, m_2, m_3\}$ are $s = m_1 + m_2 + m_3$ and $c = m_1m_2 + m_2m_3 + m_3m_1$, respectively. Also, they are $s = m_1 + m_2$ and $c = m_1m_2$ for each two-member group $\{m_1, m_2\}$ and $s = m_1$ and $c = 0$ for the one-member group $\{m_1\}$. Note that the first row of Figure 5 (2^0 bit position) represents a two-member group, while the last two rows (2^3 and 2^4 bit positions) have three-member and one-member groups, respectively. The other rows (2^1 and 2^2 bit positions) have a three-member and a two-member groups. All carry-out polynomials are inserted into the next row (higher bit-position), and a set of sum polynomials is assumed as a new partial polynomial set of the current row. We continue this procedure to obtain a one-member summation set. The third step transforms this polynomial to form a bit-level specification by adding the related output variable, Z_i (see the third step in Figure 5). Note that $P_{S\_b}(i)$ is obtained from P_S and is independent of the implementation.

In the case of signed arithmetic circuits, all negative partial products must be sign extended. The other steps are the same as the unsigned ones. For example, expanding the specification polynomial for the signed MAC in figure 2 would be $P_S : -16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - ((-2F_1 + F_0) \times (-2G_1 + G_0) + (-8H_3 + 4H_2 + 2H_1 + H_0)) = 0$. Rewriting the specification polynomial, according to the coefficients is rewritten as follows. $P_S : -16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (8(-H_3) + 4(F_1G_1) + 2(-F_1G_0 - F_0G_1 + H_1) + (F_0G_0 + H_0) = 0$. So, the partial products are shown in the first box of Figure 6, where the modification of each negative term is represented in a second box in Figure 6.

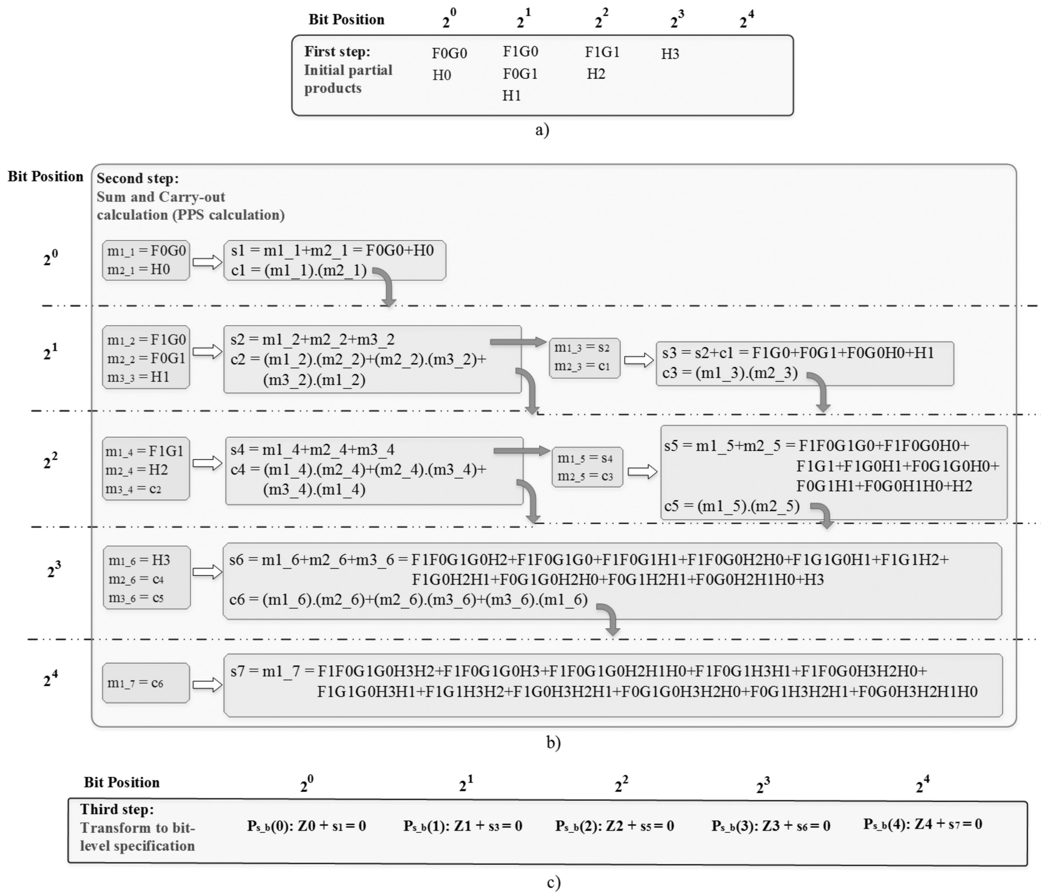
Bit Position
2^0
2^1
2^2
2^3
2^4
First step: Initial partial products
F0G0
H0
F1G0
F0G1
H1
F1G1
H2
H3
a)
Bit Position
Second step: Sum and Carry-out calculation (PPS calculation)
m1_1 = F0G0
m2_1 = H0
s1 = m1_1+m2_1 = F0G0+H0
c1 = (m1_1).(m2_1)
m1_2 = F1G0
m2_2 = F0G1
m3_3 = H1
s2 = m1_2+m2_2+m3_2
c2 = (m1_2).(m2_2)+(m2_2).(m3_2)+(m3_2).(m1_2)
m1_3 = s2
m2_3 = c1
s3 = s2+c1 = F1G0+F0G1+F0G0H0+H1
c3 = (m1_3).(m2_3)
m1_4 = F1G1
m2_4 = H2
m3_4 = c2
s4 = m1_4+m2_4+m3_4
c4 = (m1_4).(m2_4)+(m2_4).(m3_4)+(m3_4).(m1_4)
m1_5 = s4
m2_5 = c3
s5 = m1_5+m2_5 = F1F0G1G0+F1F0G0H0+F1G1+F1G0H1+F0G1G0H0+F0G1H1+F0G0H1H0+H2
c5 = (m1_5).(m2_5)
m1_6 = H3
m2_6 = c4
m3_6 = c5
s6 = m1_6+m2_6+m3_6 = F1F0G1G0H2+F1F0G1G0+F1F0G1H1+F1F0G0H2H0+F1G1G0H1+F1G1H2+F1G0H2H1+F0G1G0H2H0+F0G1H2H1+F0G0H2H1H0+H3
c6 = (m1_6).(m2_6)+(m2_6).(m3_6)+(m3_6).(m1_6)
m1_7 = c6
s7 = m1_7 = F1F0G1G0H3H2+F1F0G1G0H3+F1F0G1G0H2H1H0+F1F0G1H3H1+F1F0G0H3H2H0+F1G1G0H3H1+F1G1H3H2+F1G0H3H2H1+F0G1G0H3H2H0+F0G1H3H2H1+F0G0H3H2H1H0
b)
Bit Position
Third step: Transform to bit-level specification
Ps_b(0): Z0 + s1 = 0
Ps_b(1): Z1 + s3 = 0
Ps_b(2): Z2 + s5 = 0
Ps_b(3): Z3 + s6 = 0
Ps_b(4): Z4 + s7 = 0
c)

Fig. 5. Constructing the bit level specification of a 2-bit MAC. (a) Initial Partial Product list. (b) Partial Product Summation calculation. (c) Convert to bit-level specification.

Proposition. *The bit-level specification obtained by the proposed method is the same as the assumed correct circuit with the given specification polynomial.*

Proof. The bit-level specification is obtained with the assumption of implementing SPEC by the arithmetic circuit containing half- and full-adders with ripple carry adder structure in the PPS stage. Since the word-level specification polynomial of the given structure and the assumed ripple carry adder structure are the same, makes that the bit-level polynomials are equal. To prove this proposition, assume that the bit-level functionality for each output in the correct circuit (without any bug) is $\{P_{S\_b}(N\text{-}1),\ P_{S\_b}(N\text{-}2),\ \ldots\ P_{S\_b}(1),\ P_{S\_b}(0)\}$. On the other hand, the obtained bit-level specification from $\mathrm{P_S}$ is $\{P'_{S\_b}(N\text{-}1),\ P'_{S\_b}(N\text{-}2),\ \ldots\ P'_{S\_b}(1),\ P'_{S\_b}(0)\}$. Note that, each $\mathrm{P_{S\_b}}(i)$ and $\mathrm{P'_{S\_b}}(i)$ are binary polynomials. So, the obtained specification polynomials are $P_S : 2^{n-1}P_{S\_b}(n-1) + 2^{n-2}P_{S\_b}(n-2) + \ldots + 2P_{S\_b}(1) + P_{S\_b}(0) = 0$ and $P'_S : 2^{n-1}P'_{S\_b}(n-1) + 2^{n-2}P'_{S\_b}(n-2) + \ldots + 2P'_{S\_b}(1) + P'_{S\_b}(0) = 0$ for the correct circuit and the assumed arithmetic circuit with ripple carry adder structure. Based on *SPEC*, $P_S = P'_S$, so the difference between each pair of $P_{S\_b}(i) - P'_{S\_b}(i)$ must be zero. On the other hand, based on the coefficient of $P_{S\_b}(i) - P'_{S\_b}(i)$ in $P_S - P'_S$, it is impossible that $P_S - P'_S$ becomes zero.

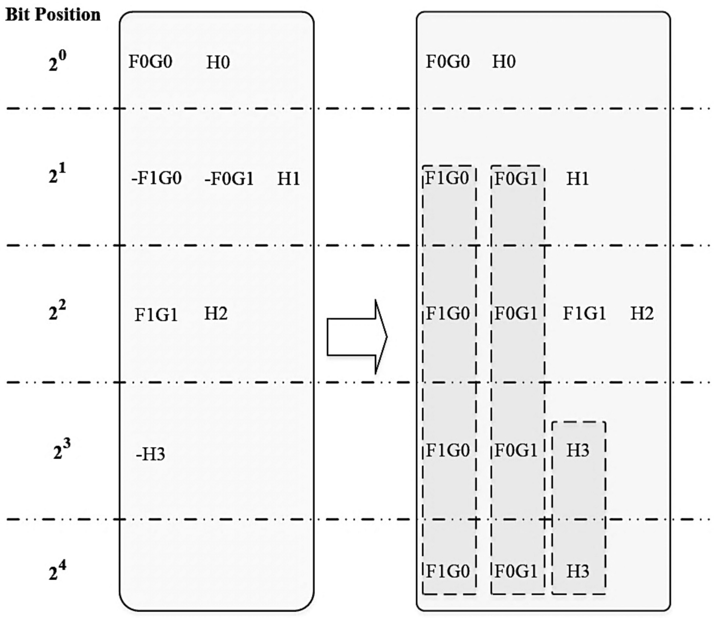

Fig. 6. Partial product generation of 2-bit signed MAC.

4.3 Ideal Basis Replacement

As mentioned before, to automatically correct the arithmetic circuits, our idea is to modify the cone ideal. For doing so, based on Definition 1, we need to replace the ideal basis. Algorithm 3 shows how to replace the basis for i^{th} output cone ideal, which gets the related bit-level specification $P_{S\_b}(i)$, the cone ideal $I_{CONE}(i)$, the suspicious generators F_{SUSP}, and the set of legal basis replacements *LegalRep* as inputs and returns the correct replacement solution, *Solution*.

In this algorithm, first of all, those suspicious generators related to $I_{CONE}(i)$ are determined (line 3). Then, all possible replacements for them are extracted from *LegalRep* (*totalRep* in line 4), and all combination of replacements is calculated (line 7). After that, $I_{CONE}(i)$ is modified according to each $rc \in totalRep$ ($I_{CONE_{rep}}(i)$ in line 9), and the membership of $P_{S\_b}(i)$ in $I_{CONE_{rep}}(i)$ is tested (line 10). Those replacements which resulting in zero-remainder are kept in line 12, and finally, they are merged by the given *LegalRep* and returned (line 15).

Proposition. *The obtained basis is a Groebner basis for the modified ideal.*

Proof. Since all functions in the ideal are related to a logical gate, ordering by RTTO, the first monomial of them is in the form of $1 \times m$, where m is the variable related to the output of the gate. Since each wire is driven by one gate, the first monomial of each polynomial is repeated only once, as a leading monomial. When each pair of polynomials has prime leading monomials, the given set of polynomials is a Groebner basis of the related ideal [37]. Our proposed algorithm replaces one basis polynomial with another one, with the same leading monomial, due to gate replacement assumption. This means that all pairs of polynomials of the modified set have prime leading monomials, which obtains the modified Groebner basis.

Note that, partitioning the circuit into cones of each output bit helps reduce the search space of the higher bit position output cones, due to correcting some suspicious gates during the correction process. To clarify it, it is assumed that the relativity of each pair of suspicious gates

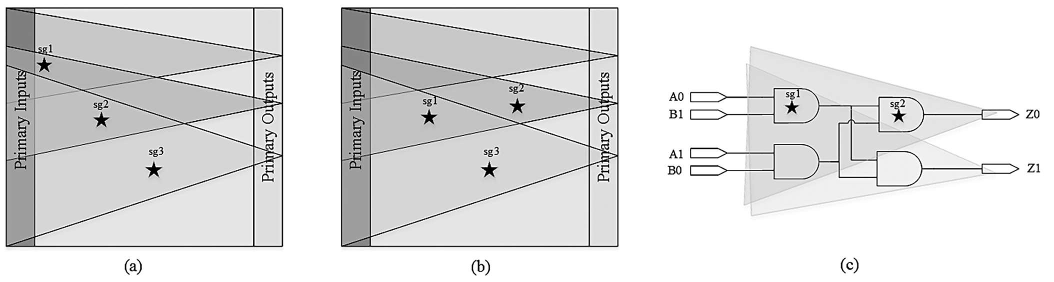

Fig. 7. Types of multiple suspicious gate locations, (a) distributed suspicious gates in separate cones, (b) suspicious gates in a cone, (c) example circuit of dependent suspicious gates, implementing 2Z1+Z0−A1B0−A0B1=0.

ALGORITHM 3: Ideal Basis Replacement

Inputs: Bit-level Specification ($P_S(i)$), Cone Ideal($I_{CONE}(i)$), Suspicious generators (F_{SUSP}), Legal replacements(*LegalRep*)

Output: Solutions (*Solution*)

```
1:   $ConeSolution = \{\}$
2:   for j in 0 to $|F_{SUSP}|$ do
3:       if $F_{SUSP}(j) \in I_{CONE}(i)$do
4:           Rep[j] ← FindBasisReplacements ($F_{SUSP}(j)$, $LegalRep$);
5:       end if
6:   end for
7:   $totalRep \leftarrow \prod_{j=0}^{|F_{SUSP}|} Rep[j]$;     //all possible replacements
8:   for each $rc$ in $totalRep$ do
9:       $I_{CONE_{rep}}(i) \leftarrow$ ReplaceIdealGenerator ($I_{CONE}(i)$, $rc$);
10:      $P_{S\_b}(i) \xrightarrow{GB(I_{CONE_{rep}}(i))} REM$
11:      if $REM == 0$ then
12:          $ConeSolution \leftarrow ConeSolution \cup \{rc\}$;
13:      end if
14:  end for
15:  $Solution \leftarrow$ MergeSolutions ($LegalRep$, $ConeSolution$);
return $Solution$;
```

can be classified into three categories: (i) **Independent Gates in Separate Output** cones (**IGSO**), (ii) **Independent Gates in One Output** cone (**IGOO**), and (iii) **Dependent Gates in One Output** cone (**DGOO**). In the IGSO case, it is assumed that the suspicious gates are located far apart, in a way that each output cone has at most one suspicious gate. This case is shown in Figure 7(a), where *sg* stands for suspicious gate. In this figure, although, the second cone has two suspicious gates sg_1 and sg_2, while analyzing the first cone, sg_1 will be resolved and only sg_2 must be resolved during analyzing the second cone. The second and the third category are related to suspicious gates, located in one output cone. It means that none of them belongs to the cone of the output in the lower bit-position, like sg_1 and sg_2 in Figure 7(b). There is only a set of correct types for independent suspicious gates in analyzing the desired output cone, while the dependent ones have more than one legal replacement. For example, whether sg_1 and sg_2 in Figure 7(b) are independent, correcting the second cone should find the correct types for them. In this case, analyzing the last cone tries to find the correct type of sg_3. In the dependent case, more than one

Table 2. Applying Ideal Basis Replacement to Z_2

Candidates (p_{4_b}, p_{16_b}, p_{17_b})	**Gates** (g_4, g_{16}, g_{17})	**REM**	**Solution**
(W_3+F_1+G_1, W_{13}+W_{11}+W_3, -)	(XOR, XOR, -)	F_1G_1+F_1+G_1	{}
(W_3+F_1+G_1, W_{13}+W_{11}+W_3+1, -)	(XOR, XNOR, -)	F_1G_1+F_1+G_1+1	{}
…		…	…
(W_3+F_1G_1, W_{13}+W_{11}+W_3, -)	(AND, XOR, -)	0	{(W_3+F_1G_1, W_{13}+W_{11}+W_3, -)}
…		…	…
(W_3+F_1G_1+1, W_{13}+W_{11}+W_3+1, -)	(NAND, XNOR, -)	0	{(W_3+F_1G_1, W_{13}+W_{11}+W_3, -), (W_3+F_1G_1+1, W_{13}+W_{11}+W_3+1, -)}

possible type is found by analyzing the second cone. So, the correct type of sg_1 and sg_3 will be found while analyzing the third cone. So, IGSO is the best distribution due to resolving one gate while analyzing each output cone. The next one is DGSO, while analyzing the related output cone, resolves all suspicious gates placed there. However, the DGOO is the last ranking distribution, in which more than one correct solution is found. As an example, there are two correct solutions for correcting the first cone of Figure 7(c): $(sg_1, sg_2) \in \{(AND, XOR), (NAND, XNOR)\}$. In these cases, resolving suspicious gates requires analyzing more than one output cone.

For example, consider the buggy circuit in Figure 2, again, with three suspicious gates, g_4, g_{16}, and g_{17}. The correction is started from Z_2 due to the non-membership of any of $F_{SUSP} = \{p_{4\_b}, p_{16\_b}, p_{17\_b}\}$ in $I_{CONE}(0)$ and $I_{CONE}(1)$. Since $F_{SUSP}[0] = p_{4\_b} \in I_{CONE}(2)$ and $F_{SUSP}[1] = p_{16\_b} \in I_{CONE}(2)$, $Rep[0]$ and $Rep[1]$ are collections of all polynomials related to all types of gate replacements and gate removements. Also, $F_{SUSP}[2] = p_{17\_b} \notin I_{CONE}(2)$ obtains $Rep[2]$={}. Combining *Rep* sets gives us all replacements *totalRep*, which are listed in Table 2. Also, the results of testing each replacement by $P_{S\_b}(2) \xrightarrow{I_{CONE_{rep}}(2)} REM$ are listed in Table 2, which shows that the cone replacement solution is *ConeSolution* = {(W_3+F_1G_1, W_{13}+W_{11}+W_3, -), (W_3+F_1G_1+1, W_{13}+W_{11}+W_3+1, -)}. The obtained solution shows that g_4 and g_{16} are dependent on suspicious gates. Merging obtained solution with *LegalRep* returns *ConeSolution*, itself. Continuing applying the algorithm on the last output bit gives *ConeSolution* = {(W_3+F_1G_1, -, W_{14}+$W_{11}W_3$)} and the final replacement is *LegalRep* = {(W_3+F_1G_1, W_{13}+W_{11}+W_3, W_{14}+$W_{11}W_3$)}. It means the correct solution gate types are (g_4: AND, g_{16}: XOR, g_{17}: AND), which correct the circuit without an area overhead compared to the buggy circuit shown in Figure 2. As shown in the example, our proposed method depends on the suspicious gates rather than buggy gates, and all computations are performed over the given suspicious list.

4.4 Time Complexity

Since Algorithm 1 depends on Algorithms 2 and 3, computing time complexity must be started from the others. There are some effective parameters on each algorithm, like the size of the cone, the number of suspicious gates, the number of legal replacements of suspicious gates, and so on. Starting from the third algorithm, assume S suspicious gates are placed in the correcting output cone. The maximum number of legal replacements is 10^S, which is reduced according to the effect of correcting lower bit-position output cones on the desired cone. The reduction process is performed on each gate in the cone, which is assumed to be m gates. The ratio of the gate numbers (m) over the cone depth (d), seems to be a propositional parameter for the reduction complexity. Note that the depth of the circuit is defined as the longest path length from the primary inputs to the desired output bit. So, the time complexity of applying Algorithm 3 to an output cone is $O(10^S m/d)$.

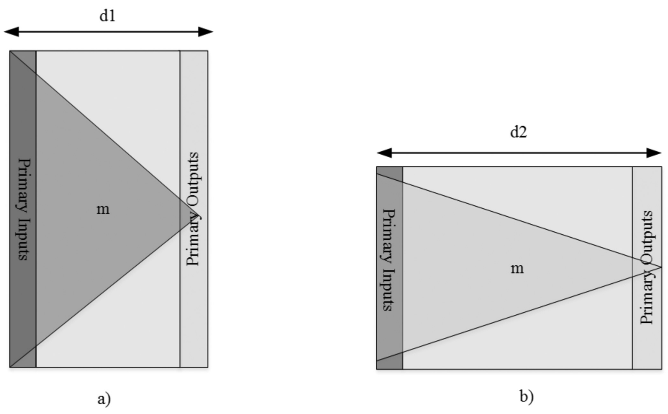

Fig. 8. Various architectures with the same number of gates m: (a) compact architecture with small depth. (b) deep architecture.

The effect of the circuit architecture can be seen here. In Figure 8, correcting the left cone with the same gate number and the same suspicious gates as the right one requires less operation time due to $d1 < d2$, which means Figure 8(a) has a compact architecture rather than Figure 8(b).

Algorithm 2 has $O(m)$ time complexity for a cone with m gates. To calculate Algorithm 1 time complexity, in addition to running the other two algorithms, the time complexity of calculating bit-level specification must be considered. Assume there are T terms in the PPG stage of the correcting output cone. Considering N output bit, the whole bit-level specification is obtained roughly in $O(\frac{(\frac{3}{2})^N T}{N})$ for each output bit. Since the bit-level specification only depends on the specification polynomial, it can be computed in the pre-process stage. So, assuming m gates and S suspicious gates in each output cone, the time complexity of Algorithm 1 is $O(N \times (m + \frac{10^S m}{d}))$.

5 Experimental Results

In order to evaluate the proposed method, various buggy multiplier circuits generated and flattened by the Genmul [39] and Yosys [40] tools are taken into account. The correction method is implemented in Python to benefit from **Zero Suppressed Diagrams** (**ZDD**) utilized in the PolyBori framework [25] and C++. Three experiments are performed on a 3.7 GHz intel Core$^{\text{TM}}$ i7 processor and 32 GB RAM running Linux OS, which will be explained below. Note that each experiment is repeated ten times and the reported correction time, is the average of them.

The first experiment compares the correction time of the proposed method and the method of [34], which are applied to numbers of unsigned multipliers with a buggy gate and various architectures. In general, arithmetic circuits are implemented in two main stages: PPG and PPS. PPS is usually implemented (especially in multipliers) in two stages: **Partial Product Addition** (**PPA**), and **Final Stage Addition** (**FSA**). In the first experiment, the architecture of multipliers is denoted as $\alpha o \beta o \gamma$. Here, α refers to the architecture of the PPG, β shows the architecture of PPA, and γ is related to FSA architecture. The first part of Table 3 is general information about the multipliers, bug locations and the first buggy output according to the buggy gate (First Affected Output), the second part is related to the gate count and area overhead, and the last part is about the correction time.

The area overhead of [34] is at least 64.54%, while the proposed method corrects circuits without area overhead. Since the proposed method partitions the circuit and utilizes binary polynomials, its run-time is 15.9× faster than the remainder-based method [34]. Since obtaining the word-level remainder needs whole circuit analysis, increasing the circuit size leads to increase the run-time of

Table 3. Area Overhead and Correction Time Comparison between the Proposed Method and the Previous Method [34] with Single Bug and One Suspicious Gate

				Size of the circuit (gate count)			Run-Time (seconds)		
					[34]				
Bug location	***Architecture***	***Size***	***FAO***	***Buggy***	***Corrected***	***%overhead***	**[34]**	**This Work**	***Speedup***
PPG	SPoARoCK	32×32	6	7,912	17,946	126.82	3.34	0.5	6.68×
		64×64	7	32,200	76,002	136.03	33.03	2.21	14.95×
		128×128	9	129,936	–	NA	OOM	35.59	NA
	SPoARoRC	32×32	6	7,873	17,907	127.45	2.98	0.46	6.48×
		64×64	8	32,129	75,209	134.08	25.73	4.75	5.42×
		128×128	8	129,793	314,059	141.97	236.5	13.48	17.54×
	SPoCWToCL	32×32	7	14,982	24,672	64.68	6.39	1.47	4.35×
		64×64	9	65,637	108,001	64.54	79.77	103.57	0.77×
		128×128	6	275,449	—	NA	OOM	8	NA
	SPoCWToKS	32×32	7	11,181	20,871	86.66	4.74	1.41	3.36×
		64×64	8	46,692	89,772	92.26	53.85	7.68	7.01×
		128×128	8	190,183	—	NA	OOM	17.97	NA
	SPoDToKS	32×32	8	8,432	17,784	110.91	3.52	6.66	0.53×
		64×64	7	33,643	77,445	130.2	36.16	2.3	15.72×
		64×64	40	33,643	56,791	68.8	22.74	TO	NA
		128×128	8	133,605	317,871	137.92	394.09	16.3	24.18×
	SPoWToRC	32×32	6	8,191	18,225	122.5	3.29	0.48	6.85×
		64×64	8	33,039	76,121	130.4	27.73	8.27	3.35×
		128×128	6	131,735	318,985	142.14	264.47	5.34	49.53×
PPA	SPoARoRC	32×32	7	7,873	13,764,350	174,729.8	197.61	0.98	201.64×
		64×64	8	32,129	—	NA	OOM	4.29	NA
		128×128	7	129,793	—	NA	OOM	7.16	NA
	SPoCWToCL	32×32	7	14,982	32,282,796	215,377.21	443.28	1.57	282.34×
		64×64	9	65,637	14,407,563	21,850.37	478.61	169.21	2.83×
		128×128	7	275,449	8,180,065	2,869.72	2052.36	12.26	167.4×
	SPoDToLF	32×32	6	8,049	2,958,690	36,658.48	44.06	0.45	97.91×
		64×64	7	32,683	—	NA	OOM	2.29	NA
		128×128	6	131,301	—	NA	OOM	5.58	NA
	SPoWToRC	32×32	7	8,191	4,254,684	5,1843.4	62.31	1.26	49.45×
		64×64	8	33,039	—	NA	OOM	7.84	NA
		128×128	7	131,735	—	NA	OOM	7.43	NA
FSA	SPoCWToBK	32×32	8	10,671	—	NA	OOM	9.69	NA
		64×64	9	45,249	—	NA	OOM	183.35	NA
		128×128	10	186,478	—	NA	OOM	OOM	NA
	SPoCWToCL	32×32	8	14,982	36,574	144.12	2191.53	9.81	223.4×
		64×64	9	65,637	—	NA	OOM	184.73	NA
		128×128	10	275,449	—	NA	OOM	OOM	NA

(Continued)

Table 3. Continued

				Size of the circuit (gate count)			Run-Time (seconds)		
					[34]				
Bug location	*Architecture*	*Size*	*FAO*	*Buggy*	*Corrected*	*%overhead*	[34]	This Work	*Speedup*
FSA	SPoDToLF	32×32	2	8,049	47,343	488.18	0.64	0.31	2.06×
		64×64	8	32,683	–	NA	OOM	35.35	NA
		128×128	5	131,301	–	NA	OOM	5.25	NA
				minimum overhead = 64.54%			**average speedup = 51.90×**		

TO: Time-Out (3 hours) **OOM**: Out Of Memory (25GB) **NA:** Not Applicable
FAO: First Affected Output by Buggy Gate
PPG stage ⇒ **SP:** Simple Partial Product Generator
PPA stage ⇒ **AR:** Array **DT:** Dadda Tree **WT:** Wallace Tree **CWT:** Counter-based Wallace Tree
FSA stage ⇒ **RC:** Ripple Carry Adder **CL:** Carry Look-ahead Adder **CK:** Carry Skip Adder **BK:** Brent-Kung Adder
KS: Kogge Stone Adder **LF:** Ladner Fischer Adder **SE:** Serial Prefix Adder

the method [34]. However, our proposed method analyzes the cone of the buggy output. Whether the buggy gate is placed in the cone of the output in lower bit-positions, it needs less process time, although in a larger multiplier. For example, correcting the 32- 64-, and 128-bit multiplier with SPoARoRC architecture with a bug in PPG, needs 2.98, 25.73, and 236.5 seconds to be corrected by [34], respectively, while it can be corrected in 0.46, 4.75, and 13.48 seconds by the proposed method. Also, when a buggy gate is placed in the lower output cone bit-position, the proposed method corrects the circuit, while in large circuits, calculating the word-level remainder is impossible, especially when a buggy gate is located in levels near the output. For example, correcting the 64-bit multiplier with SPoCWToCL architecture and a bug located in the FSA stage is performed in 184.73 seconds, while the method in [34] cannot correct the circuit due to the memory issue. On the other hand, in some cases like 32-bit SPoDToKS multiplier with a bug in PPG, the run-time of the proposed method is greater than the method in [34]. Calculating the remainder is the main time-consuming process of the method [34]. In cases of buggy circuits with a bug in PPG, calculating the remainder requires less time budget, rather than locating in deeper layers and/or multiple bugs. On the other hand, the correction time of the method [34] becomes greater for smaller FAO, due to inserting a correcting sub-circuit to the buggy circuit, starting from FAO. On the other hand, the greater FAO means an output cone with a larger size, producing a larger ideal generator set, which requires greater run-time for performing the Groebner basis reduction by the proposed method. So, the correction time in some cases with a buggy gate in PPG is greater than the method in [34]. In addition, the required time budget for bit-level specification calculation limits the proposed method. As shown in Table 3, the proposed method disables to correct 64-bit SPoDToKS multiplier with a buggy gate in PPG.

In Table 4, we have compared our method with [34] and [36] in terms of the correction time. In this table, column #*Bug* is the number of buggy gates. Note that the list of suspicious gates is assumed to be the same as buggy gates in this experiment. The buggy gates in the IGSO category are located far apart, in a way that each cone has at most one suspicious gate. The first and second major rows of Table 4 show the results of this category. In the IGOO category, it is possible to have more than one uncorrected buggy gate in a cone, while they are functionally independent from each other. The third major row of Table 4 lists the results of this category. In the DGOO category, multiple uncorrected dependent buggy gates in a cone are considered which require more time to be corrected. The fourth major row of Table 4 shows the results of this category. Also, the last two major rows are a combination of IGSO and DGOO, in which there are at least two separate bugs in the cone, while others are dependent ones. As seen in Table 4 the proposed method can

Table 4. Correction Time Comparison in the Case of Multiple Bugs

Architecture	*Size*	*#Bug*	*stages*	*FAO*	*Maximum ZDD nodes*	*Run-Time (sec)*			*Speedup (%)*	
						[34]	[36]	This Work	relative to [34]	relative to [36]
SPoDToLF (IGSO)	128×128	2	PPG, PPA	6, 7	574	241.45	868.81	8.69	27.79×	99.98×
		3	PPG, PPA	5, 6, 7	574	253.56	TO	9.47	26.77×	NA
		4	PPG, PPA, FSA	4, 5, 6, 7	574	1341.79	TO	12.24	109.62×	NA
SPoARoCK (IGSO)	32×32	2	PPG, FSA	5, 6	207	55.03	OOM	0.53	103.84×	NA
		3	PPG, PPA, FSA	5, 6, 25	207	OOM	OOM	15.12 (TO)	NA	NA
		4	PPG, PPA, FSA	5, 6, 7, 25	766	OOM	OOM	14.95 (TO)	NA	NA
SPoDToKS (IGOC)	128×128	2	PPG, PPA	7	1,803	256.03	OOM	42.68	6×	NA
		3	PPG, PPA	7	1,803	488.76	OOM	344.45	1.42×	NA
		4	PPG, PPA, FSA	7	2,131	OOM	OOM	3080.02	NA	NA
SPoWToCL (DGOC)	64×64	2	PPG, PPA	7	1,803	64.14	92.79	25.73	2.49×	3.61×
		3	PPG, PPA	7	1,815	OOM	TO	204.84	NA	NA
		4	PPG, PPA, FSA	7	1,815	OOM	TO	1608.58	NA	NA
SPoCWToBK (IGSO, DGOO)	64×64	2	PPG	6, 7	599	479.66	591.603	4.92	94.24×	120.24×
		3	PPG, PPA	6, 7	617	TO	OOM	46.88	NA	NA
		4	PPG, PPA	6, 7, 8	1,771	TO	OOM	53.84	NA	NA
SPoWToRC (IGSO, DGOO)	64×64	2	PPG	7, 8	1,771	14.81	37.83	10.75	1.38×	3.52×
		3	PPG	7, 8	1,822	267.769	OOM	45.28	5.91×	NA
		4	PPG	7, 8	5,772	455.49	OOM	385.34	1.18×	NA
average									34.60×	56.84×

TO: Time-Out (3 hours) **OOM**: Out Of Memory (25GB) **NA:** Not Applicable **FAO:** First Affected Output by Buggy Gate

automatically correct different multipliers with multiple bugs in 34.6× and 56.84× faster than the methods of [34] and [36], respectively.

In summary, the first category is the best distribution of buggy gates, where each cone needs to correct only one buggy gate. The second category is a bit more complex than the first one. Here, the iteration of applying the correction method to the cone is multiplied by the number of all correction candidates of the buggy gates. However, at the end of correction phase, all of buggy gates located in the cone will be corrected. The third category is the complicated one, where some of the buggy gates remained unresolved which will be resolved during the process of higher output cone.

In the second major row of Table 4 (32-bit multiplier with SPoARoCK architecture), the output cones (5, 6, 25) and (5, 6, 7, 25) with three and four buggy gates in PPG, PPA, and FSA stages are considered. The methods of [34] and [36] fail because the remainder cannot be obtained. However, the proposed method could correct the buggy gates, located in the output cones of (5, 6) and (5, 6, 7), and the buggy gate in the higher output bit-position cannot be resolved. In other words, the advantage of the proposed method compared to [34, 36] is the fact that the correction method is able to start analyzing the circuit and also, possibly correct some bugs, even if there are one or more buggy gates, prevent the remainder calculation in the reasonable time and/or memory.

The last experiment evaluates the impact of the number of buggy and suspicious gates on the correction time of the signed 8-bit SPoDToSE, unsigned 64-bit SPoCWToBK, and unsigned 128-bit SPoARoRC multipliers. The results are extracted for 2, 3, 4, and 5 buggy gates with a various number of suspicious gates from 2 to 13. As shown in Figure 9, correction time grows by increasing the number of suspicious gates, where the number of buggy gates has a negligible impact on it because the proposed method tries all possible replacements of suspicious generators independent of the buggy gates.

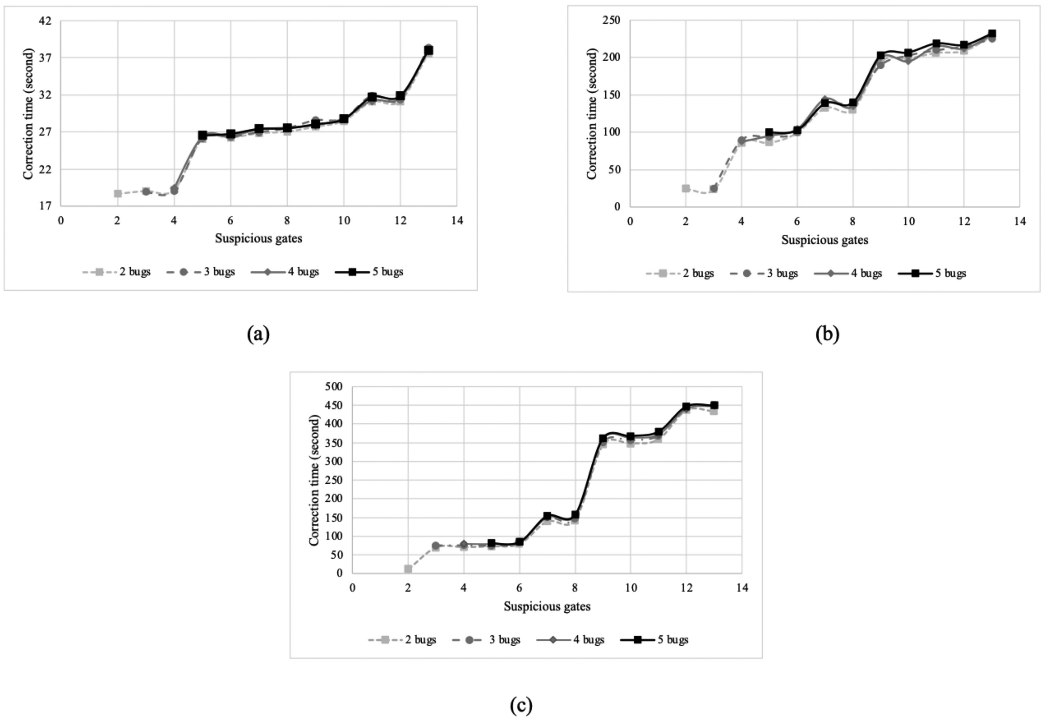

Fig. 9. Correction time with various number of suspicious and buggy gates, (a) 8-bit signed SPoDToSE multiplier, (b) 64-bit unsigned SPoCWToBK multiplier, and (c) 128-bit unsigned SPoARoRC multiplier.

6 Conclusion and Future Work

Existing SCA-based correction methods are based on the remainder, which makes correction of large arithmetic circuits impossible. The proposed correction method partitioned the circuit to make it independent of the word-level remainder. Correcting each partition is based on the ideal basis replacement approach, which reduces the size of the problem. The results illustrated that the proposed method corrects the arithmetic circuits 51.9× and 45.72× faster, on average, than the state-of-the-art correction techniques, having single and multiple bugs, respectively, without area overhead.

In the future, we are going to merge the method proposed in this work and the method discussed in [34] as a complete automatic correction tool for arithmetic circuits. Moreover, in order to come up with a complete solution for arithmetic circuit correction, we are going to add existing works like method [36] to our auto-correction package in the future.

References

[1] H. Foster. 2022. 2022 Wilson research group functional verification study: IC/ASIC functional verification trend report. *Wilson Research Group and Mentor, A Siemens Business, White Paper.* (2022). https://resources.sw.siemens.com/en-US/white-paper-2022-wilson-research-group-functional-verificatio

[2] H. Foster. 2022. 2022 Wilson research group functional verification study: FPGA functional verification trend report. *Wilson Research Group and Mentor, A Siemens Business, White Paper.* 2022. https://resources.sw.siemens.com/en-US/white-paper-2022-wilson-research-group-functional-verificatio

[3] R. E. Bryant and Y. Chen. 2001. Verification of arithmetic circuits using binary moment diagrams. In *International Journal on Software Tools for Technology Transfer* 3, 2 (2001), 137–155.

[4] J. Kumar, Y. Miyasaka, A. Srivastava, and M. Fujita. 2023. Formal verification of integer multiplier circuits using binary decision diagrams. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 4 (2023), 1365–1378.

[5] M. Ciesielski, P. Kalla, and S. Askar. 2006. Taylor expansion diagrams: A canonical representation for verification of data flow designs. In *IEEE Transactions on Computers* 55, 9 (2006), 1188–1201.

[6] B. Alizadeh and M. Fujita. 2010. Modular datapath optimization and verification based on modular-HED. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 9 (2010), 1422–1435.

[7] B. Keng, and A. Veneris. 2012. Path directed abstraction and refinement in sat-based design debugging. In *Proceedings of the 49th Annual Design Automation Conference* (2013), 947–954.

[8] K. L. McMillan. 2003. Interpolation and SAT-based model checking. In *Proceedings of the International Conference on Computer Aided Verification*. 1–13.

[9] T. Matsumoto, S. Ono, and M. Fujita. 2012. An efficient method to localize and correct bugs in high-level designs using counterexamples and potential dependence. In *Proceedings of the International Conference on VLSI and System-on-Chip*. 291–294.

[10] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een. 2006. Improvements to combinational equivalence checking. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*. 836–843.

[11] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita. 2009. A formal approach for debugging arithmetic circuits. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 28, 5 (2009), 742–754.

[12] F. Farahmandi and B. Alizadeh. 2015. Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction. In *Microprocessors and Microsystems*, 39, 2 (2015), 83–96.

[13] F. Farahmandi, B. Alizadeh, and Z. Navabi. 2014. Effective combination of algebraic techniques and decision diagrams to formally verify large arithmetic circuits. In *IEEE Computer Society Annual Symposium on VLSI* (2014), 338–343.

[14] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski. 2016. Formal verification of arithmetic circuits by function extraction. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 35, 12 (2016), 2131–2142.

[15] Y. Watanabe, N. Homma, T. Aoki, and T. Higuchi. 2008. Arithmetic circuit verification based on symbolic computer algebra. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 91, 10 (2008), 3038–3046.

[16] D. Kaufmann, and A. Biere. 2021. AMulet 2.0 for verifying multiplier circuits. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 357–364.

[17] D. Kaufmann, A. Biere, and M. Kauers. 2019. Verifying large multipliers by combining SAT and computer algebra. In *Formal Methods in Computer Aided Design* (2019), 28–36.

[18] A. Mahzoon, D. Große, Ch. Scholl, and R. Drechsler. 2020. Towards formal verification of optimized and industrial multipliers. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 544–549.

[19] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler. 2016. Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 1048–1053.

[20] C. Yu and M. Ciesielski. 2018. Formal analysis of Galois field arithmetic circuits-parallel verification and reverse engineering. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 2 (2018), 354–365.

[21] N. Homma, K. Saito, and T. Aoki. 2013. Toward formal design of practical cryptographic hardware based on Galois field arithmetic. In *IEEE Transactions on Computers* 63, 10 (2013), 2604–2613.

[22] A. Ito, R. Ueno, and N. Homma. 2021. Efficient formal verification of galois-field arithmetic circuits using ZDD representation of Boolean polynomials. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2021), 794–798.

[23] J. Lv, P. Kalla, and F. Enescu. 2013. Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 9 (2013), 1409–1420.

[24] U. Gupta, P. Kalla, and V. Rao. 2018. Boolean gröbner basis reductions on finite field datapath circuits using the unate cube set algebra. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 3 (2018), 576–588.

[25] M. Brickenstein and A. Dreyer. 2009. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. In *Journal of Symbolic Computation* 44, 9 (2009), 1326–1345.

[26] M. Barhoush, A. Mahzoon, and R. Drechsler. 2021. Polynomial word-level verification of arithmetic circuits. In *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*. 1–9.

[27] A. Mahzoon, D. Große, and R. Drechsler. 2018. Combining symbolic computer algebra and boolean satisfiability for automatic debugging and fixing of complex multipliers. In *IEEE Computer Society Annual Symposium on VLSI* (2018), 351–356.

[28] F. Farahmandi and P. Mishra. 2018. Automated test generation for debugging multiple bugs in arithmetic circuits. In *IEEE Transactions on Computers* 68, 2 (2018), 182–197.

[29] V. N. Kravets, N. Lee, and J. R. Jiang. 2019. Comprehensive search for ECO rectification using symbolic sampling. In *Proceedings of the Design Automation Conference*. 1–6.

[30] Y. Kimura, A. M. Gharehbaghi, and M. Fujita. 2019. Signal selection methods for efficient multi-target correction. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. 1–5.

[31] B. Alizadeh and S. R. Sharafinejad. 2018. Incremental SAT-Based accurate auto-correction of sequential circuits through automatic test pattern generation. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 2 (2018), 245–252.

[32] B. Alizadeh and Y. Abadi. 2020. Incremental SAT-based correction of gate level circuits by reusing partially corrected circuits. In *IEEE Transactions on Circuits and Systems II: Express Briefs* 67, 12 (2020), 3063–3067.

[33] U. Gupta, P. Kalla, I. Ilioaea, and F. Enescu. 2019. Exploring algebraic interpolants for rectification of finite field arithmetic circuits with gröbner bases. In *IEEE European Test Symposium* (2019), 1–6.

[34] N. A. Sabbagh and B. Alizadeh. 2021. Arithmetic circuit correction by adding optimized correctors based on groebner basis computation. In *IEEE European Test Symposium* (2021), 1–6.

[35] V. Rao, I. Ilioaea, H. Ondricek, P. Kalla, and F. Enescu. 2021. Word-level multi-fix rectifiability of finite field arithmetic circuits. In *Proceedings of the 22nd International Symposium on Quality Electronic Design*. 41–47.

[36] V. Rao, H. Ondricek, P. Kalla, and F. Enescu. 2021. Rectification of integer arithmetic circuits using computer algebra techniques. In *Proceedings of the 39th International Conference on Computer Design*. 186–195.

[37] D. Cox, J. Little, and D. OShea. 2013. Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra. In *Proceedings of the Springer Science and Business Media.*

[38] M. Fujita. 2018. Automatic correction of logic bugs in hardware design: Partial logic synthesis. *Procedia Computer Science* 125 (2018), 790–800.

[39] A. Mahzoon, D. Große, and R. Drechsler. Multiplier Generator GenMul. Retrieved from http://www.sca-verification.org/genmul

[40] C. Wolf, J. Glaser, and J. Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, Vol. 97.

Received 30 January 2024; revised 18 April 2024; accepted 7 June 2024