Formal Verification of Modular Multipliers using Symbolic Computer Algebra and Boolean Satisfiability

Alireza Mahzoon  
University of Bremen  
Bremen, Germany  
mahzoon@informatik.uni-bremen.de

Daniel Große  
Johannes Kepler University Linz  
Linz, Austria  
daniel.grosse@jku.at

Christoph Scholl  
University of Freiburg  
Freiburg, Germany  
scholl@informatik.uni-freiburg.de

Alexander Konrad  
University of Freiburg  
Freiburg, Germany  
konrada@informatik.uni-freiburg.de

Rolf Drechsler  
University of Bremen/DFKI  
Bremen, Germany  
drechsler@uni-bremen.de

ABSTRACT

Modular multipliers are the essential components in cryptography and Residue Number System (RNS) designs. Especially, $2^n - 1$ and $2^n + 1$ modular multipliers have gained more attention due to their regular structures and a wide variety of applications. However, there is no automated formal verification method to prove the correctness of these multipliers. As a result, bugs might remain undetected after the design phase.

In this paper, we present our modular verifier that combines Symbolic Computer Algebra (SCA) and Boolean Satisfiability (SAT) to prove the correctness of $2^n - 1$ and $2^n + 1$ modular multipliers. Our verifier takes advantage of three techniques, i.e. coefficient correction, SAT-based local vanishing removal, and SAT-based output condition check to overcome the challenges of SCA-based verification. The efficiency of our verifier is demonstrated using an extensive set of modular multipliers with up to several million gates.

1 INTRODUCTION

Modular arithmetic nowadays plays an important role in many applications such as cryptography and RNS. Several modular arithmetic units, e.g. adders and multipliers are proposed and implemented to meet the increasing demands for efficient modular computations. Among these units, $2^n - 1$ and $2^n + 1$ modular multipliers have gained special focus due to their variety of applications and regular structures. The $2^n \pm 1$ modular multipliers are used in International Data Encryption Algorithm (IDEA) block cipher [5] for encryption. They are also employed in some implementations for fast conversion of RNS (IDEA) block cipher [5] for encryption. They are also employed in some implementations for fast conversion of RNS to weighted number system [3]. Moreover, they are used for Fermat number transformation and pseudorandom number generation.

Several architectures have been proposed for $2^n \pm 1$ modular multipliers [13, 14, 17]. They aim to implement the modular multiplier function while minimizing the area and delay. These architectures are usually complex and contain a huge number of gates. Thus, they are highly error-prone. An important phase after the design of modular multipliers is formal verification to prove their correctness. Unfortunately, there is a very limited number of works on the verification of modular multipliers. The authors of [15] proposed a formal verification method based on theorem proving to guarantee the correctness of Montgomery modular multipliers. However, the technique is not automated, and it cannot be applied to $2^n \pm 1$ modular multipliers.

Recently, SCA-based methods have shown very good results for the verification of integer arithmetic circuits, including multipliers [4, 6, 8, 16] and dividers [11, 12]. 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 gates (or nodes of an AND-Inverter Graph (AIG)) as a set of polynomials $P_G$, and (3) use the Gröbner basis theory to prove the membership of $SP$ in the ideal generated by $P_G$. The just mentioned 3rd step consists of the step-wise division of $SP$ by $P_G$ (or equivalently substitution of variables in $SP$ with $P_G$) known as backward rewriting, and eventually the evaluation of the remainder. If the remainder is zero, the multiplier is correct. Otherwise, it is buggy.

Despite the success of SCA-based method for the verification of integer multipliers, it fails when it comes to the verification of modular multipliers. The failure is due to the three obstacles: (1) The modular computations in the multiplier changes some bit positions. This effect is reflected in backward rewriting as the change of some coefficients which leads to an explosion in the size of the intermediate polynomial after a few substitution steps. (2) Several multi-variable monomials known as vanishing monomials appear during the backward rewriting. These monomials are reduced to zero after several steps of backward rewriting or under input conditions. However, they cause the generation of many new monomials and variables before cancellation. (3) Obtaining the zero remainder is not enough to prove that a modular multiplier is correct. It must also be shown that the output is always smaller than the modulo value.

In this paper, we propose a modular verifier to prove the correctness of $2^n - 1$ and $2^n + 1$ modular multipliers. Our modular verifier combines SCA and SAT. We come up with three techniques to overcome the challenges of verifying modular multipliers. First, we introduce a coefficient correction technique for SCA to obtain the desired coefficients after each substitution step and avoid the explosion. Then, we propose a SAT-based method to remove the vanishing monomials locally and avoid the generation of large number of monomials during global backward rewriting. Finally, we introduce a SAT-based technique to check whether the output condition holds for a modular multiplier. To the best of our knowledge, this paper is the first attempt for automated formal verification of modular multipliers.
2 PRELIMINARIES

2.1 Multiplier Structure
An integer multipliers consist of: (1) Partial Product Generator (PPG) which generates partial products from two inputs, (2) Partial Product Accumulator (PPA) which reduces partial products using multioperand adders and computes their sums, and (3) Final Stage Adder (FSA) which converts these sums to the corresponding binary output.

The input of a verification method is usually a gate-level netlist or an AIG. We use both representations in this paper. However, we prefer the AIG for the experiments since it gives us the possibility of advanced reverse engineering to identify atomic blocks, e.g. Half-Adders (HAs) and Full-Adders (FAs) [7, 16].

2.2 Verification using SCA
The goal of SCA-based verification is to formally prove that all signal assignments consistent with the gate-level or AIG representation evaluate the specification polynomial (SP) to 0. The SP determines the function of an arithmetic circuit based on its inputs and outputs, e.g. for the 2 × 2 multiplier of Figure 1 SP = 8Z_2 + 4Z_1 + 2Z_1 + Z_0 - (2A_1 + A_0)(2B_1 + B_0), where 8Z_2 + 4Z_1 + 2Z_1 + Z_0 represents the word-level representation of the 4-bit output, and (2A_1 + A_0)(2B_1 + B_0) represents the product of the 2-bit inputs.

Before verification, the nodes of an AIG (or gates of a gate-level representation) should be modeled as polynomials describing the relation between inputs and outputs. Based on the type of nodes and edges, five different operations might happen in an AIG. Assuming z is the output, and n_i and n_j are the inputs of a node:

\[
\begin{align*}
    z & = n_i \implies p_N = z = n_i, \\
    z & = n_i \land n_j \implies p_N = z = n_i n_j, \\
    z & = \neg n_i \implies p_N = z = 1 - n_i, \\
    z & = \neg n_i \land n_j \implies p_N = z = 1 - n_i n_j, \\
    z & = \neg n_i \land \neg n_j \implies p_N = z = 1 + n_i + n_j - n_i n_j.
\end{align*}
\]

(1)

The extracted node polynomials are in the form \( P_N = x - \text{tail}(P_N) \), where x is the node’s output, and \( \text{tail}(P_N) \) is a function based on the node’s inputs. Similarly, the polynomials for the gates can be extracted in a gate-level representation (see [6, 10]).

Based on the Gröbner basis theory, all signal assignments consistent with the AIG evaluate the specification polynomial SP to 0, iff the remainder of dividing SP by the AIG node polynomials is equal to 0 (see [4] for more details).

The step-wise division of SP by node polynomials is shown in Figure 2 for the 2 × 2 multiplier. As the remainder is zero, the circuit is bug-free. In arithmetic circuits, dividing \( SP_j \) by a node polynomial \( P_N = x_i - \text{tail}(P_N_j) \) is equivalent to substituting \( x_i \) with \( \text{tail}(P_N_j) \) in \( SP_j \). For example, dividing \( SP_j \) by \( P_N_j \), in Figure 2 is equivalent to substituting \( n_{11} \) with \( \text{tail}(P_N_j) = n_{11} p_j \) in \( SP_j \). The process of step-wise division (substitution) is called backward re-writing. We refer to this intermediate polynomial as SP_j in the rest of the paper.

2.3 Modular Multipliers

A Modular Multiplier produces the product of two unsigned integers modulo a fixed constant m:

\[
Z = A \times B \mod m,
\]

(2)

where the inputs and output must be always smaller than m:

\[
A, B < m \quad \text{and} \quad Z < m.
\]

(3)

There are two ways to generate modular multipliers: (1) the product of inputs is computed using a normal integer multiplier; then, a modular reduction unit is used after the multiplier to obtain the final result, (2) the modular computations are integrated into the three stages of the multiplier. Since the second implementation method is much more area and delay-efficient, it is used in almost all designs.

Now, we focus on the two special moduli \( m = 2^n - 1 \) and \( m = 2^n + 1 \).

These modular multipliers have regular structures; thus, they can be automatically generated for arbitrary sizes.

2.3.1 \( 2^n - 1 \) Modular Multiplier. A \( 2^n - 1 \) modular multiplier produces the product of two n-bit unsigned integers modulo \( 2^n - 1 \).

Figure 3 shows the structure of a \( 2^4 - 1 = 15 \) modular multiplier, which receives two 4-bit numbers and returns their modular product (see [9, 17] for the details). The multiplier is created using AND gates in the first stage to generate partial products and HAs and FAs in the second and third stages to reduce them. For simplification, we show the AND operations as \( P_{i,j} = A_i \land B_j \). Please note that the weight of partial products should be calculated modulo \( 2^n - 1 \) at each step of partial product reduction. Therefore, the bits in position \( n \) (i.e. partial products with weight \( 2^n \)) are reinserted in position 0. The red wires depict the bits whose positions have been changed.

2.3.2 \( 2^n + 1 \) Modular Multiplier. A \( 2^n + 1 \) modular multiplier produces the product of two \( (n + 1) \)-bit unsigned integers modulo \( 2^n + 1 \).

Figure 4 shows the structure of a \( 2^4 + 1 = 17 \) modular multiplier, which receives two 5-bit numbers (see [9, 14] for the details). The first stage consists of AND, OR, and NOT gates. The second and the third stages of the multiplier are made of HAs and FAs. In \( 2^n + 1 \) modular multipliers, the bits in position \( n \) (i.e. partial products with weight \( 2^n \)) are inverted and then reinserted in position 0. The red wires show the bits with changed positions.
3 CHALLENGES OF VERIFICATION

There are three challenges when it comes to the SCA-based verification of modular multipliers. We explain them in three subsections.

3.1 Effect of Modulo on Coefficients

Atomic blocks, including HAs and FAs, have a compact word-level relation between their inputs and outputs:

\[ HA(in : (X, Y), out : (C, S)) \Rightarrow 2C + S = X + Y, \]
\[ FA(in : (X, Y), out : (C, S)) \Rightarrow 2C + S = X + Y + Z. \]  

Thus, for example, if we have \( 2C + S \) in our intermediate polynomial \( SP_i \), we can directly substitute it with \( X + Y + Z \) for a FA with \( X, Y, Z \) inputs, and \( S, C \) outputs during backward rewriting. However, it is possible that \( 2C + S \) does not appear in \( SP_i \), i.e. a polynomial consisting of \( C \) and \( S \) monomials but with different coefficients (e.g. \( -C + S \)) occurs. In this case, the sum (S) and carry (C) are substituted separately with the corresponding polynomials.

Eq. (5) shows the first six backward rewriting steps of the 2\(^n\) + 1 modular multiplier in Figure 4.

\[ SP : 16Z_3 + 8Z_2 + 4Z_1 + 2Z_0 + Z_6 - A \times B, \]
\[ SP \xrightarrow{Z_1} SP_1 := 8S_3 + 8S_2 + 4Z_0 + 2Z_6 - A \times B, \]
\[ \ldots \]
\[ SP_{c_3S_3} \xrightarrow{Z_3} SP_3 := -c_3 + 8S_3 + 4S_2 + 2S_1 + S_0 + 1 - A \times B, \]
\[ SP_{c_3S_3} \xrightarrow{Z_4} SP_4 := 8p^{(3)}_3 + 8p^{(3)}_2 + 8Z_0 - 17p^{(3)}_1 p^{(3)}_0 - 17p^{(3)}_2 c_2 - 17p^{(3)}_1 c_2 + 34p^{(3)}_1 p^{(3)}_0 c_2 + 4S_2 + 2S_1 + S_0 + 1 - A \times B, \]
\[ \ldots \]  

The first four steps consist of substituting the HAs’ polynomials in \( SP_i \). For each HA, we have the polynomial \( 2kC + kS \) in \( SP_i \), where \( k \) is an integer number, and \( C \) and \( S \) are the carry and sum bits, respectively. Therefore, we can directly substitute it with the addition of HA’s inputs. For example, in the first step of backward rewriting, we have \( 16Z_4 + 8Z_3 \) in \( SP_i \). Since \( Z_3 \) and \( Z_4 \) are the carry and sum bits of the HA, we substitute \( 16Z_4 + 8Z_3 \) with \( 8S_3 + 8S_2 \) to obtain \( SP_1 \). This process is repeated in the next three steps.

In the sixth step of backward rewriting \( SP_{c_3S_3} \rightarrow SP_4 \), the polynomial for the FA with \( c_3, s_3 \) outputs and \( p^{(3)}_1, p^{(3)}_0, c_2 \) inputs is substituted. The output polynomial for the FA is in the form \( -c_3 + 8S_3 \) (see blue polynomial in Eq. (5)), which is different from what we need for a direct input polynomial substitution, i.e. \( 16c_3 + 8S_3 = 8p^{(3)}_1 + 8p^{(3)}_0 + 8Z_2 \).

The deviating form is due to the effect of modular computations in the multiplier, which leads to the change of some bit positions (see red wires in Figure 3 and Figure 4). As a result, the original weight of \( c_3 \) has changed from 16 to 1 under modulo 17. This effect is also reflected in the coefficient of \( c_3 \) in \( SP_i \), i.e. \( 16c_3 \) is converted to \(-c_3\).

As a consequence, the substitution of \( c_3 \) and \( s_3 \) is carried out separately. The result is a polynomial with seven terms in \( SP_4 \) (see red polynomial in Eq. (5)). The first three terms (i.e. \( 8p^{(3)}_1 + 8p^{(3)}_0 + 8Z_2 \)) are the addition of inputs; however, the remaining red terms are extra terms that resulted from a change in the coefficient of \( c_3 \). In the next steps of backward rewriting, these extra terms create many new terms. As a result, the size of \( SP_i \) grows exponentially, e.g. the size of \( SP_1, SP_2, \) and \( SP_3 \) equals 56, 74, and 156, respectively. Moreover, even if we successfully finish the backward rewriting, the remainder is not zero, i.e. all extra terms will be propagated to the remainder. Similarly, the effect of modulo on coefficients can be observed during the backward rewriting of the 2\(^n\) + 1 modular multiplier.

In order to avoid the explosion during the backward rewriting and obtain the zero remainder, we need to prove that the extra terms can be reduced to zero. Alternatively, we can prevent the generation of extra terms during backward rewriting.

3.2 Generation of Vanishing Monomials

Vanishing monomials are non-linear monomials generated during backward rewriting and reduced to zero after several steps or under certain input conditions. The value of a vanishing monomial is equal to zero, i.e. it can be removed from \( SP_i \) immediately. However, it is impossible to detect vanishing monomials purely at the polynomial level during backward rewriting. Thus, they only get canceled out after several steps of backward rewriting and sometimes with considering input conditions. The vanishing monomials create many new monomials and variables before cancellation; therefore, we observe a large increase in the number of monomials and variables, which might lead to a polynomial explosion and verification failure.

In modular multipliers, vanishing monomials can be categorized into two groups: (1) they are generated during backward rewriting and then canceled out after several steps, (2) they get canceled out only under input conditions; thus, without considering the conditions, they propagate to remainder.

We first give an example for the first group of vanishing monomials: Eq. (6) depicts the first steps of backward rewriting for the 2\(^n\) − 1 modular multiplier of Figure 3. In the first step, the polynomial for the HA is substituted. Since the HA does not have a carry output, only \( Z_3 \) and \( Z_4 \) are substituted separately with the corresponding polynomials.

\[ SP := 8Z_3 + 4Z_2 + 2Z_1 + Z_6 - A \times B, \]
\[ SP \xrightarrow{Z_3} SP_1 := 8S_3 + 8S_2 + 4Z_0 + 2Z_6 - A \times B, \]
\[ SP \xrightarrow{Z_4} SP_2 := 8s^{(3)}_3 + 8s^{(3)}_2 + 8Z_0 - Z_1 p^{(3)}_1 p^{(3)}_0 - Z_1 p^{(3)}_2 c_2 + 34s^{(3)}_1 p^{(3)}_0 c_2 + 4S_2 + 2S_1 + S_0 + 1 - A \times B, \]
\[ SP \xrightarrow{Z_4} SP_3 := 8s^{(3)}_3 + 8s^{(3)}_2 + 8s^{(3)}_1 + 4S_2 + 2S_1 + Z_0 - A \times B, \]
\[ \ldots \]  

Now, we give an example for the second group: the PPG stage of 2\(^n\) + 1 modular multiplier in Figure 4 contains several chains of OR gates (see inputs of FAs in the 2nd and 3rd rows). The polynomial for each chain contains several vanishing monomials, which are zero under input conditions. Eq. (7) shows the polynomial for one of the chains in Figure 4. We can prove that the red monomials are reduced to zero under input conditions: In a 2\(^n\) + 1 modular multiplier, the inputs should be always smaller than 2\(^n\) + 1, i.e. \( A, B < 2^n + 1 \). Therefore, if \( A_n (B_n) \), which is the most significant bit of \( A \) (\( B \)) equals 1, the remaining bits must equal 0. On the other hand, if \( A_n (B_n) \) is equal to 0, the remaining bits might have any values. As a result, the product of \( A_n (B_n) \) and any other bit is always equal to zero, i.e. \( A_n A_l = 0 \) (\( B_n B_l = 0 \)). Based on this, we can conclude all the monomials containing \( A_4 A_1 \) or \( B_4 B_1 \), i.e. red monomials, equal zero and can be omitted.

\[ P_{A_3} \land P_{A_4} \land P_{A_5} = P_{A_3} \land P_{A_4} \land (P_{A_0} P_{A_3} - P_{A_0} P_{A_2} - P_{A_2} P_{A_3} - P_{A_0} P_{A_4} P_{A_3} = A_1 B_0 + A_1 B_4 + A_4 B_1 - A_1 A_4 B_0 - A_4 A_1 B_0 + A_4 A_1 B_1 B_0) \]  

However, it is not efficient to first obtain the non-zero remainder and then prove that all remaining monomials equal zero. There are
many chains of OR gates in large $2^n + 1$ modular multipliers and each one introduces several vanishing monomials to the remainder. The huge number of vanishing monomials can lead to an explosion in the number of monomials. Thus, we need to remove vanishing monomials immediately after the substitution of each OR gate polynomial in order to avoid the explosion. In Section 4.2, we introduce a SAT-based technique to remove vanishing monomials locally before global backward rewriting.

### 3.3 Correctness under Output Conditions

Proving that the remainder of SCA-based verification equals zero is not sufficient to prove that a circuit implements a modular multiplier. We have to show that the output $Z < m$ holds. Thus, we have to replace the $2^n - 1$ and $2^n + 1$ modular multipliers that $Z < 2^n - 1$ and $Z < 2^n + 1$ hold, respectively. In the next section, we present a SAT-based approach to check the conditions in the outputs.

### 4 OVERCOMING THE CHALLENGES

In this section, we come up with three techniques to overcome the challenges of verifying modular multipliers.

#### 4.1 Coefficient Correction

In a modular multiplier, all computations are performed modulo $m$, including the weight reduction for the initial and newly generated partial products. We can also take advantage of the modular reduction during backward rewriting. An intermediate polynomial $SP_i$ represents the difference between the output of a normal multiplier (i.e. $Z$) and the expected function (i.e. $AXB$) based on the intermediate signals. However, for a modular multiplier, this difference has to be equal to zero modulo $m$, i.e. the output and the expected function are equivalent modulo $m$. As a consequence, we can reduce $SP_i$ modulo $m$ after performing a substitution.

Eq. (8) shows the fifth and sixth steps of backward rewriting for the modular multiplier of Figure 4. After each substitution, $SP_i$ is reduced modulo $2^4 + 1 = 17$. Thus, the four extra terms that would cause an explosion in the next steps are reduced to zero.

$$SP_4 \mapsto SP_5 := (−c_4 + 8s_4 + 4s_3 + 2s_2 + s_1 + 1 − A × B) \text{ mod } 17,$$

$$SP_5 \mapsto SP_6 := (8p_3^{(3)} + 8p_3^{(0)} + 8s_3 − 32p_4^{(0)} + 17p_2^{(0)} + 17p_2^{(3)} + 34p_0^{(3)} + 4s_2 + 2s_1 + s_0 + 1 − A × B) \text{ mod } 17.$$ (8)

The reduction of $SP_i$ modulo $m$ helps us to avoid the explosion. However, it is not the most efficient way since the generation of extra terms causes peaks in the size of $SP_i$ during backward writing. Thus, we introduce an efficient approach to avoid the generation of extra terms: Whenever we replace a FA or a HA, we check whether the only occurrences of the sum bit $S$ and the carry bit $C$ in the current polynomial are two monomials of the form $KS$ and $2KC$ for some integer $k$. If this is the case, then we can replace the FA/HA as a whole. If the only occurrences of $S$ and $C$ are of the form $KS$ and $PC$, then we check whether $p = 2k + m \cdot q$ for some integer $q$. In this case, we can rewrite $PC$ into $2KC$ (since this does not change the polynomial modulo $m$) and replace the FA/HA as a whole as well.

In all other cases, we have to replace the FA/HA outputs separately. For example, after the fifth step of backward rewriting in Eq. (9), the only occurrences of $S_3$ and $C_4$ are of the form $8S_3$ and $−C_4$. Since $−1 = 2 \cdot 8 + 17 \cdot (−1)$, we can rewrite $−C_4$ into $16C_4$ and replace the FA as a whole. By this, we avoid the generation of extra terms, and thus no peak occurs after each substitution.

$SP_4 \mapsto SP_5 := (−c_4 + 8s_4 + 4s_3 + 2s_2 + s_1 + 1 − A × B) \text{ mod } 17,$

rewrite into $16c_4$

$SP_5 := (16c_4 + 8s_4 + 4s_3 + 2s_2 + s_1 + 1 − A × B) \text{ mod } 17,$

$SP_6 := (8p_3^{(3)} + 8p_3^{(0)} + 8s_3 − 32p_4^{(0)} + 17p_2^{(0)} + 17p_2^{(3)} + 34p_0^{(3)} + 4s_2 + 2s_1 + s_0 + 1 − A × B) \text{ mod } 17.$ (9)

### 4.2 Vanishing Monomials Removal using SAT

In [6, 10], vanishing monomials of the form $C \cdot S \cdot f$ have been removed where $C$ and $S$ are sum and carry outputs of a HA and $f$ is the product of other variables in the monomial, respectively. This was based on the observation that the sum and the carry of a HA cannot be 1 at the same time, so the corresponding monomials will vanish at least once the input signals of the circuit are reached. Here, we generalize this observation: If for a pair of signals $a$ and $b$, both $a$ and $b$ cannot be 1 at the same time by assigning primary input variables in accordance with existing input conditions on the primary inputs, then all monomials of the form $a \cdot b \cdot f$ are vanishing (at least if the input condition is taken care of) and can be removed immediately.

Checking whether $a \cdot b = 0$ can be performed by SAT solving.

We propose a 5-step technique to take advantage of this generalized removal of vanishing monomials in the context of modular multipliers: (1) extracting the Conjunctive Normal Form (CNF) of the modular multiplier using Tseitin transformation, (2) adding the input conditions, i.e. $A, B < m$, as the new clauses to the CNF; for example, the input conditions for the $2^n + 1$ modular multiplier (i.e. $A_3A_2 = 0$ and $B_0B_1 = 0$) can be translated into the new clauses ($A_3 \lor A_2$) and ($B_0 \lor B_1$) for $0 < i < n$; (3) extracting fanout-free cones for the remaining gates/nodes after reverse engineering; (4) performing the local backward rewriting for each fanout-free cone, and (5) checking at each step whether the generated multi-variable monomials are equal to zero using SAT, i.e. if a multi-variable monomial $xy$ appears in our polynomial, we check whether the CNF is UNSAT under the constraint $x \land y = 1$. If yes, $xy$ is a vanishing monomial and can be removed from the polynomial.

Figure 5(a) depicts one of the OR chains in the PPG stage of the $2^n + 1$ modular multiplier in Figure 4. This chain is captured as a fanout-free cone after the reverse engineering. Eq. (10) shows the local backward rewriting steps for the OR chain. In the first step, we prove using our SAT-based technique that $P_{3,4}W_1$ equals zero and can be removed from the polynomial. Similarly, $P_{0,9}W_2$ and $P_{3,3}P_{3,4}$ are removed in the second and third steps, respectively. As a result, we remove all vanishing monomials locally.

$$f \rightarrow P_{4,4} + W_1 − D_{W_0}W_1 \rightarrow P_{4,4} + P_{3,0} + W_0 − D_{W_0}W_1 \rightarrow P_{4,4} + P_{3,0} + P_{4,3} + P_{3,4} − D_{W_0}.$$ (10)

The removal of vanishing monomials can be generalized even more which is useful when considering fanout-free cones in the AIG representation. Figure 5(b) shows the just mentioned OR chain now in AIG. In the first step of local backward rewriting (see Eq. (12)), we cannot find multi-variable monomials which are equal to zero. However, since $P_{3,4} \land \overline{W_1} = 0$, we can replace $P_{3,4}W_1$ with $P_{3,4}$.

$$P_{4,4} \land \overline{W_1} = 0 \Rightarrow P_{4,4} \cdot (1 − W_1) = 0 \Rightarrow P_{4,4} − P_{4,4}W_1 = 0 \Rightarrow P_{4,4}W_1 = P_{4,4}.$$ (11)
We propose a SAT-based technique to check the output condition. Algorithm 1 shows the pseudo-code of our modular verifier. In the first step (Line 1), the remainder is evaluated under the input conditions. Then, the polynomial for each cone is extracted by local backward rewriting. In each step of local backward rewriting, the multi-variable monomial $xy$ is created based on the input and output conditions. We have implemented our verifier in C++. In order to solve the SAT problems, we used the minisat library [2]. All experiments are performed on an Intel(R) Core(TM) i7-8565U CPU with 1.80GHz and 24 GByte of main memory. In order to evaluate the efficiency of our verifier, we consider $2^n - 1$ and $2^{n} + 1$ modular multipliers with different sizes as our benchmarks. The benchmarks have been generated by an extended version of the open-source multiplier generator GenMul.

Table 1 and Table 2 report the verification results for $2^n - 1$ and $2^n + 1$ multipliers, respectively. The Time-out (T.O.) has been set to 48 hours. The first column Size denotes the size of the multiplier based on the two inputs’ bit-width. The verification data is reported in the second column Data consisting of four sub-columns: #Nodes gives the number of nodes in the AIG representation of the circuit. #Simpl. Mono. refers to the total sum of multi-variable monomials which are removed or simplified during the local backward rewriting (see Section 4.2). #MaxPoly presents the maximum size of $SP_i$ during the global backward rewriting based on the number of monomials. #Rem. presents the size of remainder after the backward rewriting. The remainder should be evaluated to zero under input conditions. The run-time (in seconds) of our proposed method is reported in detail in the third column Run-time consisting of five sub-columns: Rev. Eng. reports the required time for identifying atomic blocks, including HAs and FAs in the AIG representation of a multiplier. Local Backw. Rewriting reports the time for local backward rewriting of fanout-free cones and simplifying multi-variable monomials. Glob. Backw. Rewriting reports the time for the global backward rewriting phase. Check Out. Cond. presents the time for checking the condition on the output. The overall run-time of our proposed method is presented in Overall. The run-time of a commercial formal verification tool is given in the fourth column Com.. The fifth column Pure SCA reports the run-time of the Pure SCA-based verification method without integrating the coefficient correction and vanishing monomials removal techniques. Finally, the last column SAT presents the run-time of SAT-based verification method. We generated a reference model by synthesizing a high-level modular multiplier description into an AIG using %blast command in abc [1], and then use SAT to prove that our benchmark and the reference model are equivalent.

The results in Table 1 and Table 2 confirm that our verifier can prove the correctness of modular multipliers with more than 3M AIG nodes, e.g. $2^n - 1$ and $2^n + 1$ modular multipliers with $512 \times 512$ input sizes. The number of simplified monomials for $2^n - 1$ multipliers in Table 1 equals 1, independent of the multiplier size. This monomial is originating from the HA that generates the MSB bit of output (see the bottom left HA in Figure 3) and appears in the first step of backward rewriting (see Eq. (6)). If this monomial remains in $SP_i$, it generates several monomials and variables in the next steps. Finally, these monomials are reduced to zero after several steps of backward rewriting. Thus, removing a small number of vanishing monomials can avoid a big peak size of the intermediate polynomials. Figure 6 shows the size of the intermediate polynomials based on the number of monomials (Figure 6(a)) and the number of variables (Figure 6(b)) for the $2^n - 1$ modular multiplier with $8 \times 8$ input size. In the absence of the vanishing removal technique, an undetected vanishing monomial generates several new monomials containing many variables (see blue lines). On the other hand, removing a vanishing monomial using our proposed technique prevents the big peak size of the intermediate

---

\[ 1 \rightarrow f \rightarrow 1 + P_{x4}V_1 - V_1 \rightarrow 1 + P_{x4} + P_{y0}V_2 - V_2 \]
\[ P_{x4} \]
\[ P_{y0} \]
\[ 1 + P_{x4} + P_{y0} - 1 + P_{x3} + P_{y4} - P_{x4}P_{y4} \] (12)

Consequently, we can extend the fifth step of our proposed technique in order to also support the removal of vanishing monomials in AIGs: at each step of local backward rewriting, if a multi-variable monomial $xy$ appears in our polynomial, we check whether the CNF is unsatisfiable under one of the three constraints: $x \land y = 1$, $x \land \overline{y} = 1$, or $\overline{x} \land y = 1$. If yes, we replace $xy$ with zero, $x$, or $y$, respectively.

### 4.3 Checking Output Conditions using SAT

We propose a SAT-based technique to check the output condition $Z < m$ for a modular multiplier: the CNF of the circuit including the extra clauses related to the input conditions is available from the previous section. Thus, we have to prove that under constraint $Z \geq m$, the CNF is always unsatisfiable; thus, the output never has a value in this range. To do this, we translate $Z \geq m$ into new clauses and add them to the CNF. Then, we use a SAT-solver to check whether the new CNF is unsatisfiable. As already shown in Section 4.2, for $m = 2^n + 1$, the output condition $Z \geq m$ can be translated into $\forall_{0 \leq i < n} Z_i$. Our experiments show that unsatisfiability of this condition can be easily checked using a SAT solver.

### 5 SCA-BASED MODULAR VERIFIER

Algorithm 1 shows the pseudo-code of our modular verifier. In the first step, the specification polynomial $SP$ 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 [7] (Line 2). The rest of the nodes which are not part of any atomic blocks are grouped based on the fanout-free regions (Line 3). The CNF of the multiplier is extracted by Tseitin transformation (Line 4). Then, the polynomial for each cone is extracted by local backward rewriting. In each step of local backward rewriting, the multi-variable monomials are identified using a dedicated reverse engineering technique [7] (Line 5). Subsequently, global backward rewriting is performed by substituting atomic block and cone polynomials in $SP_i$. The coefficients are corrected after each substitution if needed (Line 6). Then, the remainder is evaluated under the input conditions to see whether it can be reduced to zero (Line 7). Finally, the output condition is checked using SAT (Line 8). If the evaluated remainder equals zero and the output condition holds, the circuit is correct; otherwise, it is buggy (Line 10–Line 12).
Table 1: Verification information for $2^n - 1$ modular multipliers

<table>
<thead>
<tr>
<th>Size</th>
<th>Data</th>
<th>Run-time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Simpl.</td>
<td>Local Backw.</td>
</tr>
<tr>
<td></td>
<td>Mono.</td>
<td>Rewriting</td>
</tr>
<tr>
<td></td>
<td>MaxPoly</td>
<td>Check Out.</td>
</tr>
<tr>
<td></td>
<td>Rem.</td>
<td></td>
</tr>
<tr>
<td>#Nodes</td>
<td>4x4</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>8x8</td>
<td>744</td>
</tr>
<tr>
<td></td>
<td>16x16</td>
<td>3,096</td>
</tr>
<tr>
<td></td>
<td>32x32</td>
<td>12,344</td>
</tr>
<tr>
<td></td>
<td>64x64</td>
<td>49,784</td>
</tr>
<tr>
<td></td>
<td>128x128</td>
<td>197,880</td>
</tr>
<tr>
<td></td>
<td>256x256</td>
<td>787,960</td>
</tr>
<tr>
<td></td>
<td>512x512</td>
<td>3,152,888</td>
</tr>
</tbody>
</table>

Table 2: Verification information for $2^n + 1$ modular multipliers

<table>
<thead>
<tr>
<th>Size</th>
<th>Data</th>
<th>Run-time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Simpl.</td>
<td>Local Backw.</td>
</tr>
<tr>
<td></td>
<td>Mono.</td>
<td>Rewriting</td>
</tr>
<tr>
<td></td>
<td>MaxPoly</td>
<td>Check Out.</td>
</tr>
<tr>
<td></td>
<td>Rem.</td>
<td></td>
</tr>
<tr>
<td>#Nodes</td>
<td>4x4</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>8x8</td>
<td>599</td>
</tr>
<tr>
<td></td>
<td>16x16</td>
<td>2,728</td>
</tr>
<tr>
<td></td>
<td>32x32</td>
<td>11,591</td>
</tr>
<tr>
<td></td>
<td>64x64</td>
<td>47,752</td>
</tr>
<tr>
<td></td>
<td>128x128</td>
<td>193,799</td>
</tr>
<tr>
<td></td>
<td>256x256</td>
<td>780,808</td>
</tr>
<tr>
<td></td>
<td>512x512</td>
<td>3,134,471</td>
</tr>
</tbody>
</table>

Figure 6: Size of $S_{P_i}$ for a $2^n - 1$ multiplier with $8 \times 8$ input size

7 CONCLUSION

In this paper, we presented our modular verifier which combines SCA and SAT to prove the correctness of $2^n - 1$ and $2^n + 1$ modular multipliers. We overcame the challenges of formal verification by integrating coefficient correction and SAT-based local vanishing removal into the SCA. We also proposed a SAT-based technique to check whether the output condition holds for a modular multiplier. The experiments using an extensive set of $2^n + 1$ and $2^n - 1$ modular multipliers demonstrated the efficiency of our verifier in proving the correctness of million-gate benchmarks.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) within the project VerA (GR 3104/6-1, SCHO 894/5-1, and DR 297/37-1) and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

REFERENCES