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 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.

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530605

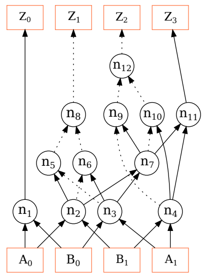


Figure 1: 2×2 mult

$$SP := 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - A \times B$$
$$SP \xrightarrow{P_{Z_3}} SP_1 := 8n_{11} + 4Z_2 + 2Z_1 + Z_0 - A \times B$$
$$SP_1 \xrightarrow{P_{Z_2}} SP_2 := 8n_{11} + 4 - 4n_{12} + 2Z_1 + Z_0 - A \times B$$
$$SP_2 \xrightarrow{P_{n_{12}}} SP_3 := 8n_{11} + 4n_9 + 4n_{10} - 4n_9n_{10} + 2Z_1 + Z_0 - A \times B$$
$$SP_3 \xrightarrow{P_{n_{11}}} SP_4 := 8n_4n_7 + 4n_9 + 4n_{10} - 4n_9n_{10} + 2Z_1 + Z_0 - A \times B$$
$$\vdots$$
$$SP_{13} \xrightarrow{P_{n_3}} SP_{14} := n_1 + n_2 - (A_0B_0 + A_0B_1)$$
$$SP_{14} \xrightarrow{P_{n_2}} SP_{15} := n_1 - (A_0B_0)$$
$$SP_{15} \xrightarrow{P_{n_1}} r := 0$$

Figure 2: Backward rewriting steps

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 multi-operand 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 $SP = 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - (2A_1 + A_0)(2B_1 + B_0)$, where $8Z_3 + 4Z_2 + 2Z_1 + Z_0$ 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:

$$\begin{aligned} &z = n_i \Rightarrow p_N := z - n_i, && z = n_i \wedge n_j \Rightarrow p_N := z - n_in_j,\\ &z = \neg n_i \Rightarrow p_N := z - 1 + n_i, && z = \neg n_i \wedge n_j \Rightarrow p_N := z - n_j + n_in_j,\\ &z = \neg n_i \wedge \neg n_j \Rightarrow p_N := z - 1 + n_i + n_j - n_in_j. \end{aligned} \tag{1}$$

The extracted node polynomials are in the form $P_N = x - tail(P_N)$, where x is the node's output, and $tail(P_N)$ is a function based on the node's inputs. 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_i by a node polynomial $P_{N_i} = x_i - tail(P_{N_i})$ is equivalent to substituting x_i with $tail(P_{N_i})$ in SP_i. For example, dividing SP_3 by $P_{n_{11}}$ in Figure 2 is equivalent to substituting n_{11} with $tail(P_{n_{11}}) = n_4n_7$ in SP_3. The process of step-wise division (substitution) is called *backward rewriting*. We refer to this intermediate polynomial as SP_i in the rest of the paper.

2.3 Modular Multipliers

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

$$Z = A \times B \bmod m, \tag{2}$$

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

$$A, B < m \qquad \text{and} \qquad Z < m. \tag{3}$$

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 $P_{i,j} = A_i \wedge B_j$. 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.

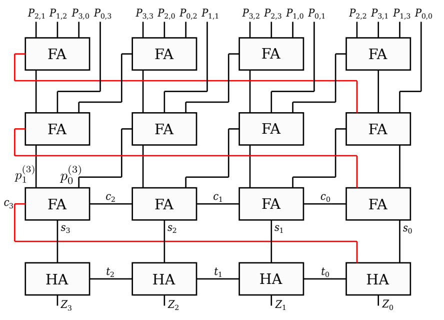

$P_{2,1}$ $P_{1,2}$ $P_{3,0}$ $P_{0,3}$
$P_{3,3}$ $P_{2,0}$ $P_{0,2}$ $P_{1,1}$
$P_{3,2}$ $P_{2,3}$ $P_{1,0}$ $P_{0,1}$
$P_{2,2}$ $P_{3,1}$ $P_{1,3}$ $P_{0,0}$
FA
HA
$p_1^{(3)}$
$p_0^{(3)}$
c_3
c_2
c_1
c_0
s_3
s_2
s_1
s_0
t_2
t_1
t_0
Z_3
Z_2
Z_1
Z_0

Figure 3: $2^n - 1$ **modular multiplier**

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.

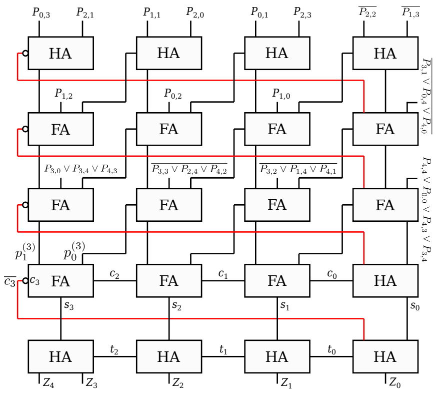

$P_{0,3}$ $P_{2,1}$
$P_{1,1}$ $P_{2,0}$
$P_{0,1}$ $P_{2,3}$
$\overline{P_{2,2}}$ $\overline{P_{1,3}}$
HA
FA
$P_{1,2}$
$P_{0,2}$
$P_{1,0}$
$\overline{P_{3,1} \vee P_{0,4} \vee P_{4,0}}$
$P_{3,0} \vee P_{3,4} \vee P_{4,3}$
$\overline{P_{3,3} \vee P_{2,4} \vee P_{4,2}}$
$\overline{P_{3,2} \vee P_{1,4} \vee P_{4,1}}$
$P_{4,4} \vee P_{0,0} \vee P_{4,3} \vee P_{3,4}$
$p_1^{(3)}$
$p_0^{(3)}$
$\overline{c_3}$
c_3
c_2
c_1
c_0
s_3
s_2
s_1
s_0
t_2
t_1
t_0
Z_4
Z_3
Z_2
Z_1
Z_0

Figure 4: $2^n + 1$ **modular multiplier**

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:

$$\begin{aligned} HA(in: X, Y \quad out: C, S) \quad &\Rightarrow \quad 2C + S = X + Y, \\ FA(in: X, Y, Z \quad out: C, S) \quad &\Rightarrow \quad 2C + S = X + Y + Z. \end{aligned} \tag{4}$$

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.

$$\begin{aligned} &SP := 16Z_4 + 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - A \times B, \\ &SP \xrightarrow{Z_4, Z_3} SP_1 := 8s_3 + 8t_2 + 4Z_2 + 2Z_1 + Z_0 - A \times B, \\ &\dots \\ &SP_4 \xrightarrow{\overline{c_3}} SP_5 := -c_3 + 8s_3 + 4s_2 + 2s_1 + s_0 + 1 - A \times B, \\ &SP_5 \xrightarrow{c_3, S_3} SP_6 := 8p_1^{(3)} + 8p_0^{(3)} + 8c_2 - 17p_1^{(3)}p_0^{(3)} - 17p_1^{(3)}c_2 - 17p_0^{(3)}c_2 \\ &\qquad\qquad +34p_1^{(3)}p_0^{(3)}c_2 + 4s_2 + 2s_1 + s_0 + 1 - A \times B, \\ &\quad \dots \end{aligned} \tag{5}$$

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_4 and Z_3 are the carry and sum bits of the HA, we substitute $16Z_4 + 8Z_3$ with $8s_3 + 8t_2$ to obtain SP_1. This process is repeated in the next three steps.

In the sixth step of backward rewriting $SP_5 \longrightarrow SP_6$, the polynomial for the FA with c_3, s_3 outputs and $p_1^{(3)}$, $p_0^{(3)}$, 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. $16c_3 + 8s_3 = 8p_1^{(3)} + 8p_0^{(3)} + 8c_2$.

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_5, 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_6 (see red polynomial in Eq. (5)). The first three terms (i.e. $8p_1^{(3)} + 8p_0^{(3)} + 8c_2$) 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_7, SP_8, and SP_9 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 (i.e. sum bit) is substituted with $s_3 + t_2 - 2s_3t_2$. The red monomial, i.e. s_3t_2, in SP_1 is a vanishing monomial. It remains in the calculations for a long time and only gets completely canceled out after 8 steps of backward rewriting. Detecting these vanishing monomials and removing them immediately is essential to keep the size of SP_i small.

$$\begin{aligned} &SP := 8Z_3 + 4Z_2 + 2Z_1 + Z_0 - A \times B, \\ &SP \xrightarrow{Z_3} SP_1 := 8s_3 + 8t_2 - 16s_3t_2 + 4Z_2 + 2Z_1 + Z_0 - A \times B, \\ &SP_1 \xrightarrow{Z_2, t_2} SP_2 := 8s_3 - 16s_3s_2t_1 + 4s_2 + 4t_1 + 2Z_1 + Z_0 - A \times B, \\ &SP_2 \xrightarrow{Z_1, t_1} SP_3 := 8s_3 - 16s_3s_2s_1t_0 + 4s_2 + 2s_1 + 2t_0 + Z_0 - A \times B, \\ &\dots \end{aligned} \tag{6}$$

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. $A_nA_i = 0$ ($B_nB_i = 0$). Based on this, we can conclude all the monomials containing A_4A_i or B_4B_i, i.e. red monomials, equal zero and can be omitted.

$$\begin{aligned} &P_{3,0} \vee P_{3,4} \vee P_{4,3} = \\ &P_{3,0} + P_{3,4} + P_{4,3} - P_{3,0}P_{3,4} - P_{3,0}P_{4,3} - P_{3,4}P_{4,3} + P_{3,0}P_{3,4}P_{4,3} = \\ &A_3B_0 + A_3B_4 + A_4B_3 - A_3B_4B_0 - A_4A_3B_3B_0 - A_4A_3B_4B_3 + A_4A_3B_4B_3B_0. \end{aligned} \tag{7}$$

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 guarantee as well that for the output $Z < m$ holds. Thus, we have to show for 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. $A \times B$) 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.

$$
\begin{aligned}
SP_4 &\xrightarrow{\overline{c_3}} SP_5 := (-c_3 + 8s_3 + 4s_2 + 2s_1 + s_0 + 1 - A \times B) \bmod 17,\\
SP_5 &\xrightarrow{c_3, S_3} SP_6 := (8p_1^{(3)} + 8p_0^{(3)} + 8c_2 - \cancel{17p_1^{(3)}p_0^{(3)}} - \cancel{17p_1^{(3)}}c_2 - \cancel{17p_0^{(3)}}c_2\\
&\qquad \cancel{+34p_1^{(3)}p_0^{(3)}}c_2 + 4s_2 + 2s_1 + s_0 + 1 - A \times B) \bmod 17. \qquad (8)
\end{aligned}
$$

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_3 are of the form $8s_3$ and $-c_3$. Since $-1 = 2 \cdot 8 + 17 \cdot (-1)$, we can rewrite $-c_3$ into $16c_3$ and replace the FA as a whole. By this, we avoid the generation of extra terms, and thus no peak occurs after each substitution.

$$
\begin{aligned}
SP_4 &\xrightarrow{\overline{c_3}} SP_5 := (\underbrace{-c_3}_{\text{rewritie into } 16c_3} +8s_3 + 4s_2 + 2s_1 + s_0 + 1 - A \times B) \bmod 17,\\
&SP_5 := (16c_3 + 8s_3 + 4s_2 + 2s_1 + s_0 + 1 - A \times B) \bmod 17,\\
SP_5 &\xrightarrow{c_3, s_3} SP_6 := (8p_1^{(3)} + 8p_0^{(3)} + 8c_2 + 4s_2 + 2s_1 + s_0 + 1 - A \times B) \bmod 17. \qquad (9)
\end{aligned}
$$

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. $A_n A_i = 0$ and $B_n B_i = 0$) can be translated into the new clauses $(\overline{A_n} \vee \overline{A_i})$ and $(\overline{B_n} \vee \overline{B_i})$ for $0 \leqslant i < n$, (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 $x \wedge y = 1$. 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 $P_{4,4}W_1$ equals zero and can be removed from the polynomial. Similarly, $P_{0,0}W_2$ and $P_{4,3}P_{3,4}$ are removed in the second and third steps, respectively. As a result, we remove all vanishing monomials locally.

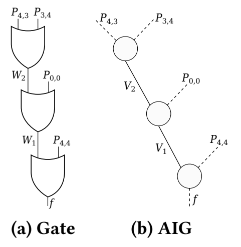


Figure 5: Chain of ORs

$$
\begin{aligned}
f &\rightarrow P_{4,4} + W_1 - \cancel{P_{4,4}W_1} \rightarrow P_{4,4} + P_{0,0} + W_2 - \cancel{P_{0,0}W_2}\\
&\rightarrow P_{4,4} + P_{0,0} + P_{4,3} + P_{3,4} - \cancel{P_{4,3}P_{3,4}}. \qquad (10)
\end{aligned}
$$

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 $P_{4,4} \wedge \overline{V_1} = 0$, we can replace $P_{4,4}V_1$ with $P_{4,4}$:

$$
P_{4,4} \wedge \overline{V_1} = 0 \Rightarrow P_{4,4} \cdot (1 - V_1) = 0 \Rightarrow P_{4,4} - P_{4,4}V_1 = 0 \Rightarrow P_{4,4}V_1 = P_{4,4}. \qquad (11)
$$

Algorithm 1 Modular verifier

```
Require: Modular multiplier AIG $G$
Ensure: TRUE if the circuit is correct, and FALSE otherwise
 1: $SP \leftarrow$ CreateSP($G$)
 2: $AB \leftarrow$ ReverseEngineering($G$)                    ▷ $AB$ is the set of atomic blocks
 3: $C \leftarrow$ FindFanoutFreeCones($G$, $AB$)              ▷ $C$ is the set of fanout-free cones
 4: $CNF \leftarrow$ ExtractCNF($G$)
 5: $PF \leftarrow$ LocalBackwardRewriting($C$, $CNF$)          ▷ $PF$ is the set of cone polynomials
 6: $r \leftarrow$ GlobalBackwardRewriting($SP$, $PF$, $AB$)     ▷ $r$ is the remainder
 7: $r_e \leftarrow$ EvaluateRemainder($r$)
 8: $o \leftarrow$ CheckOutputCondition($CNF$)                 ▷ if $o$ = TRUE, the condition is satisfied
 9: if ($r_e$ == 0  &  $o$ == TRUE) then
10:     return TRUE
11: else
12:     return FALSE
```

This simplification prevents the generation of vanishing monomials in the next steps. Similarly, the same simplification can be applied in the second step by replacing $P_{0,0}V_2$ with $P_{0,0}$. In the final step, $P_{4,3}P_{3,4}$ is directly removed since $P_{4,3} \wedge P_{3,4} = 0$.

$$1 - f \rightarrow 1 + \underbrace{P_{4,4}V_1 - V_1}_{P_{4,4}} \rightarrow 1 + P_{4,4} + \underbrace{P_{0,0}V_2 - V_2}_{P_{0,0}}$$
$$\rightarrow 1 + P_{4,4} + P_{0,0} - 1 + P_{4,3} + P_{3,4} - \cancel{P_{4,3}P_{3,4}}. \quad (12)$$

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 UNSAT under one of the three constraints: $x \wedge y = 1$, $x \wedge \overline{y} = 1$, or $\overline{x} \wedge y = 1$. 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 $Z \geqslant m$, the CNF is always UNSAT; thus, the output never has a value in this range. To do this, we translate $Z \geqslant m$ into new clauses and add them to the CNF. Then, we use a SAT-solver to check whether the new CNF is UNSAT. As already shown in Section 4.2, for $m = 2^n + 1$, the output condition $Z \geqslant m$ can be translated into $\bigvee_{0 \leq i < n} Z_n Z_i$. 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, e.g. xy, are checked using SAT to see whether the simplification is possible (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).

6 EXPERIMENTS

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[1].

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* refers to the time needed 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 512×512 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 8×8 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

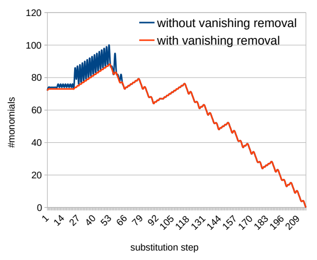
[1]http://sca-verification.org/genmul

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

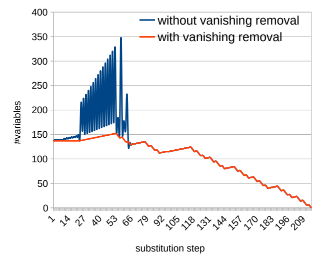
Size	Data				Run-time (seconds)							
	#Nodes	#Simpl. Mono.	#MaxPoly	#Rem.	Rev. Eng.	Local Backw. Rewriting	Glob. Backw. Rewriting	Check Out. Cond.	Overall	Com.	Pure SCA	SAT
4×4	160	1	36	0	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.01	0.01
8×8	744	1	136	0	<0.01	<0.01	<0.01	<0.01	0.01	<0.01	T.O.	2.53
16×16	3,096	1	528	0	0.04	0.01	<0.01	<0.01	0.05	66.00	T.O.	T.O.
32×32	12,344	1	2,080	0	0.13	0.03	0.05	<0.01	0.21	T.O.	T.O.	T.O.
64×64	49,784	1	8,256	0	0.58	0.14	0.68	<0.01	1.40	T.O.	T.O.	T.O.
128×128	197,880	1	32,896	0	2.62	0.98	11.55	<0.01	15.16	T.O.	T.O.	T.O.
256×256	787,960	1	131,328	0	10.57	8.37	365.45	<0.01	384.39	T.O.	T.O.	T.O.
512×512	3,152,888	1	524,800	0	42.28	80.19	6,826.52	0.01	6,948.99	T.O.	T.O.	T.O.

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

Size	Data				Run-time (seconds)							
	#Nodes	#Simpl. Mono.	#MaxPoly	#Rem.	Rev. Eng.	Local Backw. Rewriting	Glob. Backw. Rewriting	Check Out. Cond.	Overall	Com.	Pure SCA	SAT
4×4	112	11	36	0	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.01	0.01
8×8	599	18	136	0	0.01	<0.01	<0.01	<0.01	0.01	<0.01	T.O.	2.36
16×16	2,728	35	528	0	0.03	0.01	<0.01	<0.01	0.05	60.00	T.O.	T.O.
32×32	11,591	66	2,080	0	0.12	0.05	0.05	<0.01	0.22	T.O.	T.O.	T.O.
64×64	47,752	131	8,256	0	0.55	0.24	0.61	<0.01	1.39	T.O.	T.O.	T.O.
128×128	193,799	258	32,896	0	2.36	2.64	10.67	<0.01	15.66	T.O.	T.O.	T.O.
256×256	780,808	515	131,328	0	10.64	26.20	416.47	<0.01	453.31	T.O.	T.O.	T.O.
512×512	3,134,471	1,026	524,800	0	42.16	245.40	6,777.51	0.01	7,065.08	T.O.	T.O.	T.O.



(a) Number of monomials



(b) Number of variables

Figure 6: Size of SP_i for a $2^n - 1$ multiplier with 8×8 input size

polynomials (see red lines). Please note that the results in Figure 6 are obtained in the presence of our coefficient correction technique; otherwise, an explosion happens in the number of monomials and the verification times-out (see Table 1, column *#Pure SCA*). Moreover, many multi-variable monomials are simplified for the $2^n + 1$ modular multipliers, which are related to the chain of OR gates (see Table 2, column *#Simpl. Mono.*).

For the both $2^n - 1$ and $2^n + 1$ modular multipliers, the size of remainder after the backward rewriting is equal to zero. This stems from the fact that our vanishing removal technique removes all the vanishing monomials locally and avoids the propagation of them into the remainder. As a result, the remainder evaluation step in Algorithm 1 can be skipped.

Our tool can verify very large multipliers with 512×512 input sizes in less than 2 hours. On the other hand, the commercial tool only verifies multipliers up to 16×16, and it times out for the bigger benchmarks. The pure SCA-based verification without coefficient correction and local vanishing removal techniques can obtain the remainder for only 4×4 multipliers, and it times out for the bigger designs. The SAT-based verification also works for small multipliers up to 8×8 and fails for the others. Thus, verifying modular multipliers by only SAT is not feasible. In contrast, checking the output condition using SAT is easy, since it can be proven by local reasoning in the SAT solver (see column *Check Out. Cond.*).

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

[1] Abc: A system for sequential synthesis and verification. available at https://people.eecs.berkeley.edu/~alanmi/abc/, 2018.
[2] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.
[3] D. Gallaher, F. Petry, and P. Srinivasan. The digit parallel method for fast RNS to weighted number system conversion for specific moduli ($2^k - 1$,2^k,$2^k + 1$). *IEEE Trans. Circuits Syst. II*, 44(1):53–57, 1997.
[4] D. Kaufmann, A. Biere, and M. Kauers. Verifying large multipliers by combining SAT and computer algebra. In *FMCAD*, pages 28–36, 2019.
[5] X. Lai. *On the design and security of block ciphers.* PhD thesis, ETH Zurich, Zürich, Switzerland, 1992.
[6] A. Mahzoon, D. Große, and R. Drechsler. PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In *ICCAD*, pages 129:1–129:8, 2018.
[7] A. Mahzoon, D. Große, and R. Drechsler. RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In *DAC*, pages 185:1–185:6, 2019.
[8] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler. Towards formal verification of optimized and industrial multipliers. In *DATE*, pages 544–549, 2020.
[9] B. Parhami. *Computer Arithmetic : Algorithms and Hardware Designs.* Oxford University Press Inc, 2002.
[10] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler. Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In *DATE*, pages 1048–1053, 2016.
[11] C. Scholl and A. Konrad. Symbolic computer algebra and SAT based information forwarding for fully automatic divider verification. In *DAC*, pages 1–6, 2020.
[12] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler. Verifying dividers using symbolic computer algebra and don't care optimization. In *DATE*, pages 1110–1115, 2021.
[13] L. Sousa and R. Chaves. A universal architecture for designing efficient modulo 2^n+1 multipliers. *IEEE Trans. Circuits Syst. I*, 52-I(6):1166–1178, 2005.
[14] H. T. Vergos and C. Efstathiou. Design of efficient modulo $2^n + 1$ multipliers. *IET Comput. Digit. Tech.*, 1(1):49–57, 2007.
[15] C. Walther. Formally verified montgomery multiplication. In *CAV*, pages 505–522, 2018.
[16] C. Yu, M. Ciesielski, and A. Mishchenko. Fast algebraic rewriting based on and-inverter graphs. *TCAD*, 37(9):1907–1911, 2017.
[17] R. Zimmermann. Efficient VLSI implementation of modulo ($2^n \pm 1$) addition and multiplication. In *ARITH*, pages 158–167, 1999.