Nullstellensatz-Proofs for Multiplier Verification

Daniela Kaufmann, Armin Biere*

Johannes Kepler University, Linz, Austria

Abstract. Automated reasoning techniques based on computer algebra are an essential ingredient in formal verification of gate-level multiplier circuits. Generating and independently checking proof certificates helps to validate the verification results. Two algebraic proof systems, Nullstellensatz and polynomial calculus, are well-known in proof complexity. The practical application of the polynomial calculus has been studied recently. However, producing and checking Nullstellensatz certificates for multiplier verification has not been considered so far. In this paper we show how Nullstellensatz proofs can be generated as a by-product of multiplier verification and present our Nullstellensatz proof checker Nuss-Checker. Additionally, we prove quadratic upper bounds on the proof size for simple array multipliers.

1 Introduction

Formal verification aims to prove or disprove the correctness of a given system with respect to a certain specification. Nonetheless, the verification process might not be correct and contain errors. Thus it is common to produce proof certificates, which can be checked by stand-alone proof checkers in order to increase the confidence in the results of the verification process.

For example, many applications of formal verification use satisfiability (SAT) solving and various resolution or clausal proof formats [17], such as DRUP [13,14], DRAT [18], and LRAT [11] are available to validate the verification results. In the annual SAT competition it is even required to provide certificates since 2013.

However, in certain applications SAT solving cannot be applied successfully. For instance formal verification of arithmetic circuits, more precisely multiplier circuits is considered to be hard for SAT solving. The current state of the art in verifying multiplier circuits relies on computer algebra [9, 24, 31, 32]. In this approach the circuit is modeled as a set of polynomials and it is shown that the specification, also encoded as a polynomial, is implied by the polynomials that are induced by the circuit. That is, for each gate in the circuit a polynomial is defined that captures the relations of the inputs and output of the gate. These gate polynomials generate a Gröbner basis [7]. Preprocessing techniques based on variable elimination are applied to rewrite and thus simplify the Gröbner basis [24, 31]. After preprocessing the specification polynomial is reduced by the rewritten gate polynomials until no further reduction is possible. The given multiplier is correct if and only if the final result is zero.

* This work is supported by the LIT AI Lab funded by the State of Upper Austria.

Besides circuit verification, algebraic reasoning in combination with SAT solving [6] is successfully used to solve complex combinatorial problems, e.g., finding faster ways for matrix multiplication [19, 20], computing small unit-distance graphs with chromatic number 5 [16], or solving the Williamson conjecture [5], and has possible future applications in cryptanalysis [8,40]. All these applications raise the need to invoke algebraic proof systems for proof validation.

Two algebraic proof systems are commonly known in the proof complexity community, polynomial calculus (PC) [10] and Nullstellensatz (NSS) [3]. Both systems are well-studied, with the main focus on deriving complexity measures, such as degree and proof size, e.g., [2,22,33,34]. Proofs in PC allow us to dynamically capture that a polynomial can be derived from a given set of polynomials using algebraic ideal theory. However, PC as defined in [10], is not suitable for practical proof checking [23], thus we introduced the practical algebraic calculus (PAC) in [37] that can be checked efficiently.

Proofs in NSS capture whether a polynomial can be represented as a linear combination from a given set of polynomials. Since NSS proofs are more static we made the following conjecture for the application of multiplier circuit verification in [23]: "In a correct NSS proof we would also need to express the rewritten polynomials as a linear combination of the given set of polynomials and thus lose the optimized representation, which will most likely lead to an exponential blow-up of monomials in the NSS proof."

In this paper we show that this conjecture has to be rejected, at least for those multiplier architectures considered in this paper. We introduce how NSS proofs can be produced in our verification tool AMulet [24, 26] and our experimental results demonstrate that we are able to generate concise NSS proofs. For simple array multipliers, which consist only of full- and half-adders that are arranged in a grid-like structure, we prove quadratic bounds for the proof size. Furthermore, we present our NSS proof checker Nuss-Checker and discuss important design decisions that help to improve the checking time and memory usage.

2 Preliminaries

We describe our state-of-the-art approach in gate-level multiplier verification using computer algebra [24], and give an introduction to the algebraic proof systems PC, PAC, and NSS.

2.1 Multiplier Verification

Digital circuits are used in computers and digital systems and compute binary digital values for the logical function they implement, given binary values at the input. The computation is usually realized by logic gates, representing simple Boolean functions, such as NOT, AND, OR. The specification of a circuit is a desired relation between its inputs and outputs and the goal of verification is to formally prove that the circuit fulfills its specification, i.e., for all inputs the outputs of the circuit match the specification.

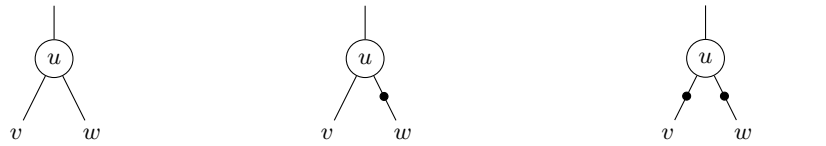


Fig. 1: All polynomial encodings covered by AIG nodes.

In our setting, we consider gate-level integer multiplier circuits C with $2n$ input bits $a_0, \ldots, a_{n-1}$, $b_0, \ldots, b_{n-1} \in \{0,1\}$ and $2n$ output bits $s_0, \ldots, s_{2n-1} \in \{0,1\}$. The internal gates are denoted by $g_1, \ldots, g_k \in \{0,1\}$. Let R be a commutative ring with unity and let $R[a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, g_1, \ldots, g_k, s_0, \ldots, s_{2n-1}] = R[X]$. Since we consider integer multipliers, we will later set the ring $R = \mathbb{Z}$, but for now let us keep the more general ring R. The multiplier C is correct iff for all possible inputs a_i, $b_i \in \{0,1\}$ the following specification $\mathcal{L} = 0$ holds:

$$\mathcal{L} = -\sum_{i=0}^{2n-1} 2^i s_i + \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right) \tag{1}$$

A common representation of circuits are And-Inverter-Graphs (AIG) [28], which are directed acyclic graphs consisting of two-input nodes that represent logical conjunction. The edges may contain a marking that indicates logical negation. The semantics of each node implies a polynomial relation, cf. Fig. 1.

Let $G(C) \subseteq R[X]$ be the set of polynomials that contains for each gate of the given circuit C the corresponding polynomial of Fig. 1, with u, v, and w replaced by corresponding variables $x \in X$. We call these polynomials *gate constraints*.

All variables $x \in X$ are Boolean and we enforce this property by adding for each variable a *Boolean value constraint* $x(x-1) = 0$. Let $B(Y) = \{y(1-y) \mid y \in Y\} \subseteq R[X]$ for $Y \subseteq X$, be the set of Boolean value constraints for Y.

On the set of terms we fix an order $\leq$ such that for all terms τ, σ_1, σ_2 it holds that $1 \leq \tau$ and $\sigma_1 \leq \sigma_2 \Rightarrow \tau\sigma_1 \leq \tau\sigma_2$. An order is called a *lexicographic term order* if for all terms $\sigma_1 = x_1^{d_1} \cdots x_r^{d_r}$, $\sigma_2 = x_1^{e_1} \cdots x_r^{e_r}$ we have $\sigma_1 < \sigma_2$ iff $\exists i \in \mathbb{N}$ with $d_j = e_j$ for all $j < i$, and $d_i < e_i$. For a polynomial $p = c\tau + \cdots$ the largest term τ (w.r.t. $\leq$) is called the *leading term* $\mathrm{lt}(p) = \tau$. Furthermore $\mathrm{lc}(p) = c$ is called the *leading coefficient* and $\mathrm{lm}(p) = c\tau$ is called the *leading monomial* of p.

Definition 1 ([24]). *Let $P \subseteq R[X]$. If for a term order, all leading terms of P only consist of a single variable with exponent 1, are unique, and further $\mathrm{lc}(p) \in R^\times$ for all $p \in P$, we say P has* unique monic leading terms *(UMLT).*

We order the polynomials in $G(C)$ according to a lexicographic term order, such that the output variable of a gate is always greater than the inputs of the gate. Such an order is also called *reverse topological term order* [29]. It immediately follows that $G(C)$ has UMLT. Let $X_0 \subseteq X$ be the set of all variables that do not occur as leading terms in $G(C)$ and let $J(C) = \langle G(C) \cup B(X_0) \rangle \subseteq R[X]$. The circuit fulfills its specification if and only if we can derive that $\mathcal{L} \in J(C)$ [24].

For the remainder of this section let $R = \mathbb{Z}$. Because of the UMLT property of the gate polynomials, $G(C) \cup B(X_0)$ defines a D-Gröbner basis [4] for $J(C) \subseteq \mathbb{Z}[X]$ [24]. We further showed in [24] that $J(C) = \langle G(C) \cup B(X) \rangle \subseteq \mathbb{Z}[X]$, i.e., $J(C)$ contains all Boolean value constraints for $x \in X$. Thus the correctness of the circuit can be established by reducing $\mathcal{L}$ by the gate polynomials and all Boolean value constraints and checking whether the result is zero.

It was shown in [30] that simply reducing the specification by $G(C) \cup B(X)$ leads to large intermediate reduction results. Thus, we developed preprocessing techniques based on variable elimination [24]. Typical components in multipliers are full- and half-adders, which are used to add three resp. two bits and produce a two-bit output c, s. The specification is $-2c - s + x + y + z = 0$ for a full-adder and $-2c - s + x + y = 0$ for a half-adder, with x, y, z representing the inputs. We include these specifications in the D-Gröbner basis by eliminating the internal variables of the full- and half-adders in C. After preprocessing $\mathcal{L}$ is reduced by the rewritten D-Gröbner basis $G(C)'$ until completion.

However, parts of the multiplier, more precisely final stage adders that are generate-and-propagate (GP) adders [36], are hard to verify using computer algebra. Contrarily, equivalence checking of adder circuits is easy for SAT solving. Hence, we combine SAT solving and computer algebra and our verification tool AMulet automatically replaces the complex GP adders by simple ripple-carry adders [24]. The correctness of the replacement is verified by SAT solvers and the rewritten multiplier is verified using computer algebra techniques. We generate DRUP proofs in SAT solvers and PAC proofs in AMulet. These proofs can be merged into one single PAC proof [25].

2.2 Algebraic proof systems

In the following we introduce algebraic proof formats, which are able to generate proof certificates using algebraic reasoning methods. Algebraic proof systems typically reason over polynomials in $\mathbb{K}[X]$, where $\mathbb{K}$ is a field and the variables X represent Boolean values. The aim of an algebraic proof is to derive whether a polynomial f can be derived from a given set of polynomials $G = \{g_1, \ldots, g_l\} \subseteq \mathbb{K}[X]$ together with the Boolean value constraints $B(X) = \{x_i^2 - x_i \mid x_i \in X\}$. In algebraic terms this means to show that the polynomial $f \in \langle G \cup B(X) \rangle$.

The first proof system we consider is the *polynomial calculus* (PC) [10]. A proof in PC is a sequence of proof rules $P = (p_1, \ldots, p_m)$, with $p_i \in \mathbb{K}[X]$ and $p_m = f$. Each rule has the following form that model the properties of an ideal:

Axiom	$\dfrac{}{p}$	$p \in G \cup B(X)$
Addition	$\dfrac{p \quad q}{p + q}$	p, q both appear in P
Multiplication	$\dfrac{p}{qp}$	p appears earlier in P, $q \in \mathbb{K}[X]$

The following metrics for PC are common in proof complexity, e.g. in [22,34]:

Definition 2. *Let* $\deg(p)$ *be the degree of a polynomial* p*. The degree of a PC proof* P *is the maximum degree of any proof rule* p_i*, i.e.,* $\deg(P) = \max\{\deg(p_i)\}$.

Definition 3. *The length of a PC proof* P *is defined as the maximum number of proof rules, i.e.* $\text{length}(P) = m$.

Definition 4. *Let* $\text{msize}(p)$ *denote the number of monomials in a polynomial* p*. The size of a PC proof* P *is the number of monomials in all proof rules* p_i*, i.e.,*

$$\text{size}(P) = \sum_{i=1}^{m} \text{msize}(p_i).$$

However, PC proofs cannot be checked efficiently, as the sequence of proof rules only contains the conclusion polynomials of each proof rule. Thus we modified PC in [23,37] and extended PC by adding information on the derivation of each p_i, yielding the practical polynomial algebraic calculus (PAC).

Furthermore, in our application with $G = G(C)$, all polynomials in G have UMLT. Thus we were able to generalize the soundness and completeness arguments of PC to polynomial rings $R[X]$ over commutative rings R with unity [24], thus also to $\mathbb{Z}[X]$. Additionally, we treat the Boolean value constraints implicitly, i.e., we consider proofs in the ring $\mathbb{Z}[X]/\langle B(X)\rangle$ to admit shorter proofs [23,27]..

The metrics degree, length, and size can be directly applied to PAC proofs. PAC proofs can be checked using our proof checkers PACHECK or PASTÈQUE [23, 27]. The proof checkers read the given set of polynomials $G \cup B(X)$ and verify the correctness of each proof line by checking whether the necessary conditions are fulfilled. We furthermore check whether it holds for one proof rule that $p_i = f$.

The *Nullstellensatz proof system* [3] derives whether a polynomial $f \in \mathbb{K}[X]$ can be represented as a linear combination from a given set of polynomials $G = \{g_1, \ldots, g_l\} \subseteq \mathbb{K}[X]$ and the Boolean value constraints $B(X)$. That is, an NSS proof for a given polynomial f and a set of polynomials G is an equality

$$\sum_{i=1}^{l} h_i g_i + \sum_{x_j \in X} r_j(x_j^2 - x_j) = f, \text{ for } h_i, r_j \in \mathbb{K}[X]. \tag{2}$$

By the same arguments given for PAC [24], we are able to generalize the soundness and completeness arguments of NSS proofs to rings $R[X]$ for our application where $G = G(C)$ has UMLT. We consider $R = \mathbb{Z}$ and again treat the Boolean value constraints implicitly to yield shorter proofs. Thus, the NSS proof we consider for a given polynomial $f \in \mathbb{Z}[X]/\langle B(X)\rangle$ and a set of polynomials $G = \{g_1, \ldots, g_l\} \subseteq \mathbb{Z}[X]/\langle B(X)\rangle$ is an equality P, such that

$$\sum_{i=1}^{l} h_i g_i = f \in \mathbb{Z}[X]/\langle B(X)\rangle, \tag{3}$$

with $h_i \in \mathbb{Z}[X]/\langle B(X)\rangle$. We call g_i the *base of the NSS proof* and h_i *co-factors*.

The following metrics for NSS are common in proof complexity, e.g. in [1,15]:

Definition 5. *The degree of an NSS proof P is* $\max\{\deg(h_i g_i)\}$.

Definition 6. *The size of an NSS proof P is given as*

$$\text{size}(P) = \sum_{i=0}^{l} \text{msize}(h_i)\,\text{msize}(g_i).$$

A further metric is the representation size that measures the total number of monomials in the polynomials g_i and the co-factors h_i. As the name indicates, it estimates the number of monomials needed to write down an NSS proof.

Definition 7. *The representation size of an NSS proof P is given as*

$$\text{repsize}(P) = \sum_{i=0}^{l} (\text{msize}(h_i) + \text{msize}(g_i)).$$

Checking NSS proofs seems straightforward as we simply need to expand the products $h_i g_i$, calculate the sum, and compare the derived polynomial to the given target polynomial f. However, we discuss practical issues of proof checking in Sect. 5, where we introduce our proof checker Nuss-Checker.

3 Proof Generation

In this section we discuss how NSS proofs can be generated in our verification tool AMulet [24]. We introduced in Sect. 2 that we distinguish two phases during verification of multipliers. In the preprocessing step we eliminate variables from the induced D-Gröbner basis $G(C)$ to gain a simpler polynomial representation $G(C)'$. In the second step the specification is reduced by the rewritten D-Gröbner basis $G(C)'$ to determine whether the given circuit is correct. Both phases have to be included in the NSS proof to yield a representation of the specification $\mathcal{L}$ as a linear combination of the original gate constraints $G(C) \in \mathbb{Z}[X]/\langle B(X)\rangle$.

AMulet reads the given AIG, determines a reverse topological term ordering and encodes each AIG node by a corresponding polynomial to derive the set of gate constraints $G(C)$. All polynomials from $G(C)$ are kept in the memory even if they are removed from the D-Gröbner basis during preprocessing.

In the preprocessing step, we repeatedly eliminate all variables $v \in X \setminus X_0$ from $G(C)$ that occur in the tail of only one polynomial, cf. Sect. 4.2. in [26]. Let $p_v \in G(C)$ such that $\text{lt}(p_v) = v$. Since $G(C)$ has UMLT and $v \notin X_0$, such a p_v exists. All polynomials $p \in G(C) \setminus \{p\}$, with $v \in p$ are reduced by p_v to remove v from p. The reduction algorithm is depicted in Alg. 1 and returns polynomials h, $r \in \mathbb{Z}[X]/\langle B(X)\rangle$ such that $p + hp_v = r \in \mathbb{Z}[X]/\langle B(X)\rangle$. In contrast to more general polynomial division/reduction algorithms we use the fact in Alg. 1 that $\text{lm}(p_v) = -v$.

We replace the polynomial p by the calculated remainder r, and remove p_v from the D-Gröbner basis [24]. To keep track of the rewriting steps we want to store information on the derivation of the rewritten polynomials r.

Algorithm 1: Reduction(p, p_v, v)

Input : Polynomials p, $p_v \in \mathbb{Z}[X]/\langle B(X)\rangle$, $\mathrm{lm}(p_v) = -v$
Output : Polynomials h, $r \in \mathbb{Z}[X]/\langle B(X)\rangle$ such that $p + hp_v = r$

$t \leftarrow p, r \leftarrow p, h \leftarrow 0$;
while $t \neq 0$ **do**
  if $v \in \mathrm{lt}(t)$ **then**
    $h = h + \mathrm{lm}(t)/v$;
    $r = r + p_v\,\mathrm{lm}(t)/v \mod \langle B(X)\rangle$;
  $t = t - \mathrm{lm}(t)$;
return h, r

Algorithm 2: Add-to-basis-representation$(p_v, h, \mathrm{base}(r))$

Input : Polynomials p_v, $h \in \mathbb{Z}[X]/\langle B(X)\rangle$, basis representation base(r)
Output : Updated basis representation base(r) such that (p_v, h) is included

if $p_v \rightarrow$ *orig* **then**
  if $(p_v, h_i) \in \mathrm{base}(r)$ *for any* h_i **then**
    $\mathrm{base}(r) \leftarrow (\mathrm{base}(r) \setminus \{(p_v, h_i)\}) \cup \{(p_v, h_i + h)\}$;
  else
    $\mathrm{base}(r) \leftarrow \mathrm{base}(r) \cup \{(p_v, h)\}$;
else
  foreach $(p_i', h_i') \in \mathrm{base}(p_v)$ **do** Add-to-basis-representation(p_i', hh_i');
return base(r)

Definition 8. *We call* $\mathrm{base}(r) = \{(p_i, q_i) \mid p_i \in G(C),\ q_i \in \mathbb{Z}[X]/\langle B(X)\rangle\}$ *the basis representation of* $r \in \mathbb{Z}[X]/\langle B(X)\rangle$*, such that* $r = \sum_{(p_i,q_i)\,\in \mathrm{base}(r)} q_i p_i$.

For the rewritten polynomial r that is derived by Alg. 1, we have to include the tuples $(p, 1)$, (p_v, h) in the basis representation base(r), cf. Alg. 2. However, we want to represent r in terms of the original gate constraints $G(C)$ only, thus we need to take into account whether the polynomial p resp. p_v are original gate constraints or whether they are rewritten, that is base$(p) \neq \{\}$.

If p_v is an original gate constraint we include the tuple (p_v, h) in base(r). If p_v does not occur in any tuple in base(r), we simply add (p_v, h) to base(r). Otherwise base(r) contains a tuple (p_v, h_i) that has to be updated to $(p_v, h_i + h)$, which corresponds to merging common factors in base(r).

If the polynomial p_v is not an original gate constraint, base$(p_v) \neq \{\}$, i.e., p_v can be written as a linear combination $p_v = h_1' p_1 + \cdots + h_l' p_l$ for some original constraints p_i and $h_i' \in \mathbb{Z}[X]/\langle B(X)\rangle$. Thus the tuple (p_v, h) corresponds to $hp_v = hh_1' p_1 + \cdots + hh_l' p_l$. We traverse through the tuples $(p_i, h_i') \in \mathrm{base}(p_v)$, multiply each of the co-factors h_i' by h and add the corresponding tuple (p_i, hh_i') to base(r). Multiplying and expanding the product hh_i may lead to an exponential blow-up in the size of the NSS proof as the following example shows.

Example 1. Consider a set of polynomials $G = \{-y_1 + (1 + x_0)y_0, -y_2 + (1 + x_1)y_1, \ldots, -y_k + (1 + x_{k-1})y_{k-1}\} \subseteq \mathbb{Z}[y_0, \ldots y_k, x_0, \ldots x_k]$ and assume we eliminate $y_1, \ldots, y_{k-1}$, yielding $-y_k + (1+x_0)(1+x_1)\ldots(1+x_{k-1})y_0$. The expanded form of the co-factor of y_0 contains 2^k monomials.

Algorithm 3: Spec-Reduction($\mathcal{L}, G(C)'$)

Input : Circuit specification $\mathcal{L} \in \mathbb{Z}[X]$, D-Gröbner basis $G(C)'$
Output : Remainder r, Basis representation base($\mathcal{L}$)

```
r ← L, base(L) ← {};
foreach g ∈ G(C)' do
    r, h ← Reduction(r, g, lt(g));
    base(L) ← Add-to-basis-representation(r, g, h, base(L));
return r, base(L)
```

Surprisingly our experiments, cf. Sect. 6, show that this blow-up does not occur in arithmetic circuit verification, rejecting our conjecture of [23].

Example 2. We demonstrate a sample run of Alg. 2. Let $G(C) = \{p_1, p_2, p_3\} \subseteq \mathbb{Z}[X]/\langle B(X)\rangle$ and $x, y, z \in \mathbb{Z}[X]/\langle B(X)\rangle$. Assume $q_1 = p_1 + xp_2$, and $q_2 = p_3 + yp_2$. Thus base$(q_1) = \{(p_1, 1), (p_2, x)\}$ and base$(q_2) = \{(p_2, y), (p_3, 1)\}$. Let $p = q_1 + zq_2$. We receive base(p) by adding $(q_1, 1)$ and (q_2, z) to base$(p) = \{\}$.

$(q_1, 1)$: Since $q_1 \notin G(C)$, we and add each tuple of base$(q_1) = \{(p_1, 1), (p_2, x)\}$ with co-factors multiplied by 1 to base(p). We gain base$(p) = \{(p_1, 1), (p_2, x)\}$.

(q_2, z): We consider base$(q_2) = \{(p_2, y), (p_3, 1)\}$ and add (p_2, yz) and (p_3, z) to base(p). Since p_3 is not yet contained in the ancestors of p, we directly add (p_3, z) to base(p). The polynomial p_2 is already contained in base(p), thus we add yz to the co-factor x of p_2 and we derive base$(p) = \{(p_1, 1), (p_2, x + yz), (p_3, z)\}$.

After preprocessing is completed, we repeatedly apply Alg. 1 and reduce the specification polynomial $\mathcal{L}$ by the rewritten D-Gröbner basis $G(C)'$. We consider the polynomials $g \in G(C)'$ in reverse topological order, such that each polynomial in $G(C)'$ has to be considered exactly once for reduction. We generate the final NSS proof by deriving a basis representation for $\mathcal{L}$. Therefore we add after each reduction step the tuple (g, h), where h is the corresponding co-factor of polynomial g, to the base representation base($\mathcal{L}$) using Alg. 2. Algorithm 3 shows the complete reduction process.

We check whether the final remainder r is zero. If so, base($\mathcal{L}$) represents an NSS proof and is printed to a file. If r is not zero, r contains only input variables $a_i, b_i \in X_0$ and can be used to generate counter-examples [23].

4 Proof Size

In this section we examine the proof complexity of the induced NSS proofs in AMulet for certain multiplier architectures. In particular we are interested in the degree and proof (representation) size. First, we examine these proof metrics for *btor-multipliers* that are generated by Boolector [35]. In this architecture AND-gates are used to produce the partial products, which are accumulated in an *array structure* using full- and half-adders. The final-stage adder is a ripple-carry adder. These multipliers are considered as "simple" multipliers, because they can be fully decomposed into full- and half-adders, cf. Fig. 2 for input bit-width 4. The AIG representation of full- and half-adders is shown in Figs. 3 and 4.

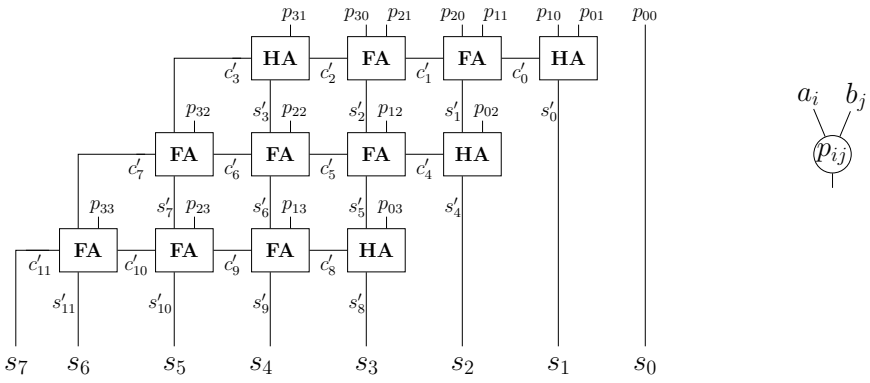
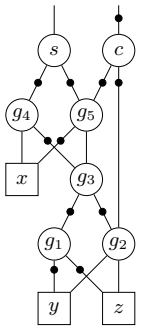


Fig. 2: The architecture of btor-multipliers for input bit-width 4.

In previous work [37] we studied the proof complexity of PAC proofs and empirically demonstrated that checking commutativity of btor-multipliers induces PAC proofs of quadratic length and cubic size. However these proofs were produced using existing computer algebra systems [41] that are not targeted for multiplier verification. In more recent work [26] we investigated the proof metrics for PAC proofs that are generated in our verification tool AMULET [24]. We formally derived that n-bit btor-multipliers generate PAC proofs with degree 3 that have a length of $16n^2 - 20n - 1$ and a proof size (cf. Def. 4) in $\mathcal{O}(n^2 \log(n))$.

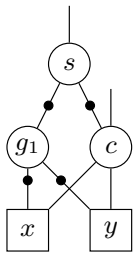
In the following we will investigate the complexity of NSS proofs that are generated by AMULET for btor-multipliers. We split the gate constraints in $G(C)$ into three categories: the *output polynomials* that link an output s_i to an output of a full- or half-adder adder, that is $-s_i + s'_k$ or $-s_i - s'_k + 1$ depending on the sign of s'_k. For example, the multiplier in Fig. 2 induces the output polynomial $-s_3 + s'_8$. Furthermore, we consider *polynomials representing partial products*, i.e., $-p_{ij} + a_i b_j$. All remaining polynomials in $G(C)$ are induced from the full- and half-adders in the circuit, i.e., *the internal adder polynomials*.

We are able to express the specification of each full- and half-adder, cf. Sect. 2 as a linear combination of the internal adder polynomials. Figure 3 shows the AIG representing a full-adder, as it occurs in btor-multipliers. Depending on the position of the full-adder in the multiplier, the sign of the inputs x, y, and z may be inverted and thus internal variables of the full-adder are negated, which affects the proof size. The full-adder in Fig. 3 represents the full-adder in btor-multipliers that yields the largest NSS proofs (input x and output c are inverted). We use the proof size of these full-adders to estimate an upper bound of the proof size. The corresponding gate polynomials can be seen on the right side of Fig. 3 together with the co-factors that are induced in AMULET. Expanding the linear combination yields the specification $-2(1-c) + s + (1-x) - y - z$. Figure 4 shows the same result for a half-adder resulting in $-2c - s + x + y$. From the polynomials in Figs. 3 and 4 we are able to derive the following lemmas.



gate constraints	co-factors
$-c + g_5g_2 - g_5 - g_2 + 1$	-2
$-s + g_4g_5 - g_4 - g_5 + 1$	1
$-g_4 - g_3x + x$	$g_5 - 1$
$-g_5 - g_3x + g_3$	$g_3x - 2g_2 + x + 1$
$-g_3 + g_1g_2 - g_1 - g_2 + 1$	$2g_2x - 2g_2 + 1$
$-g_1 + yz - y - z + 1$	$g_2 - 1$
$-g_2 + yz$	$yz - y - z + 2$

Fig. 3: Full-adder architecture in btor-multipliers.



gate constraints	co-factors
$-s + g_1c - g_1 - c + 1$	1
$-g_1 + xy - x - y + 1$	$c - 1$
$-c + xy$	$xy - x - y + 2$

Fig. 4: Half-adder architecture in btor-multipliers.

Lemma 1. *The NSS proof generated in* AMULET *for a half-adder has maximum size 61 and maximum representation size 45. The NSS proof for a half-adder has maximum size 23 and maximum representation size 19.*

Proof. Figures 3 and 4 show the representation of the full-adder and half-adders that occur in btor-multipliers that maximize the NSS proof size. Furthermore the induced co-factors in AMULET are shown. We simply count the number of monomials in the polynomials and use the definition of proof (representation) size, cf. Defs. 6 and 7 to yield the desired results.

Lemma 2. *The degree of an NSS proof for a full- or half-adder is 3.*

Proof. It can be seen in Figs. 3 and 4 that multiplying each of the gate polynomials by the corresponding co-factor yields degree at most 3 in $\mathbb{Z}[X]/\langle B(X)\rangle$.

We use the full- and half-adder specifications to derive a concise NSS proof. That is, we want to find co-factors, such that we are able to express the specification $\mathcal{L}$ cf. Eqn. 1 as a linear combination of the output polynomials, adder specifications and the polynomials that represent partial products.

It is easy to see that all the output polynomials, i.e., $-s_i + s'_k$ or $-s_i - s'_k + 1$ need to be multiplied by the corresponding constant 2^i, because neither the internal adder polynomials nor the polynomials representing partial products contain any output variable s_i of the multiplier. Furthermore, since all adder specifications are linear, we multiply these polynomials by constants to cancel output variables of an adder that are input to another adder. For example,

the multiplier of Fig. 2 induces the polynomials $-2c'_{11} - s'_{11} + c'_7 + p_{33} + c'_{10}$, $-2c'_{10} - s'_{10} + s'_7 + p_{23} + c'_9$. We multiply the first polynomial by two to cancel the monomials containing c'_{10}. It follows by the same arguments that we only need to multiply the polynomials $-p_{ij} + a_i b_j$ by constants to cancel the variables p_{ij}. Using these observations and the following lemmas that are derived in [26] we are able to derive quadratic bounds for the proof (representation) size of btor-multipliers in Thm. 1 and Thm. 2.

Lemma 3 (Lemma 2 in [26]). *Let C be a btor-multiplier of input bit-width n. Then C contains n half-adders and $n^2 - 2n$ full-adders.*

Theorem 1. *The proof size of n-bit btor-multipliers produced in* AMULET *is bounded by $63n^2 - 93n$.*

Proof. Using Lemma 3, we derive that the proof size for all full- and half-adder specifications is at most $23n + 61n^2 - 122n = 61n^2 - 99n$. These specifications are only multiplied by constants during reduction, thus reduction has no effect on the proof size. The $2n$ polynomials representing the circuit outputs are multiplied by constants, thus each polynomial contributes at most 3 monomials. Each of the n^2 polynomials representing partial products is also multiplied by a constant, adding 2 monomials to the proof size. Collecting the results leads to a proof size of $61n^2 - 99n + 6n + 2n^2 = 63n^2 - 93n$.

Theorem 2. *The proof representation size of n-bit btor-multipliers produced in* AMULET *is bounded by $48n^2 - 63n$.*

Proof. Using Lemma 3 and multiplying the co-factors by appropriate constants we derive that the proof representation size for all full- and half-adder specifications is at most $19n + 45n^2 - 90n = 45n^2 - 71n$. The $2n$ polynomials representing the circuit outputs are multiplied by constants. Thus for each of the $2n$ products we derive a representation size 4. Each of the n^2 polynomials representing partial products is also multiplied by a constant, adding 3 monomials to the proof representation size. Collecting the results leads to a proof representation size of $45n^2 - 71n + 8n + 3n^2 = 48n^2 - 63n$.

Theorem 3. *The degree of the NSS proof of n-bit btor-multipliers is* 3.

Proof. It follows from Lemma 2 that the degree of the NSS proof for an adder specification is 3. This linear adder specification is only multiplied by constants in the NSS proof for btor-multipliers. Furthermore, the degree of the output polynomials is 1 and the degree of the polynomials representing the partial products is 2, and both are multiplied only by constant factors in the NSS proof. Thus the maximum degree of a polynomial product in the NSS is 3.

Figure 5 shows the proof (representation) size together with the derived bounds of Thm. 1 and Thm. 2 for btor-multipliers with an input bit-width n in $[4, 128]$. The absolute error of the bounds can be seen in Fig. 6, which empirically indicates that the difference between the upper bound and the real proof size is in $\mathcal{O}(n)$, giving us a precise bound on the coefficient of the quadratic terms.

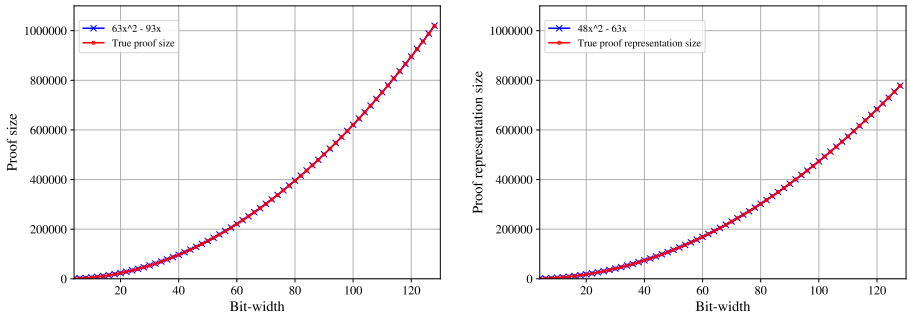


Fig. 5: Proof size (left) and proof representation size (right) for btor-multipliers.

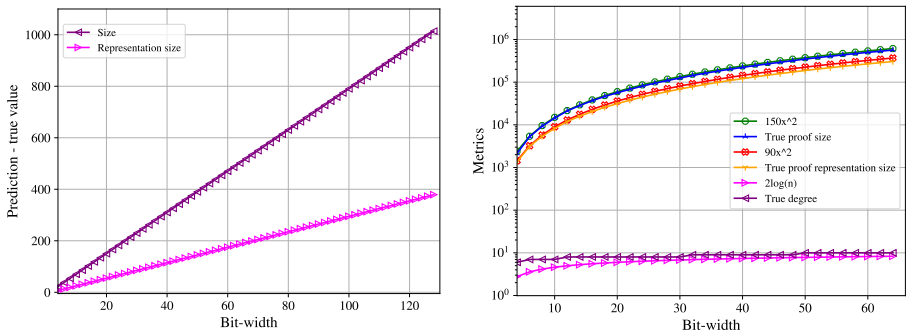


Fig. 6: Absolute error of the estimated bounds for proof (representation) size (left). Empirical evaluation of proof metrics for bp-wt-rc–multipliers (right).

The second multiplier architecture we consider are the complex *bp-wt-rc*-multipliers that are part of the AOKI benchmarks [21]. These benchmarks only scale up to input bit-width 64. The bp-wt-rc–multipliers use a Booth encoding [36] to generate the partial products, which are then accumulated using a Wallace-tree. The final-stage adder is a ripple-carry adder. The abbreviations of these components "Booth encoding" – "Wallace-tree" – "ripple-carry adder" give this architecture its name. Due to their irregular structure we only give empirical evidence for the proof metrics, which can be seen in the right side of Fig. 6.

Proposition 1. *Let C be a bp-wt-rc–multiplier of input bit-width n. The degree of the NSS proof is in $\mathcal{O}(\log(n))$. The proof (representation) size is in $\mathcal{O}(n^2)$.*

5 Proof Checking

We validate the correctness of the generated NSS proofs by checking whether $\sum_{i=1}^{l} q_i p_i = \mathcal{L} \in \mathbb{Z}[X]/\langle B(X)\rangle$ for $p_i \in G(C)$, $q_i \in \mathbb{Z}[X]/\langle B(X)\rangle$. This sounds rather straightforward as theoretically we only need to multiply the original constraints p_i by the co-factors q_i and calculate the sum of the products. However, we will discuss in this section that depending on the implementation the time and maximum amount of memory that is allocated varies by orders of magnitude.

We implemented an NSS proof checker, called Nuss-Checker in C. It consists of approximately 1800 lines of code and is published [1] as open source under the MIT license. Nuss-Checker reads three input files `<input>`, `<cofact>`, and `<target>`. The file `<input>` contains the original gate constraints p_i, `<cofact>` contains the corresponding co-factors q_i in the same order. Nuss-Checker reads the files `<input>` and `<cofact>`, generates the products and then verifies that the sum of the products is equal to the polynomial given in `<target>`.

The polynomials in Nuss-Checker are internally stored as ordered linked lists of monomials. The coefficients are represented using the GMP library and the terms are ordered linked lists of variables. All internally allocated terms are shared using a hash table. We already discussed in [23] that the variable ordering has an enormous effect on the memory usage of the tool, since different variable orderings induce different terms. In the default mode Nuss-Checker orders the variables by their name using the function `strcmp`, as this minimized memory usage for our application [23]. Nuss-Checker furthermore supports to use the same variable ordering as in the given files. That is, whenever a new variable is parsed we assign an increasing numerical `level` value and sort according to this value. Both orderings `strcmp` and `level` can be applied in reverse order too.

Nuss-Checker generates the products on the fly. That is, we parse both files `<input>` and `<cofact>` simultaneously, read two polynomials q_i and p_i from each file and calculate the product $q_i p_i$.

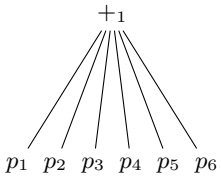
The polynomial arithmetic needed for multiplication and addition is implemented from scratch, because in the default setting we always calculate modulo the ideal $\langle B(X) \rangle$. General algorithms for polynomial arithmetic need to take exponent arithmetic over $\mathbb{Z}$ into account [38], which is not the case in our setting. Furthermore, in our previous work on PAC [37] we used modern computer algebra systems, Mathematica [41] and Singular [12], for proof checking, which turned out to be much slower than our own implemented algorithms.

Addition of two polynomials is implemented by pushing the monomials of both polynomials on a stack, which is then sorted (using Quicksort) according to the fixed term ordering and monomials with equal terms are merged to yield the final sum. Multiplication is implemented in a similar way.

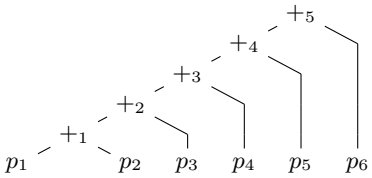
Since addition of polynomials in $\mathbb{Z}[X]$ is associative, we are able to derive different addition schemes. We experimented with four different addition patterns, which are depicted in Fig. 7 for adding six polynomials. The subscript i of "$+_i$" shows the order of the addition operation.

If we *sum up all polynomials at once*, we do not generate the intermediate addition results. Instead we push all monomials of the l products $p_i q_i$ onto one big stack. Afterwards, the monomials on the stack are sorted and merged, which corresponds to one big addition. In this addition scheme we do not compute any intermediate summands, which makes the algorithm very fast, because we sort the stack only once. However, all occurring monomials of the products are pushed on the stack and stored until the final sorting and merging, which increases the memory usage of Nuss-Checker.

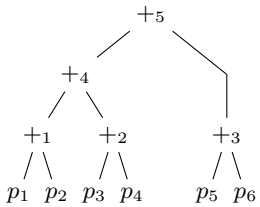
[1] `http://fmv.jku.at/nussproofs`



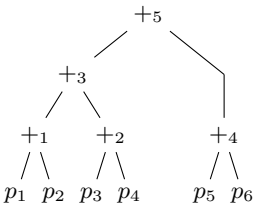
(a) Adding all polynomials at once



(b) Addition in sequence



(c) Tree structure, breadth first



(d) Tree structure, depth first

Fig. 7: Addition schemes of 6 polynomials.

If we *add up in sequence*, we only store one polynomial in the memory, and always add the lastest product $p_i q_i$. This allows for monomials to cancel, which helps to reduce the memory usage. On the other hand, in our application the target polynomial $\mathcal{L}$ contains n^2 partial products that that lead to intermediate summands of quadratic size, which slows down the checking time.

If we add up in a *tree structure with breadth first*, we add two consecutive products of the NSS proof and store the resulting sum. After parsing the proof, we have $\frac{l}{2}$ polynomials on a stack. We repeatedly iterate over the stack and always sum up two consecutive polynomials, until only one polynomial is left. This has the effect that we do not collect and carry along the n^2 partial products. However, the memory usage increases, because we store $\frac{l}{2}$ polynomials simultaneously.

In the addition scheme, where we use a *tree structure and sum up depth first*, we develop the tree on-the-fly by always adding two polynomials of the same layer as soon as possible. It may be necessary to sum up remaining intermediate polynomials that are elements of different layers, as can be seen in Fig. 7. Similar to using a tree structure with breadth first addition, we do not collect and carry along the partial products. Furthermore, we always store at most $\lceil \log(l) \rceil$ polynomials in the memory, as a binary tree with l leafs has height $\lceil \log(l) \rceil$ and we never have more polynomials than layers in the memory.

We apply the presented addition schemes on btor-multipliers, cf. Sect. 4 and it can be seen in Fig. 8 that the results compare favorably to our conjectures of checking time and memory usage for each addition scheme. However, Nuss-Checker supports all presented options for addition, with *adding up in binary tree, depth first* set as default, because for different applications, using other addition schemes may be more beneficial.

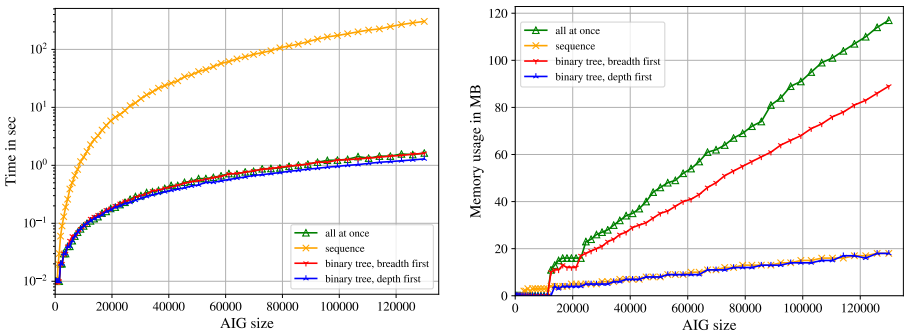


Fig. 8: Time (left) and memory usage (right) of addition schemes.

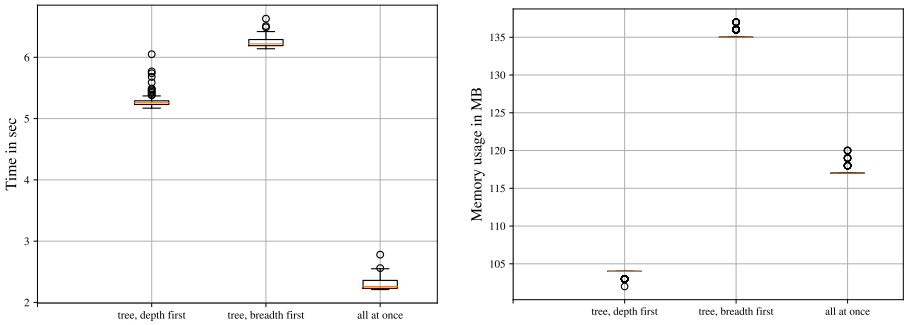


Fig. 9: Checking time (left) and memory usage (right) of shuffled NSS proofs.

For example, we shuffled the order of the polynomials in the NSS proof of 128-bit btor-multipliers 200 times and report the box-plots of the checking time and memory usage in Fig. 9. Since "adding up in sequence" always exceeded the time limit of 300 seconds, we omit its box-plot. It can be seen that the fastest addition scheme is now "all at once". However, the "tree based, depth first" approach still has the smallest memory usage.

6 Evaluation

In this section we provide experimental results for generating and checking NSS proofs for multiplier verification and we aim to provide a comprehensive comparison between PAC and NSS proofs for the selected multiplier architectures.

In our experiments we use an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 128 GB. The time is listed in rounded seconds (wall-clock time). The wall-clock time is measured from starting the tools until they are finished. Source code, benchmarks and experimental data are available at `http://fmv.jku.at/nussproofs`.

In our experiments we consider the simple btor-multipliers with an input bit-width n in $[4, 128]$ and the complex bp-wt-rc–multipliers with an input bit-width n in $[4, 64]$. These architectures are already discussed in detail in Sect. 4.

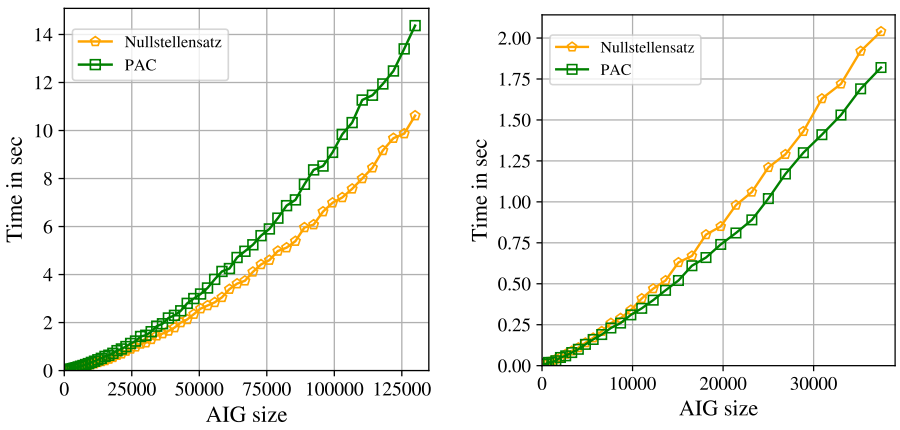


Fig. 10: Proof generation time for btor (left) and bp-wt-rc (right) multipliers.

More complex multipliers include GP adders [36]. However, in our verification approach [24] these GP adders are replaced by ripple-carry adders and only the rewritten multiplier is verified using computer algebra. Thus it suffices to consider complex multipliers that include a ripple-carry adder in this paper.

For both architectures we produce PAC proofs using AMulet as described in [23, 26], which are checked using our proof checker Pacheck. We generate NSS proofs as described in Sect. 3 and check these proofs using the default configurations of Nuss-Checker.

The results are depicted in Figs. 10–13, where we compare the proof generation and checking time as well as the size of the proof files and the memory usage of the proof checkers. In all figures we represent the measurements in terms of the size of the input AIG, i.e., the number of circuit constraints, because the number of gates in these multipliers is quadratic in the bit-width n.

Figure 10 shows the time needed to generate the NSS and PAC proofs in AMulet. It can be seen that for btor-multipliers the generation time of PAC proofs is around 30% slower than for NSS proofs. For bp-wt-rc–multipliers PAC proofs are produced slightly faster than NSS proofs.

The size of the proof files (in megabyte) is shown in Fig. 11. Depending on the multiplier architecture the size of the NSS proof file is 5–10 times smaller than the size of the PAC proof. This result is actually expected as the PAC proof includes all intermediate steps and results of generating and adding the products. In the NSS proof file we only store the co-factors without any intermediate steps.

Figure 12 depicts that NSS proofs can be checked faster than the corresponding PAC proofs. In fact, even for a btor-multiplier with input bit-width 128, where the AIG contains more than 129 000 nodes, checking the NSS proof takes around 1 second and is four times faster than checking the PAC proof. We observed that the proportion between multiplication and addition in Nuss-Checker is around 1:1.7, e.g. for 128-bit btor-multipliers 0.25 seconds are used by the multiplication function and 0.4 seconds are used by the addition operation.

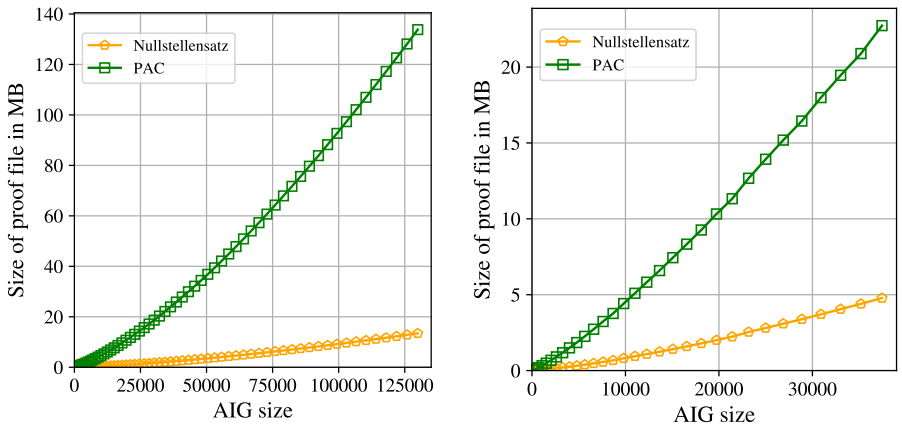


Fig. 11: Size of the proof files for btor (left) and bp-wt-rc (right) multipliers.

Last, we compare the memory usage of PACHECK and NUSS-CHECKER, i.e., the maximum amount of memory that is allocated during proof checking and it can be seen in Fig. 13 that NSS proofs need less than a third of the memory.

The AOKI benchmarks contain 192 different multiplier architectures, 168 of which can be successfully verified using AMULET. We compare the proof generation and checking time of PAC and NSS proofs for these 168 multipliers in Fig. 14. We fixed the input bit-width of all multipliers to 64. It can be seen that for multipliers that use Booth encoding to generate the partial products the generation time of NSS proofs is slightly slower than for PAC proofs. However, checking the NSS proof is almost always faster than checking PAC proofs.

7 Conclusion

In this paper we elaborated whether concise Nullstellensatz proofs can be generated to validate the results of multiplier verification using computer algebra. We discussed how Nullstellensatz proofs are developed as by-product in our verification tool AMULET. Our experiments showed that we are able to produce compact Nullstellensatz proofs that are faster to check than proof certificates based on the polynomial calculus. For simple array multipliers we formally derived quadratic bounds on the proof size for Nullstellensatz proofs. Furthermore, we presented our Nullstellensatz proof checker NUSS-CHECKER and discussed several design decisions that allow efficient proof checking.

In the future we want to further investigate the connection between polynomial calculus and Nullstellensatz for multiplier verification and want to derive possibilities to convert DRUP proofs to Nullstellensatzproofs, similar to converting DRUP proofs into PAC proofs as in [25]. Another intriguing research direction is to develop techniques that allow production of smaller Nullstellensatz proofs and connect it to SAT solving [24]. More general problems beyond the Boolean case may be also of interest [39].

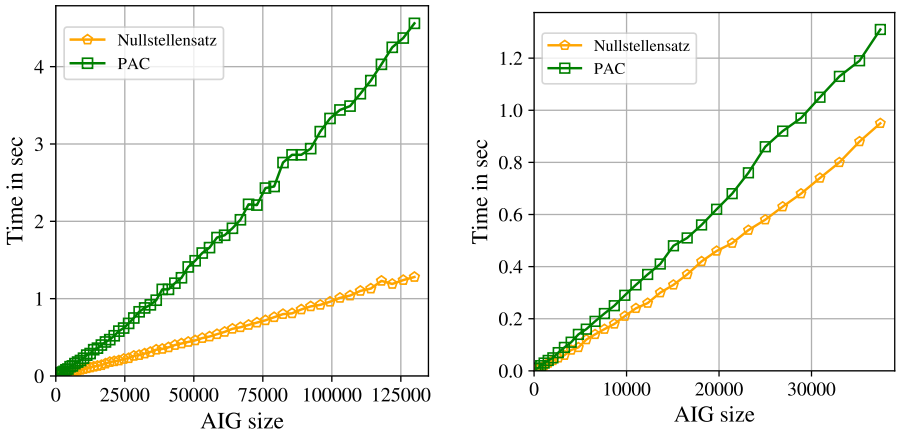


Fig. 12: Proof checking time for btor (left) and bp-wt-rc (right) multipliers.

References

1. A. Atserias and J. Ochremiak. Proof complexity meets algebra. *ACM Trans. Comput. Log.*, 20(1):1:1–1:46, 2019.
2. P. Beame, S. A. Cook, J. Edmonds, R. Impagliazzo, and T. Pitassi. The relative complexity of NP search problems. *J. Comput. Syst. Sci.*, 57(1):3–19, 1998.
3. P. Beame, R. Impagliazzo, J. Krajícek, T. Pitassi, and P. Pudlák. Lower Bounds on Hilbert's Nullstellensatz and Propositional Proofs. In *Proc. London Math. Society*, volume s3-73, pages 1–26, 1996.
4. T. Becker, V. Weispfenning, and H. Kredel. *Gröbner Bases*, volume 141 of *Graduate texts in mathematics*. Springer, 1993.
5. C. Bright, I. Kotsireas, and V. Ganesh. Applying Computer Algebra Systems and SAT Solvers to the Williamson Conjecture. *J. Symb. Comput.*, 2019. In press.
6. C. Bright, I. Kotsireas, and V. Ganesh. SAT Solvers and Computer Algebra Systems: A Powerful Combination for Mathematics. *CoRR*, abs/1907.04408, 2019.
7. B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, University of Innsbruck, 1965.
8. D. Choo, M. Soos, K. M. A. Chai, and K. S. Meel. Bosphorus: Bridging ANF and CNF solvers. In *DATE 2019*, pages 468–473. IEEE, 2019.
9. M. J. Ciesielski, T. Su, A. Yasin, and C. Yu. Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model. *IEEE TCAD*, pages 1–1, 2019. Early acces.
10. M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Groebner Basis Algorithm to Find Proofs of Unsatisfiability. In *STOC 1996*, pages 174–183. ACM, 1996.
11. L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, Jr., M. Kaufmann, and P. Schneider-Kamp. Efficient Certified RAT Verification. In *CADE-26*, volume 10395 of *LNCS*, pages 220–236. Springer, 2017.
12. W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 4-1-0 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2016.
13. A. V. Gelder. Verifying RUP Proofs of Propositional Unsatisfiability. In *ISAIM 2008*, 2008.

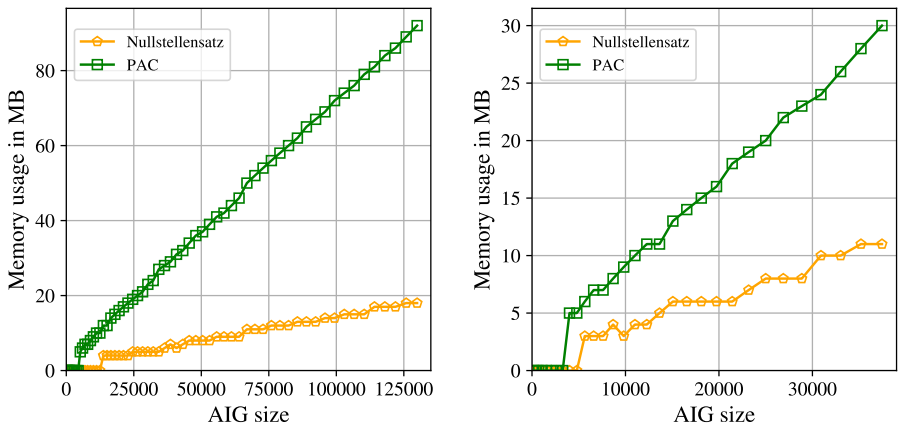


Fig. 13: Memory usage of checkers for btor (left) and bp-wt-rc (right) multipliers.

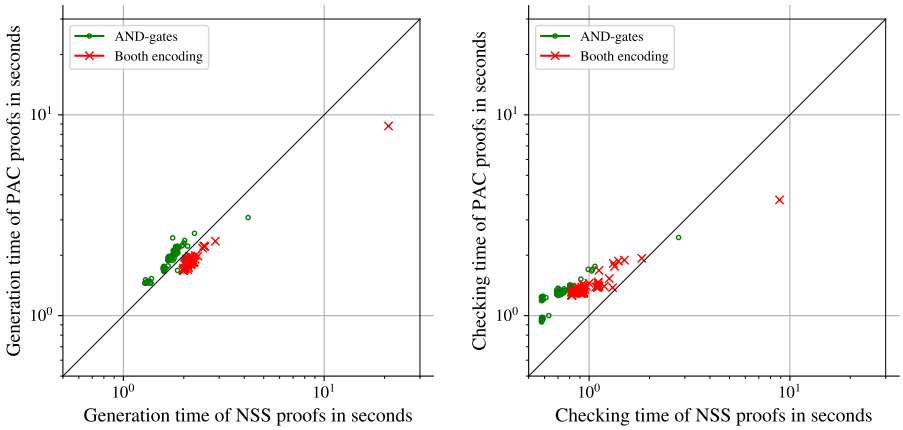


Fig. 14: Generation (left) and checking time (right) for 64-bit multipliers.

14. A. V. Gelder. Producing and verifying extremely large propositional refutations - Have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
15. D. Grigoriev, E. A. Hirsch, and D. V. Pasechnik. Exponential lower bound for static semi-algebraic proofs. In *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 2002.
16. M. J. H. Heule. Computing small unit-distance graphs with chromatic number 5. *CoRR*, abs/1805.12181, 2018.
17. M. J. H. Heule and A. Biere. Proofs for Satisfiability Problems. In *All about Proofs, Proofs for All Workshop, APPA 2014*, volume 55, pages 1–22. College Publications, 2015.
18. M. J. H. Heule, W. A. H. Jr., and N. Wetzler. Trimming while Checking Clausal Proofs. In *FMCAD 2013*, pages 181–188. IEEE, 2013.
19. M. J. H. Heule, M. Kauers, and M. Seidl. Local search for fast matrix multiplication. In *SAT 2019*, volume 11628 of *LNCS*, pages 155–163. Springer, 2019.

20. M. J. H. Heule, M. Kauers, and M. Seidl. New ways to multiply 3×3-matrices. *CoRR*, abs/1905.10192, 2019.
21. N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal Design of Arithmetic Circuits Based on Arithmetic Description Language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.
22. R. Impagliazzo, P. Pudlák, and J. Sgall. Lower bounds for the polynomial calculus and the Gröbner basis algorithm. *Computational Complexity*, 8(2):127–144, 1999.
23. D. Kaufmann. *Formal Verification of Multiplier Circuits using Computer Algebra*. PhD thesis, Informatik, Johannes Kepler University Linz, 2020.
24. D. Kaufmann, A. Biere, and M. Kauers. Verifying Large Multipliers by Combining SAT and Computer Algebra. In *FMCAD 2019*, pages 28–36. IEEE, 2019.
25. D. Kaufmann, A. Biere, and M. Kauers. From DRUP to PAC and back. In *DATE 2020*, pages 654–657. IEEE, 2020.
26. D. Kaufmann, A. Biere, and M. Kauers. SAT, Computer Algebra, Multipliers. In *Vampire 2018 and Vampire 2019*, volume 71 of *EPiC Series in Computing*, pages 1–18. EasyChair, 2020.
27. D. Kaufmann, M. Fleury, and A. Biere. Pacheck and Pastèque, Checking Practical Algebraic Calculus Proofs. In *FMCAD 2020*. IEEE, 2020. To appear.
28. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21(12):1377–1394, 2002.
29. J. Lv, P. Kalla, and F. Enescu. Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits. *IEEE TCAD*, 32(9):1409–1420, 2013.
30. A. Mahzoon, D. Große, and R. Drechsler. PolyCleaner: Clean your Polynomials before Backward Rewriting to verify Million-gate Multipliers. In *ICCAD 2018*, pages 129:1 – 129:8. ACM, 2018.
31. A. Mahzoon, D. Große, and R. Drechsler. RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. In *DAC 2019*, pages 185:1–185:6. ACM, 2019.
32. A. Mahzoon, D. Große, C. Scholl, and R. Drechsler. Towards formal verification of optimized and industrial multipliers. In *DATE 2020*, pages 544–549. IEEE, 2020.
33. O. Meir, J. Nordström, R. Robere, and S. F. de Rezende. Nullstellensatz size-degree trade-offs from reversible pebbling. *ECCC*, 137:18:1–18:16, 2019.
34. M. Miksa and J. Nordström. A Generalized Method for Proving Polynomial Calculus Degree Lower Bounds. In *Conference on Computational Complexity, CCC 2015*, volume 33 of *LIPIcs*, pages 467–487. Schloss Dagstuhl, 2015.
35. A. Niemetz, M. Preiner, C. Wolf, and A. Biere. Btor2 , BtorMC and Boolector 3.0. In *CAV 2018*, volume 10981 of *LNCS*, pages 587–595. Springer, 2018.
36. B. Parhami. *Computer Arithmetic - Algorithms and Hardware designs*. Oxford University Press, 2000.
37. D. Ritirc, A. Biere, and M. Kauers. A Practical Polynomial Calculus for Arithmetic Circuit Verification. In *SC2 2018*, pages 61–76. CEUR-WS, 2018.
38. D. S. Roche. What can (and can't) we do with sparse polynomials? In *ISSAC*, pages 25–30. ACM, 2018.
39. S. Saraf and I. Volkovich. Black-box identity testing of depth-4 multilinear circuits. *Combinatorica*, 38(5):1205–1238, 2018.
40. M. Soos and K. S. Meel. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *AAAI 2019*, pages 1592–1599. AAAI Press, 2019.
41. Wolfram Research, Inc. Mathematica, 2016. Version 10.4.